An Introduction to Object-Oriented Analysis and Design

PART V: Elaboration Iteration 3 Intermediate Topics

Dr. 邱 明
Software School of Xiamen University
mingqiu@xmu.edu.cn
Fall, 2008

Objective

- More GoF design pattern;
  - their application to the design of frameworks
- Architectural analysis;
  - documenting architecture with the N+1 view model
- Process modeling with UML activity diagrams
- Generalization and specialization
- The design of packages

Chapter 27: Iteration 3 -- Intermediate Topics

27.1. NextGen POS

- Provide failover to local services
  - When the remote services cannot be accessed.
- Provide support for POS device handling,
  - Such as the cash drawer and coin dispenser.
- Handle credit payment authorization.
- Support for persistent objects.

27.2. Monopoly

- Implement players moving around the squares of the board.
- There are now Lots, Railroads, and Utility squares.
  - If not owned, may buy it.
    - If buy it, the price is deducted from the player's money.
  - If owned by the player that landed on it, nothing happens.
  - If owned by a player other than the player that landed on it
    - pay its owner rent. The rent calculations are:
      - Lot rent is (index position) dollars;
      - Railroad rent is 25 dollars times the number of Railroads owned by the owner;
      - Utilities rent is 4 times the number shown on the dice

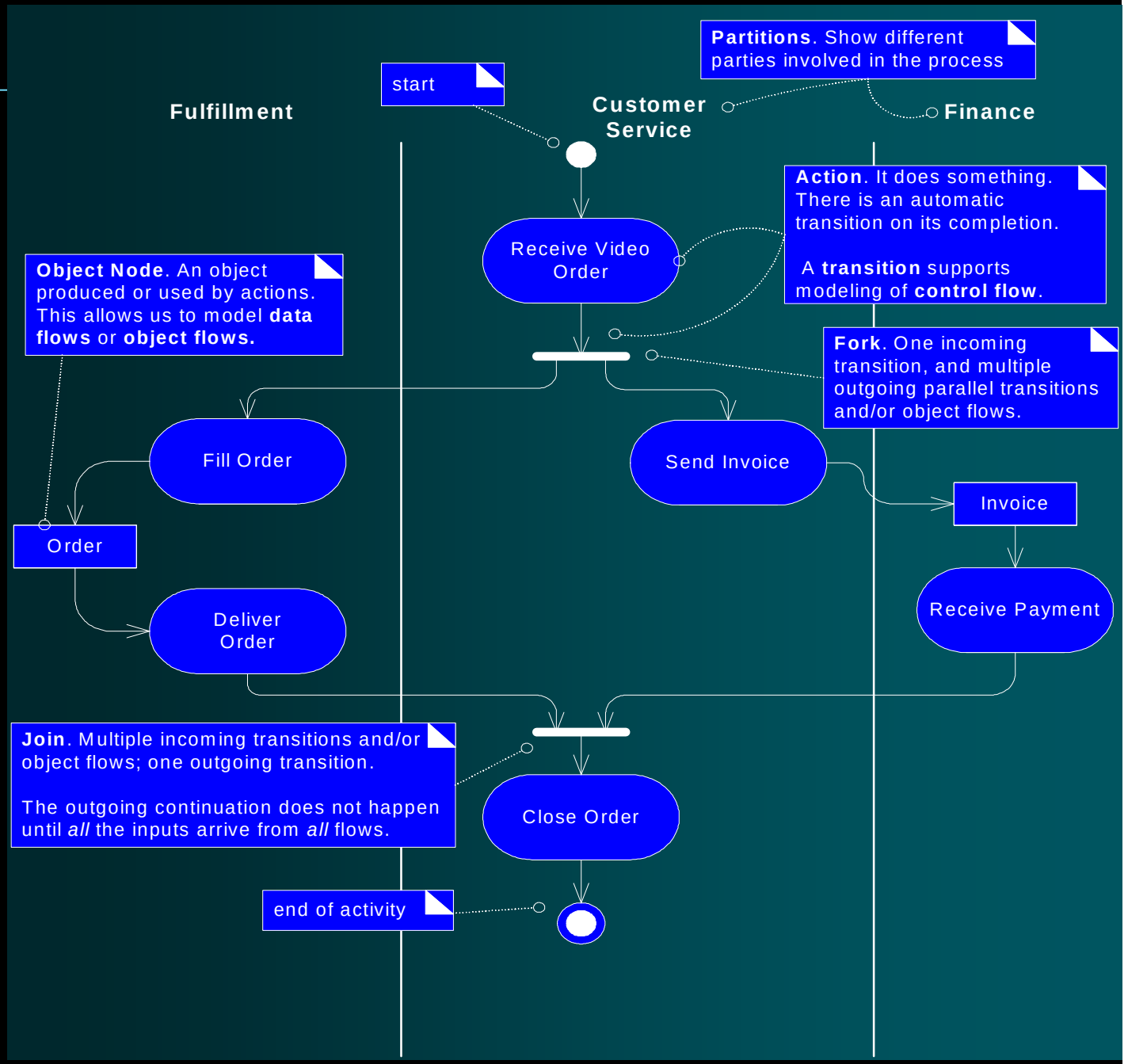
Chapter 28: UML Activity Diagrams and Modeling

Objective

- Introduce UML activity diagram notation, with examples, and various modeling applications.

28.1. Example

- Action
- Partition
- Fork
- Join
- Object Node



28.2. How to Apply Activity Diagrams?

- Business Process Modeling
- Data Flow Modeling
  - Data Flow Diagrams (DFD)
  - UML does not include DFD notation, but it can satisfy the same goals
- Concurrent Programming & Parallel Algorithm Modeling
  - Subdivide the space into blocks,
    - one parallel threads for each sub-block

28.2. How to Apply Activity Diagrams?

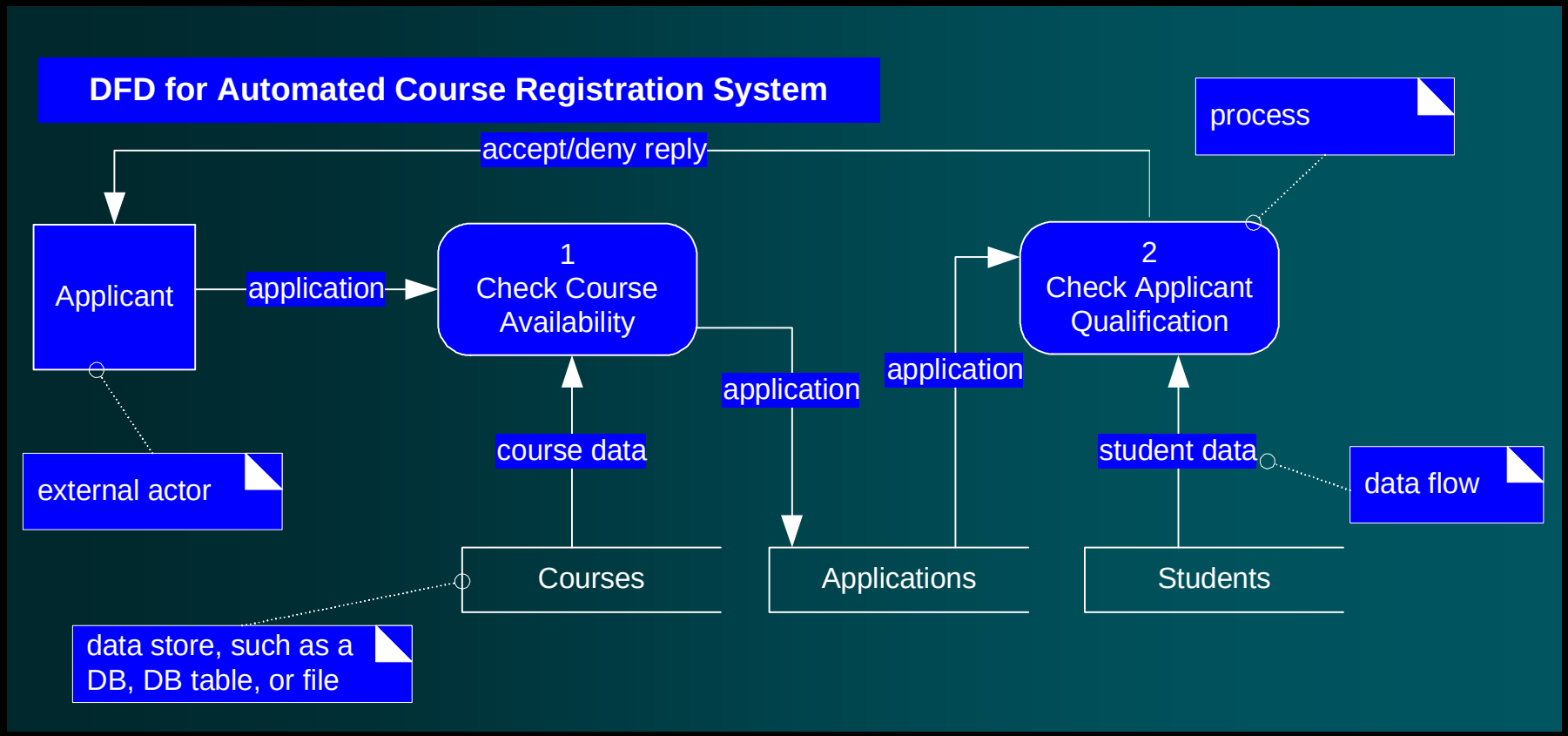
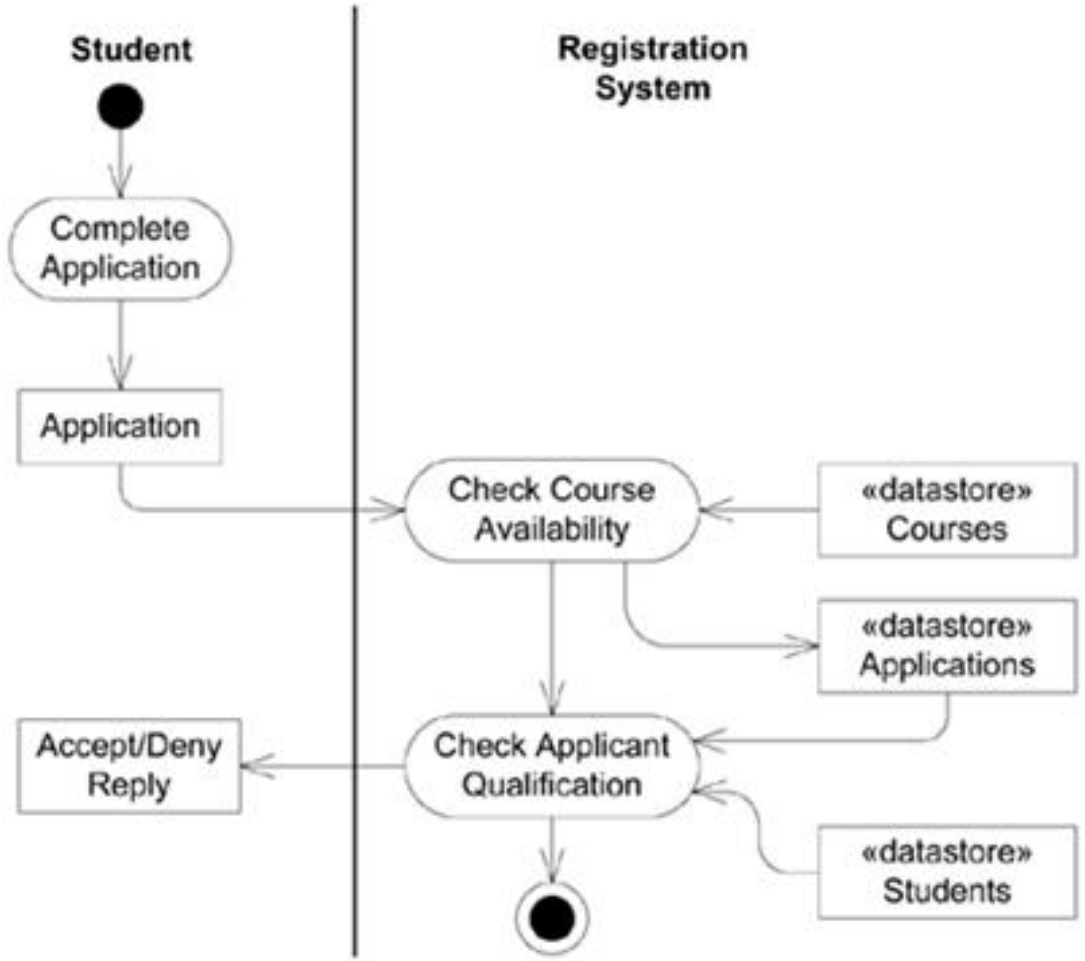


Figure 28.2. Classic DFD in Gane-Sarson notation.

28.2. How to Apply Activity Diagrams?

Figure 28.3. Applying activity diagram notation to show a data flow model.



28.3. More UML Activity Diagram Notation

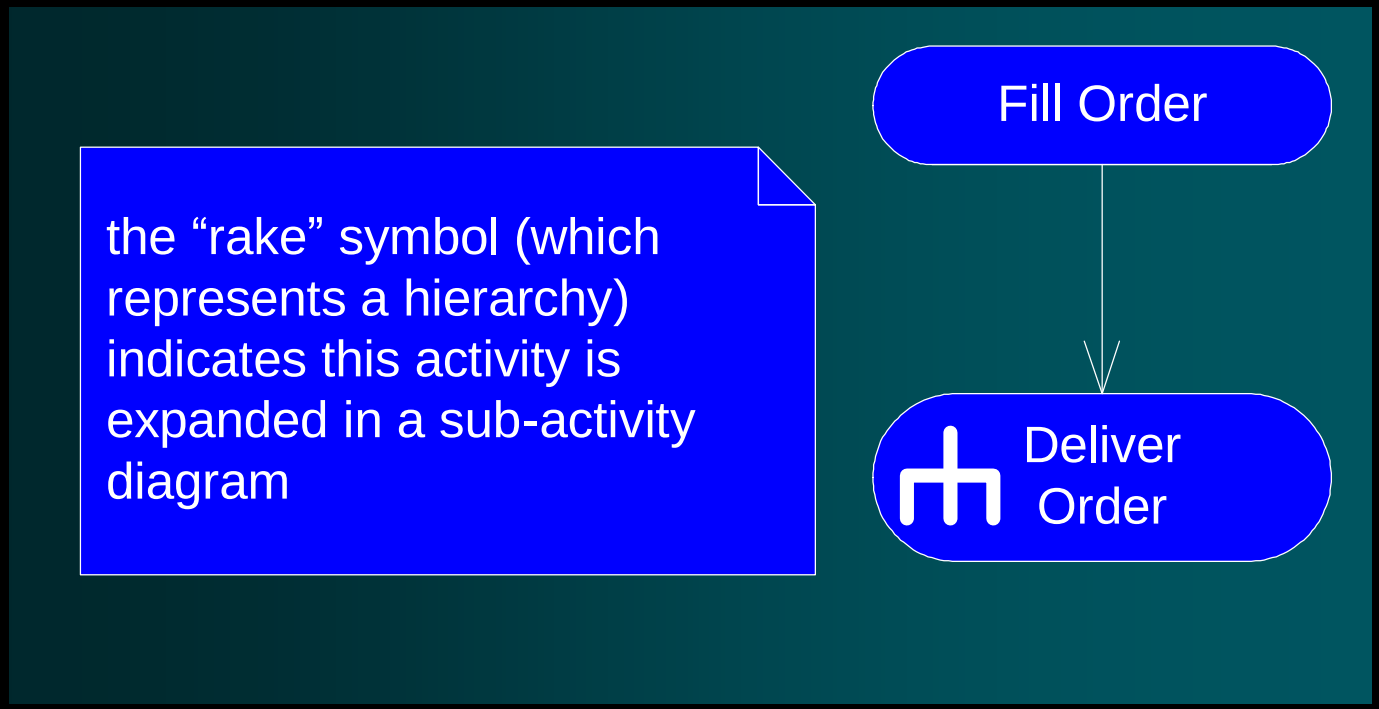


Figure 28.4. An activity will be expanded in another diagram

28.3. More UML Activity Diagram Notation

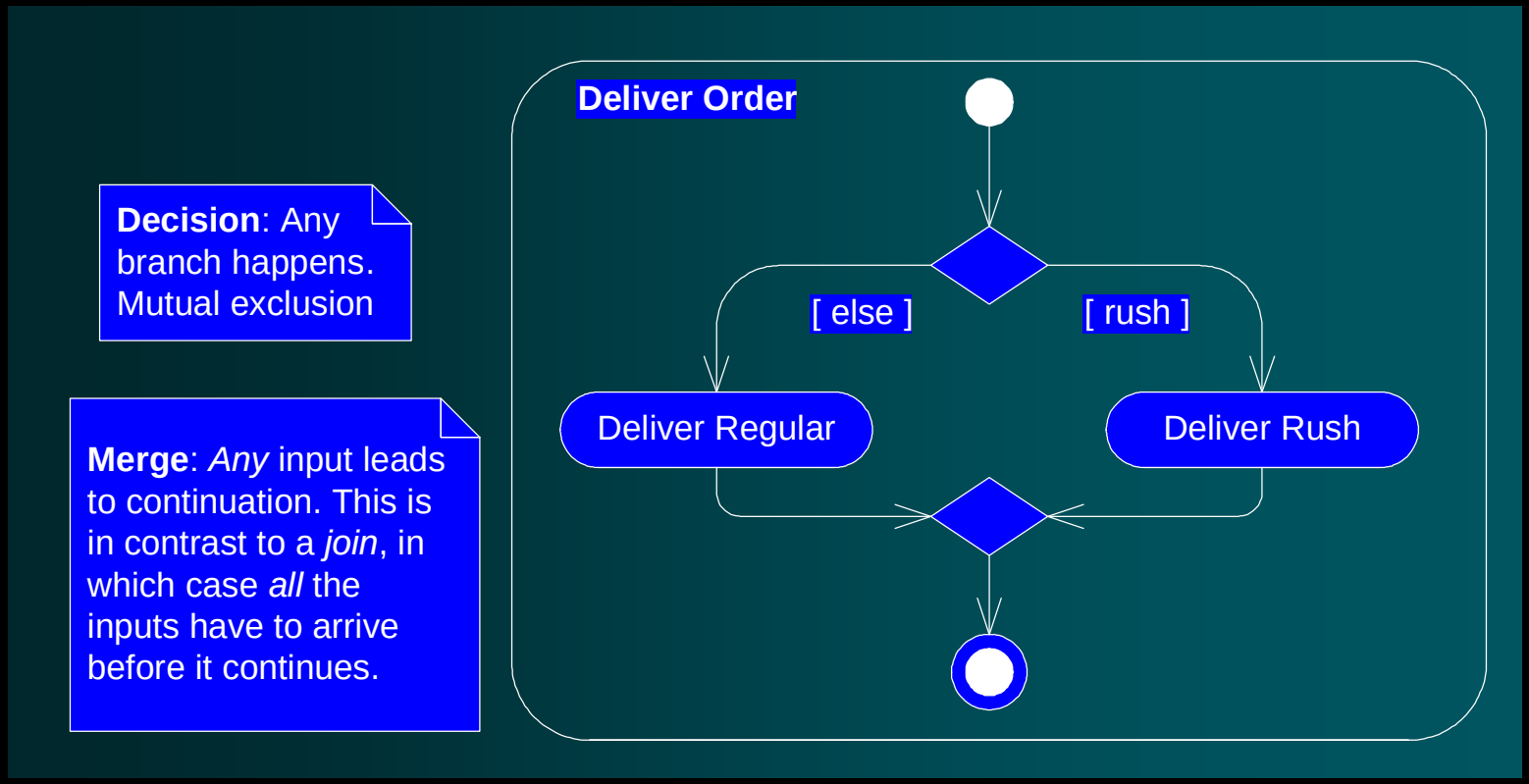
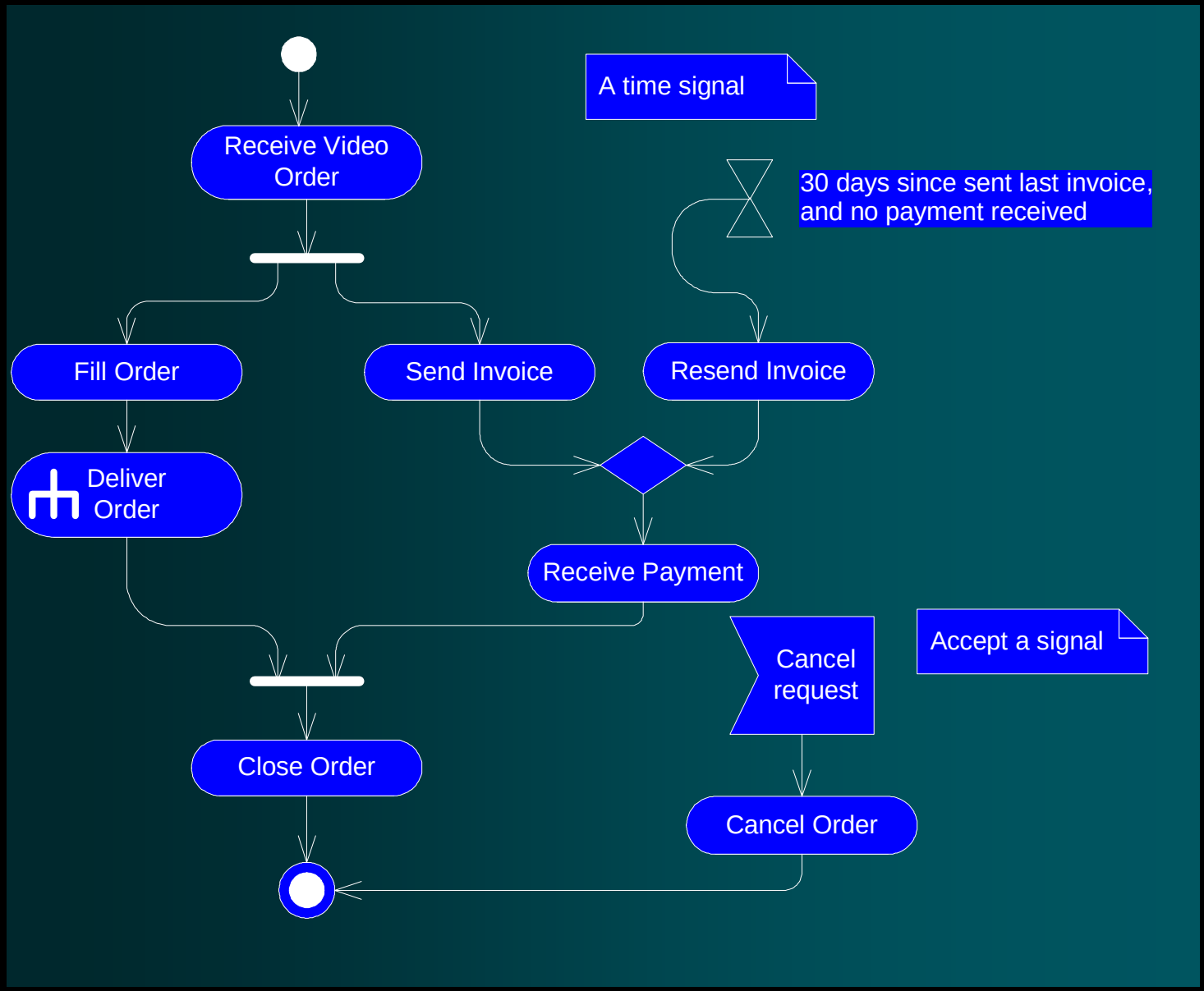


Figure 28.5. The expansion of an activity

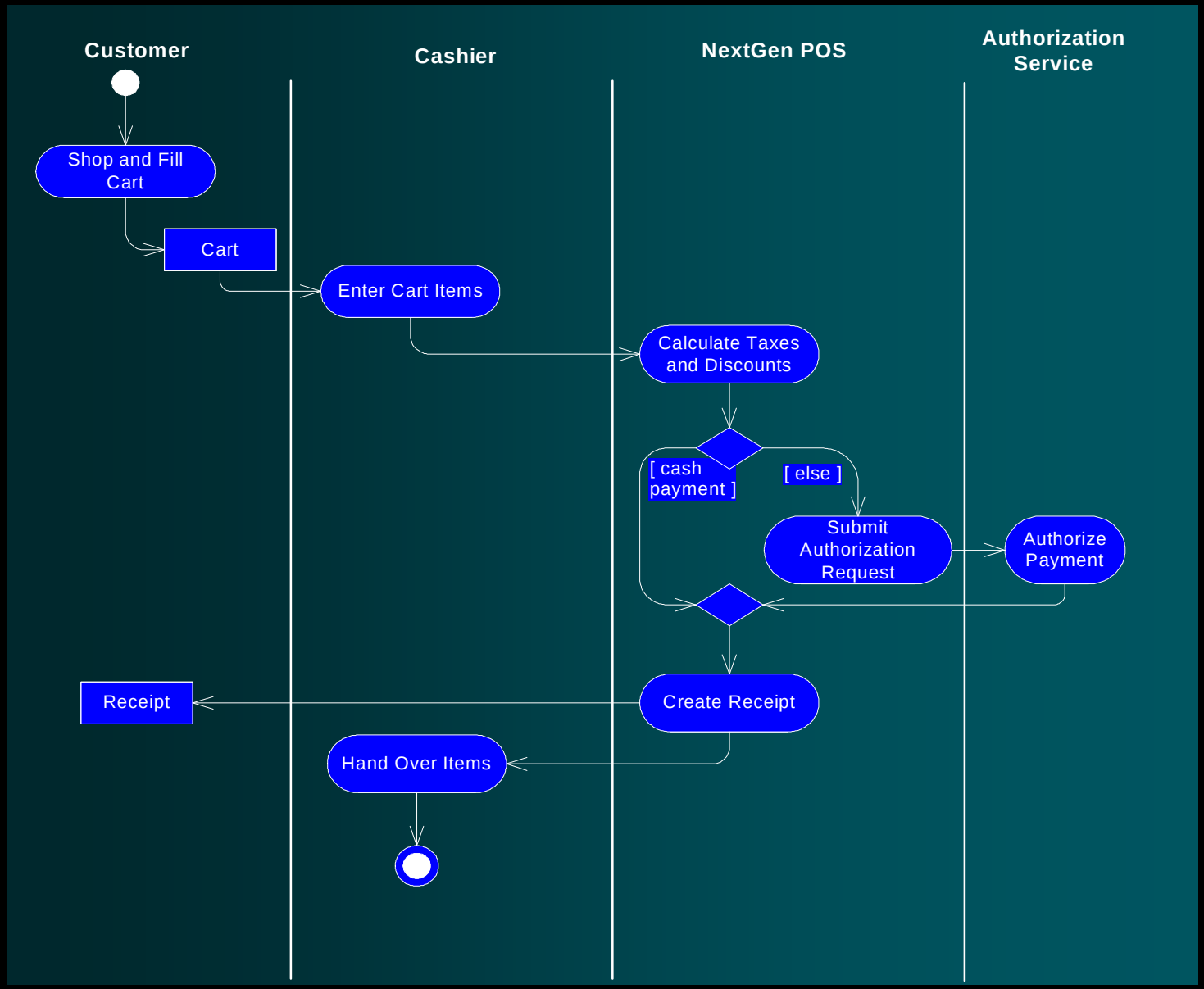
28.3. More UML Activity Diagram Notation



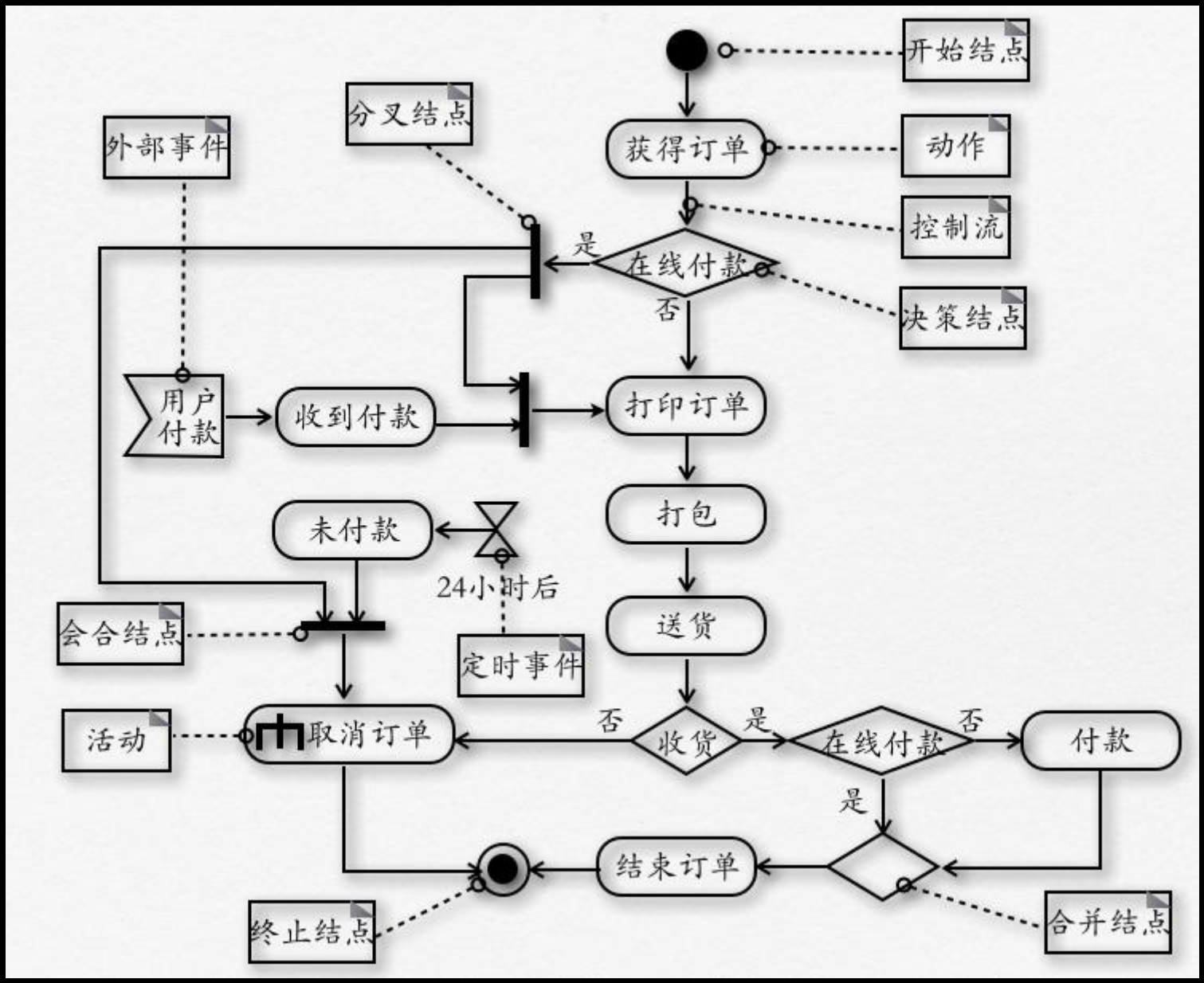
28.4. Guidelines

- Valuable for very complex processes,
  - Use-case text suffices for simple processes.
- Use "rake" notation and sub-activity diagrams to hide the complexity.
- Strive to make the level of abstraction of action nodes roughly equal within a diagram.

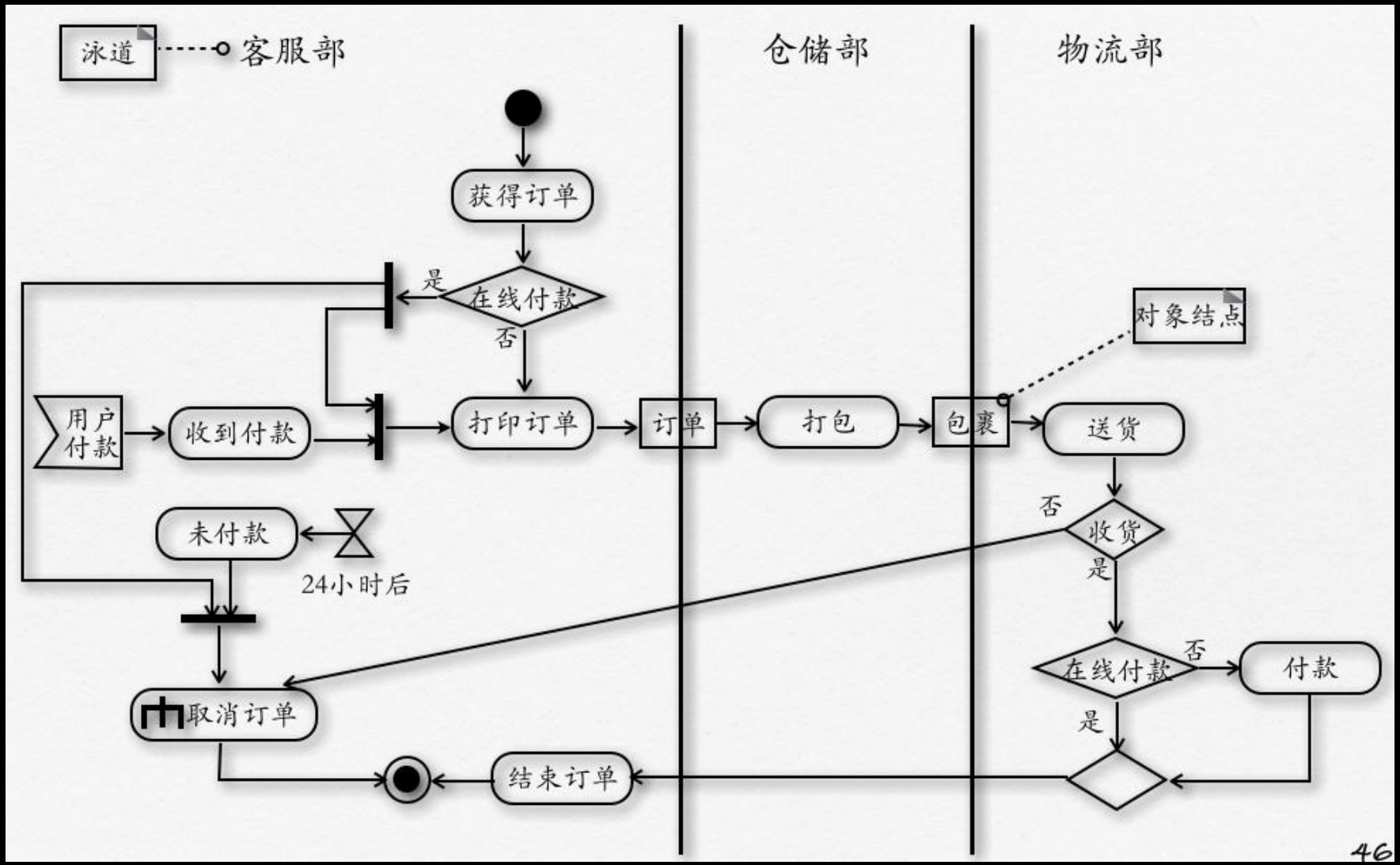
28.5. Example: NextGen Activity Diagram



28.5. Example: Online Store Activity Diagram



28.5. Example: Online Store Activity Diagram

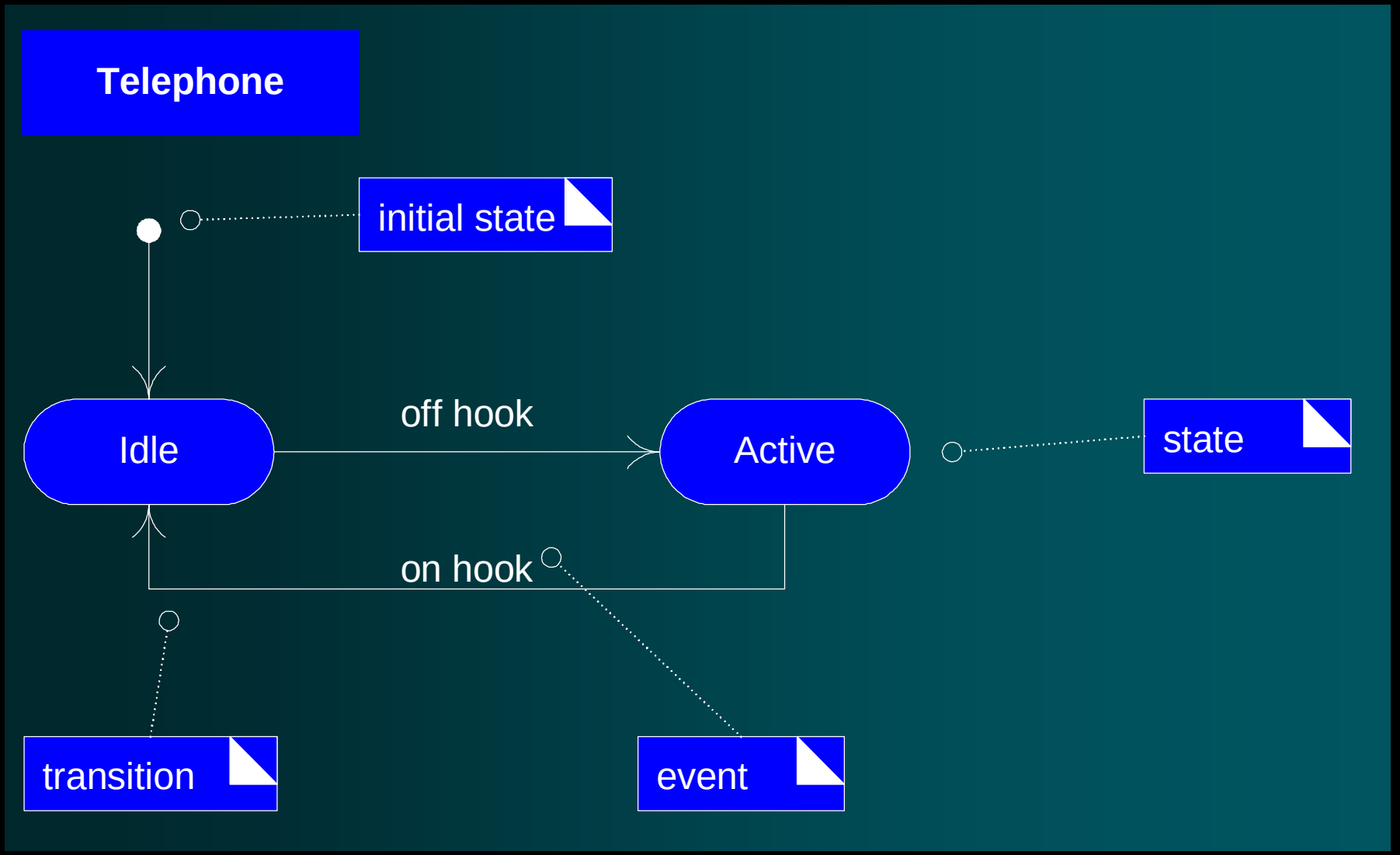


Chapter 29: UML State Machine Diagrams and Modeling

Objective

- Introduce UML state machine diagram notation, with examples, and various modeling applications.

29.1. Example


Telephone
initial state
Idle
off hook
Active
state
on hook
transition
event

29.2. Definitions: Events, States, and Transitions

- **Event**
  - a significant or noteworthy occurrence.
    - A telephone receiver is taken off the hook
- State
  - condition of an object at a moment in time (between events).
    - Idle state of a telephone
      - the time between the receiver is placed on the hook and taken off the hook.
- Transition
  - a relationship between two states that indicates
    - when an event occurs, the object moves from the prior state to the subsequent state
      - When the event "off hook" occurs, transition the telephone from the "idle" to "active" state.

29.3. How to Apply State Machine Diagrams?

- State-Independent Objects
  - always respond the same way.
- State-Dependent Objects
  - react differently to events depending on their state or mode.

29.3. How to Apply State Machine Diagrams?

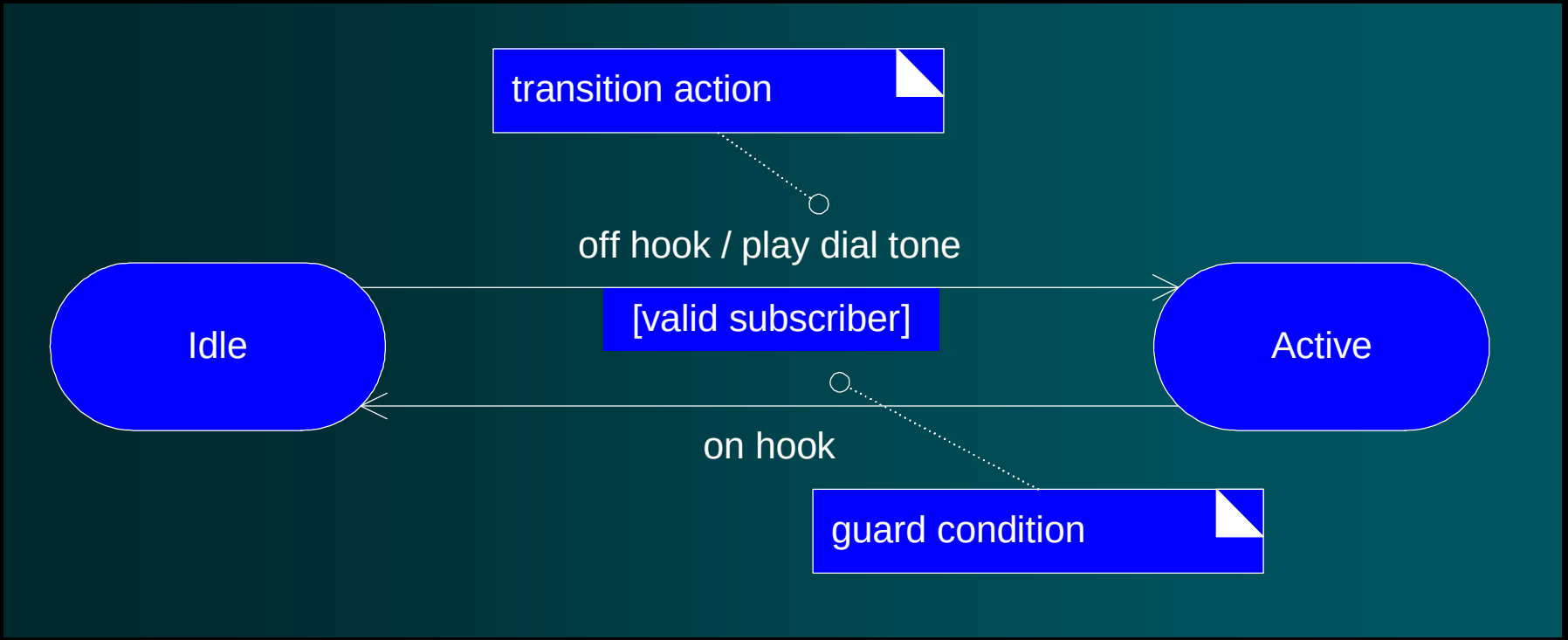
- Modeling State-dependent Objects
  - **Complex Reactive Objects**
    - model the behavior of it in response to events.
      - Physical Devices
        - Phone, car, microware oven
      - Transactions and related Business Objects
      - Role Mutators
        - objects that change their role.

29.3. How to Apply State Machine Diagrams?

- Protocols and Legal Sequence
  - model legal sequences of operations.
    - Communication Protocols
      - TCP
    - UI Page/Window Flow or Navigation
    - UI Flow Controllers or Sessions
    - Use Case System Operations
    - Individual UI Window Event Handling

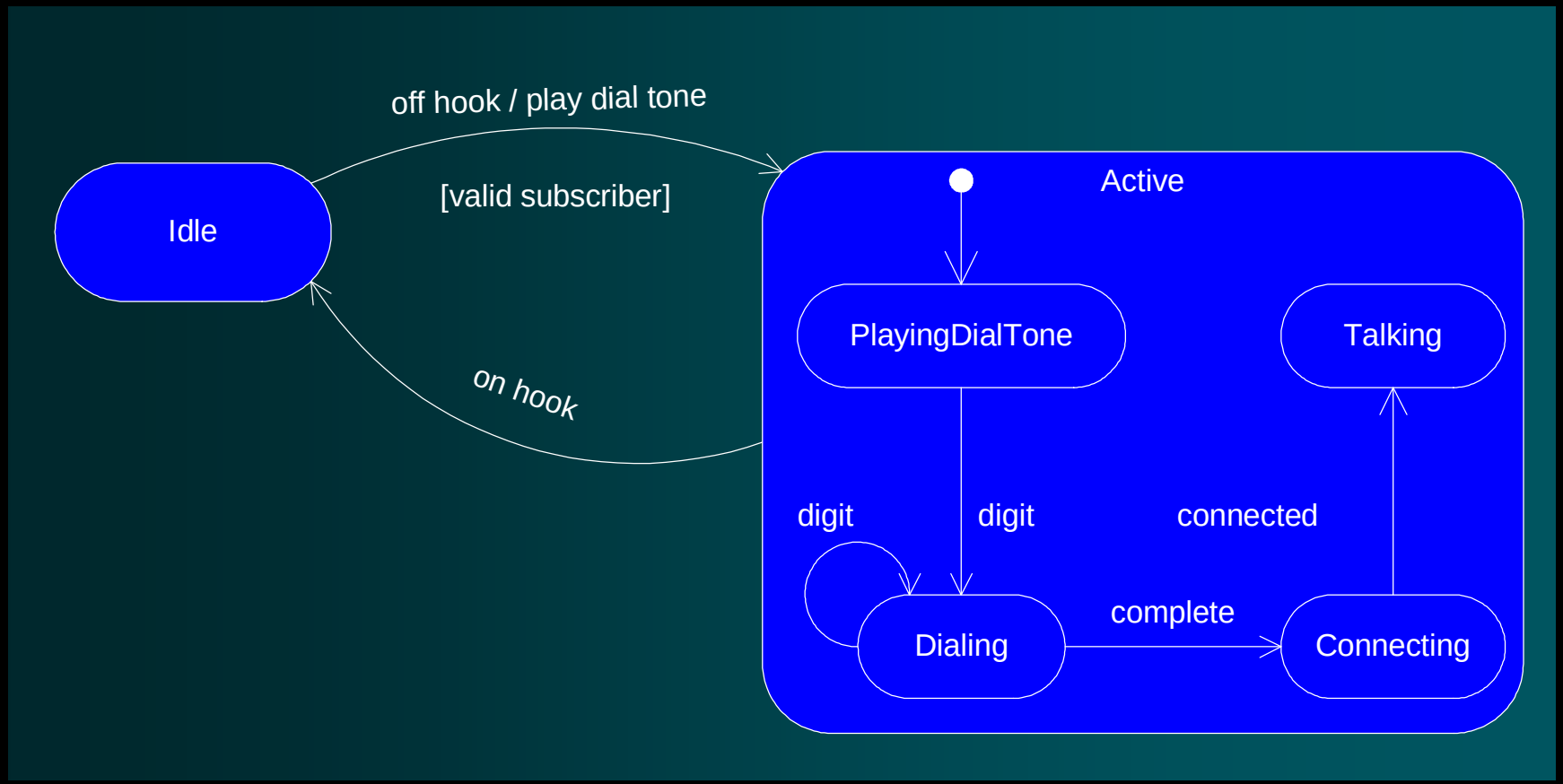
29.4. More UML State Machine Diagram Notation

w Transition Actions and Guards

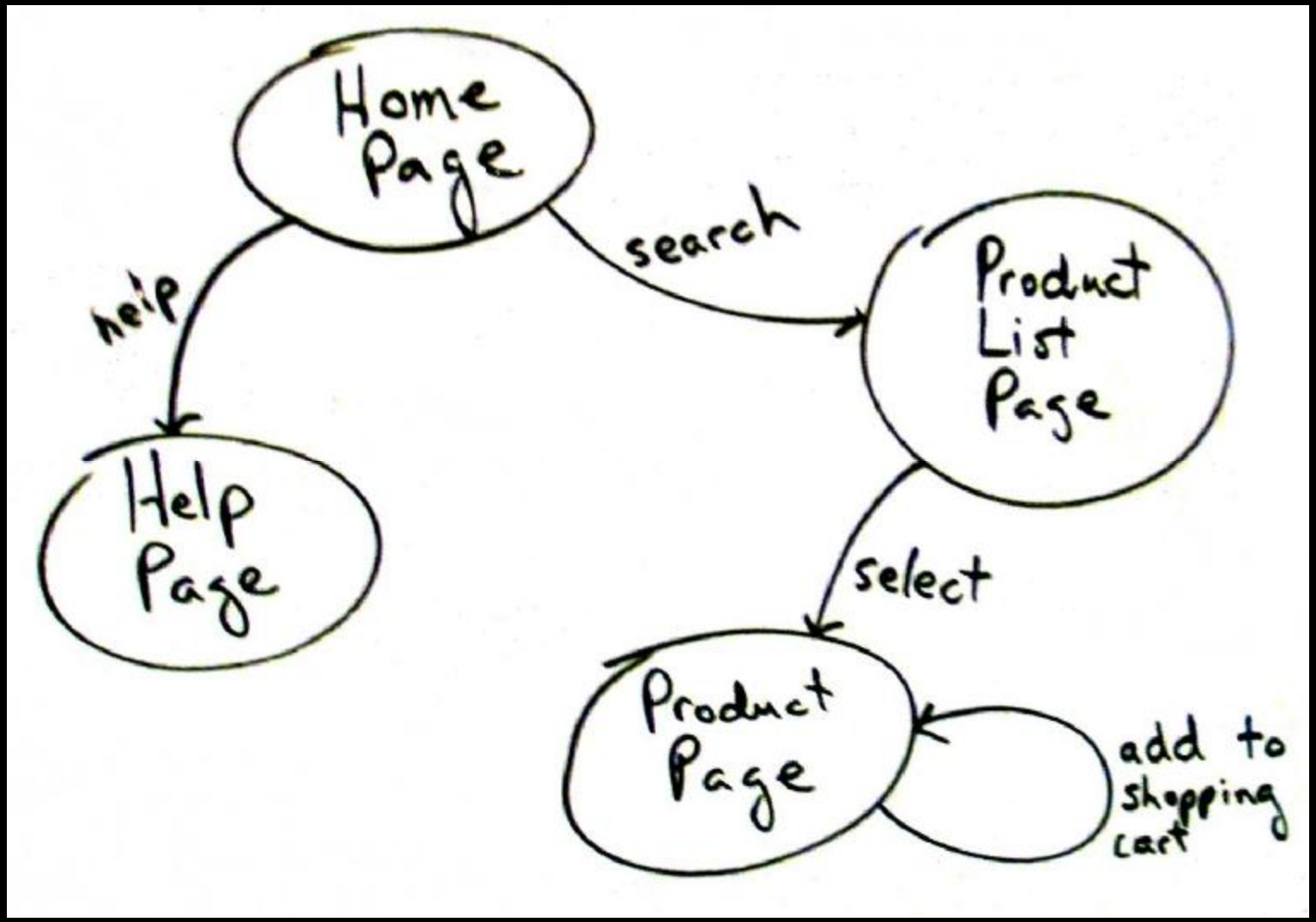


29.4. More UML State Machine Diagram Notation

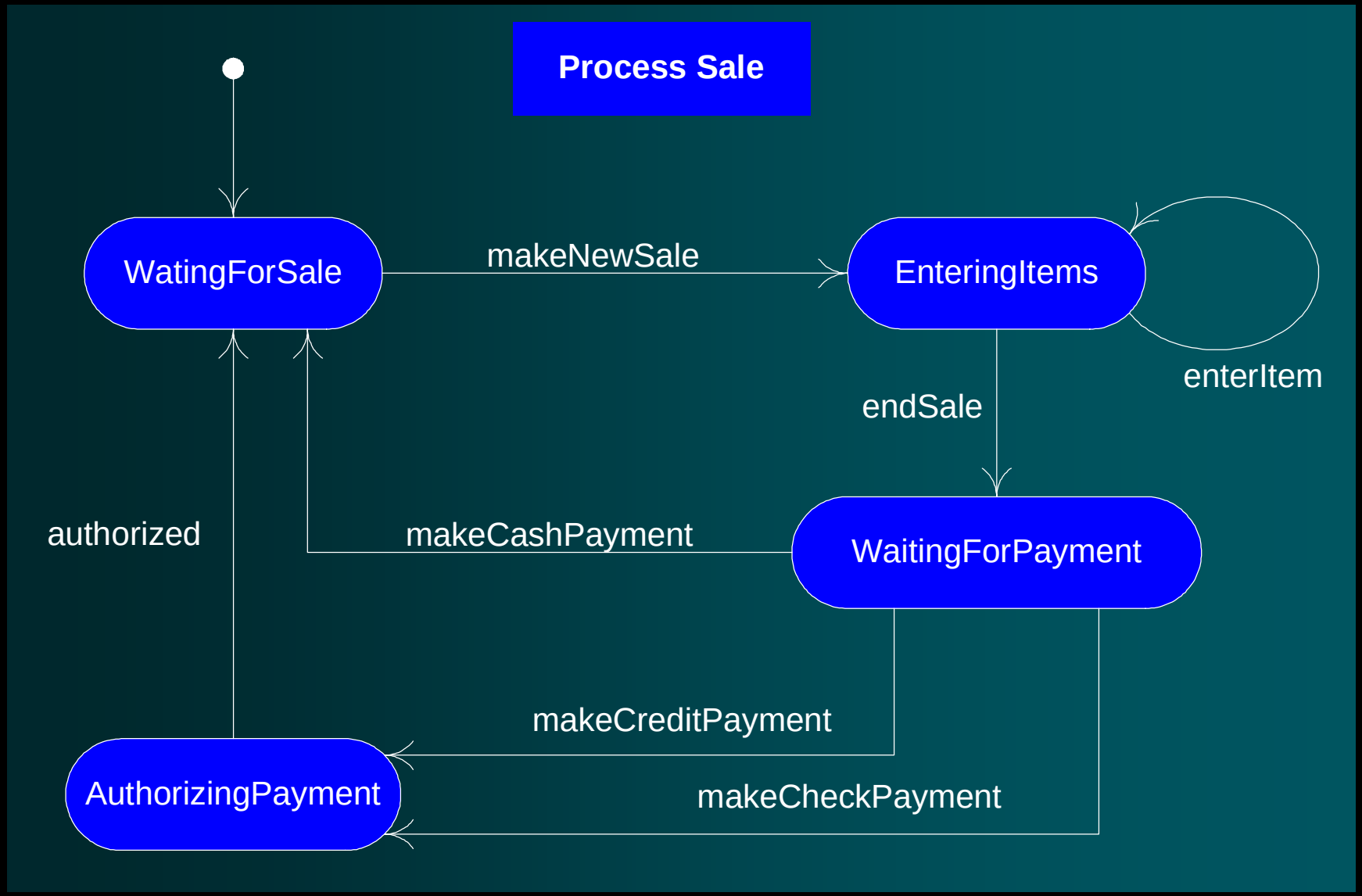
- Nested States



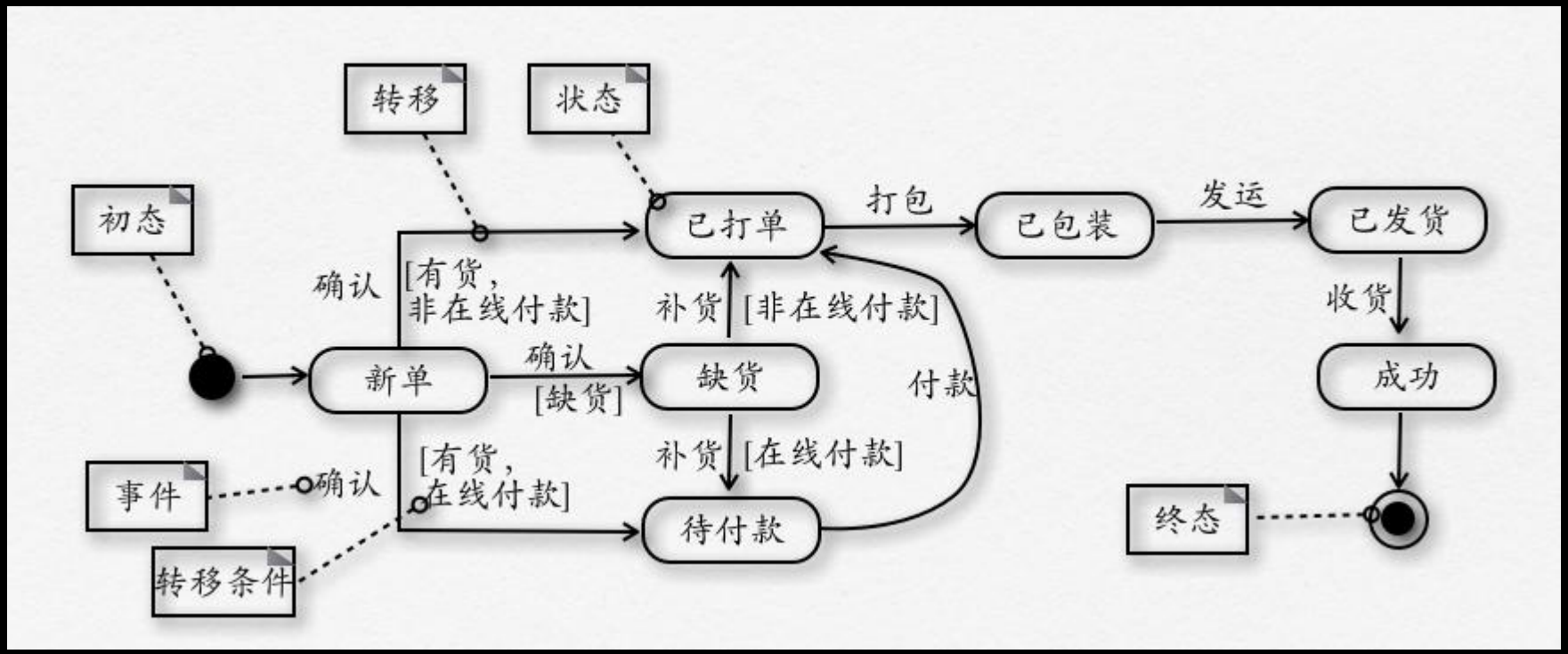
29.5. UI Navigation Modeling with State Machines



29.6. NextGen Use Case State Machine Diagram



29.6. Online Store State Machine Diagram



Chapter 30: Relating Use Cases

Objective

- Relate use cases with include and extend associations, in both text and diagram formats.

30.1. The include Relationship

- Most common and important relationship.
- Subfunction use case
  - Use include when you are repeating yourself in two or more separate use cases and you want to avoid repetition.
    - paying by credit
- Using include with Asynchronous Event Handling

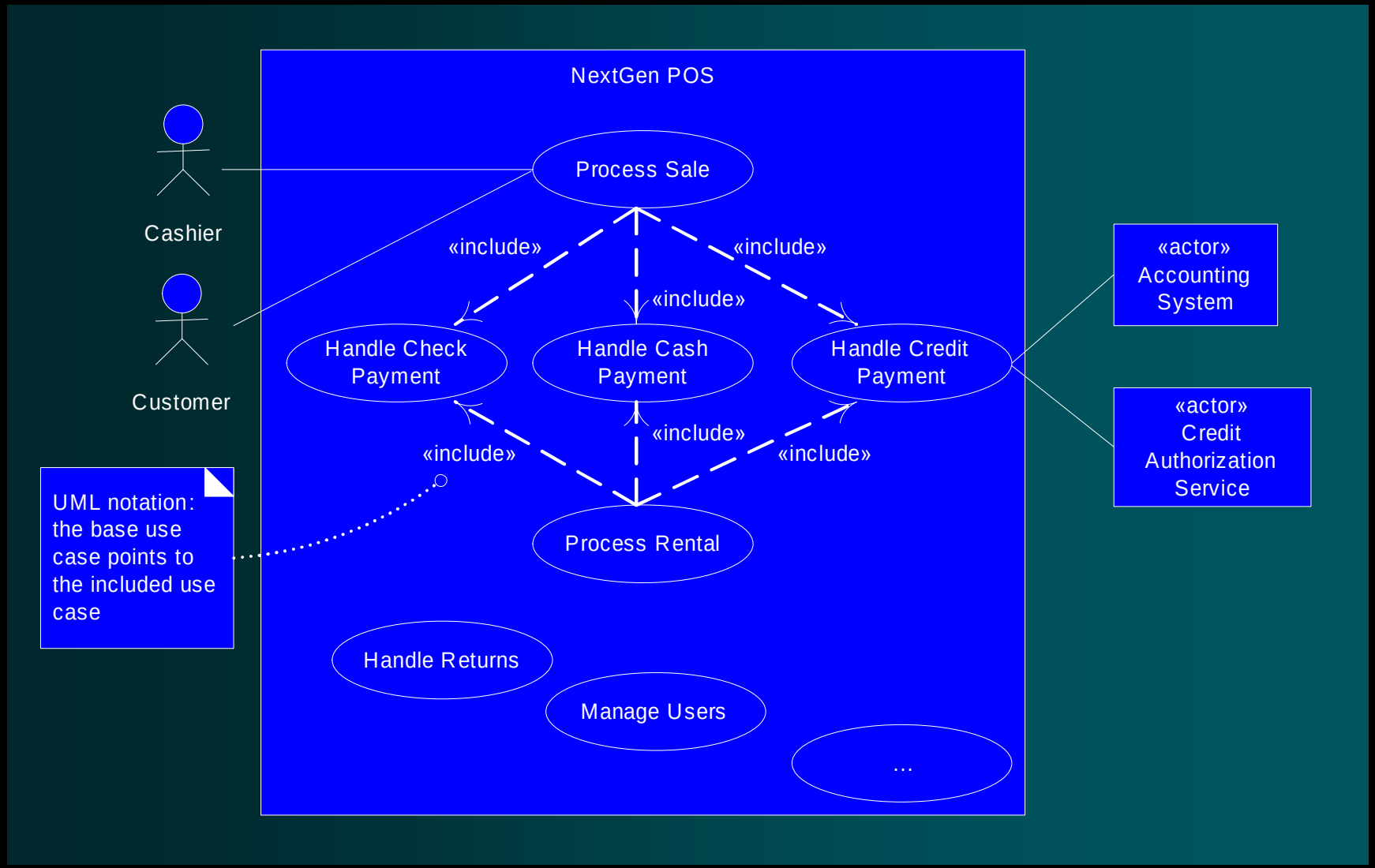
30.2. Concrete, Abstract, Base, and Addition Use Cases

- Concrete Use Case
  - initiated by an actor and performs the entire behavior desired by the actor.
- Abstract Use Case
  - a subfunction use case that is part of another use case.
- Base Use Case
  - A use case that includes another use case, or that is extended or specialized by another use case.
- Addition Use Case
  - The use case that is an inclusion, extension, or specialization.

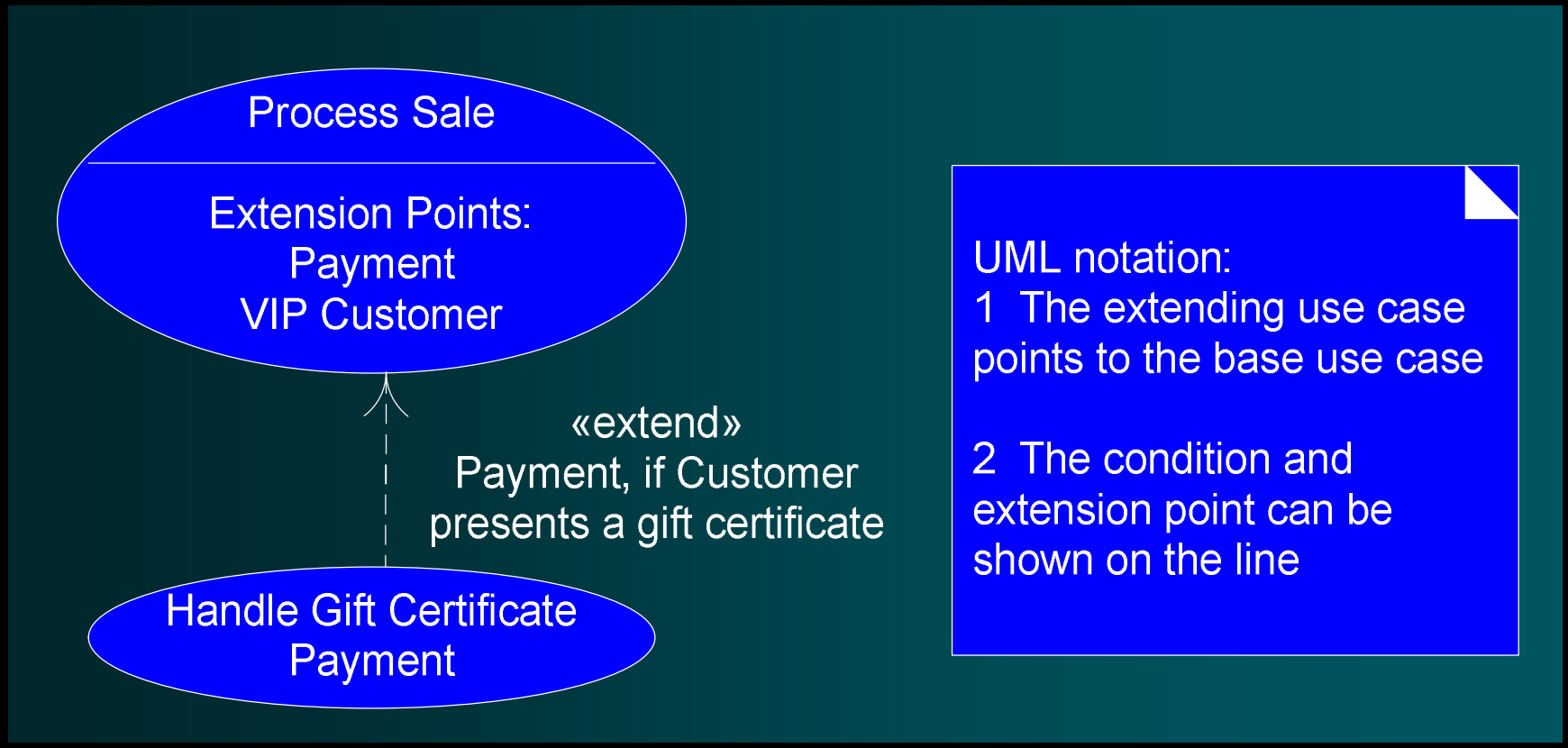
30.3. The extend Relationship

- Create an extending or addition use case
  - to describe where and under what condition it extends the behavior of some base use case.
  - An option when the base use case is closed to modification.

30.5. Use Case Diagrams



30.5. Use Case Diagrams



UML notation:
1 The extending use case points to the base use case

2 The condition and extension point can be shown on the line

Chapter 31: Domain Model Refinement

Objective

- Refine the domain model with
  - generalizations
  - specializations
  - association classes
  - time intervals
  - composition
  - packages
- Identify when showing a subclass is worthwhile

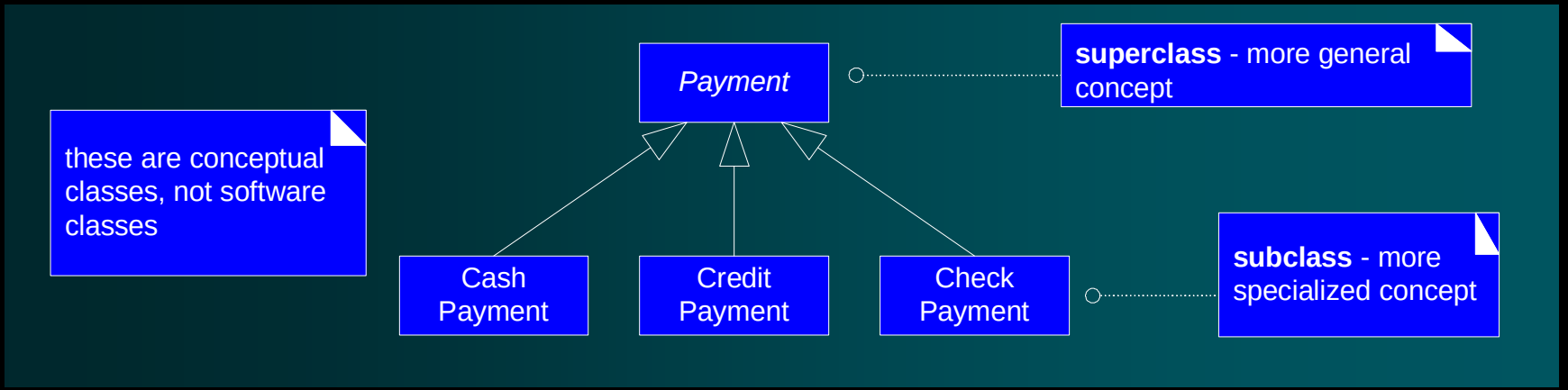
31.1. New Concepts for the NextGen Domain Model

- Concepts Category List
- Noun Phrase Identification from the Use Cases
- Authorization Service Transactions

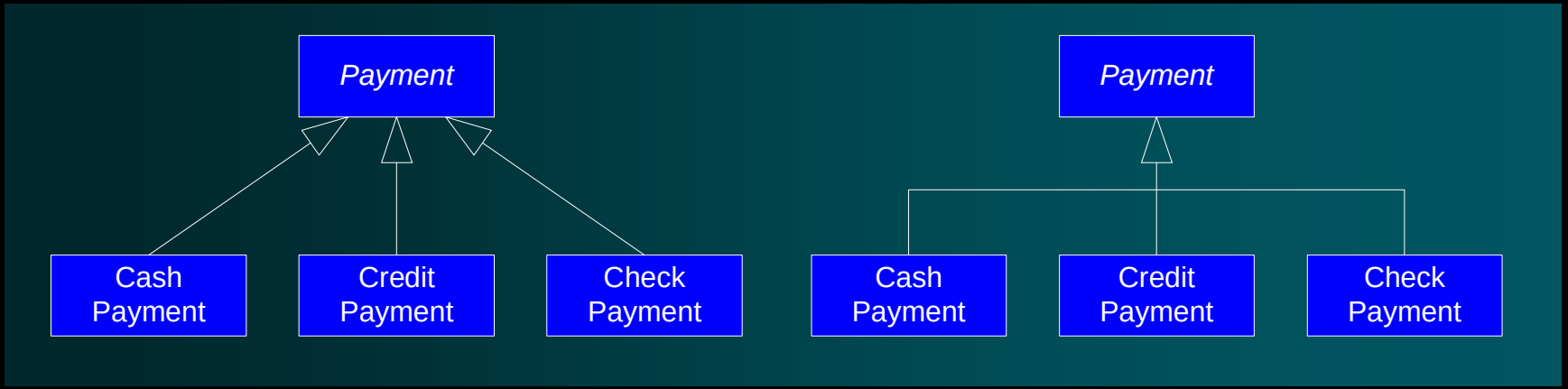
31.2. Generalization

w Generalization

- § identify commonality among concepts
- § define superclass (general concept) and subclass (specialized concept) relationships.
- § economy of expression, improved comprehension and a reduction in repeated information



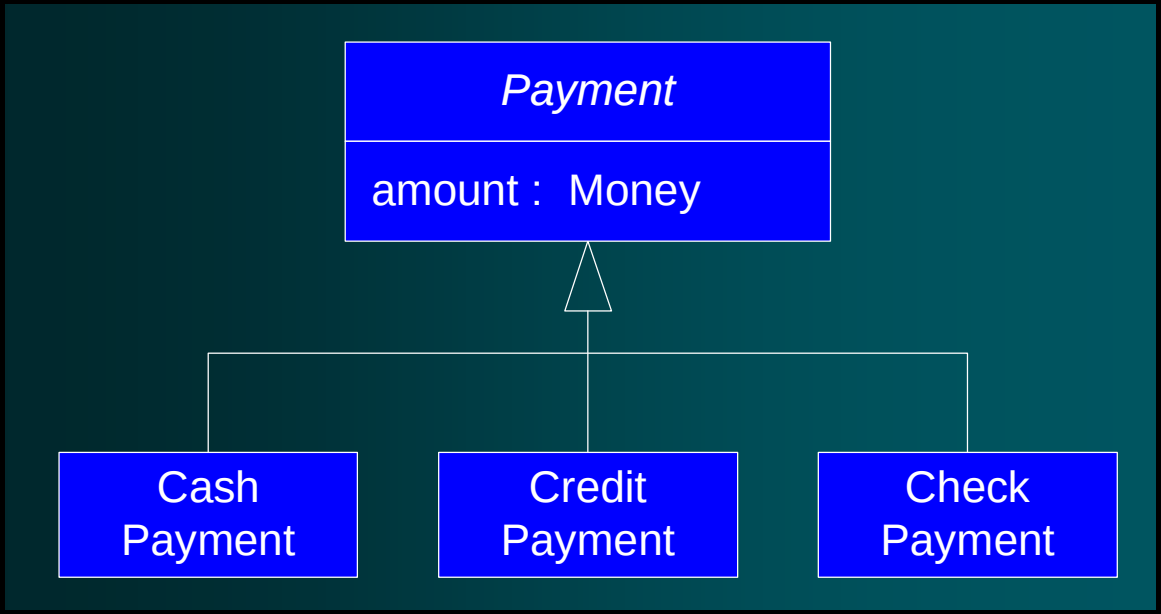
31.2. Generalization



31.3. Defining Conceptual Superclasses and Subclasses

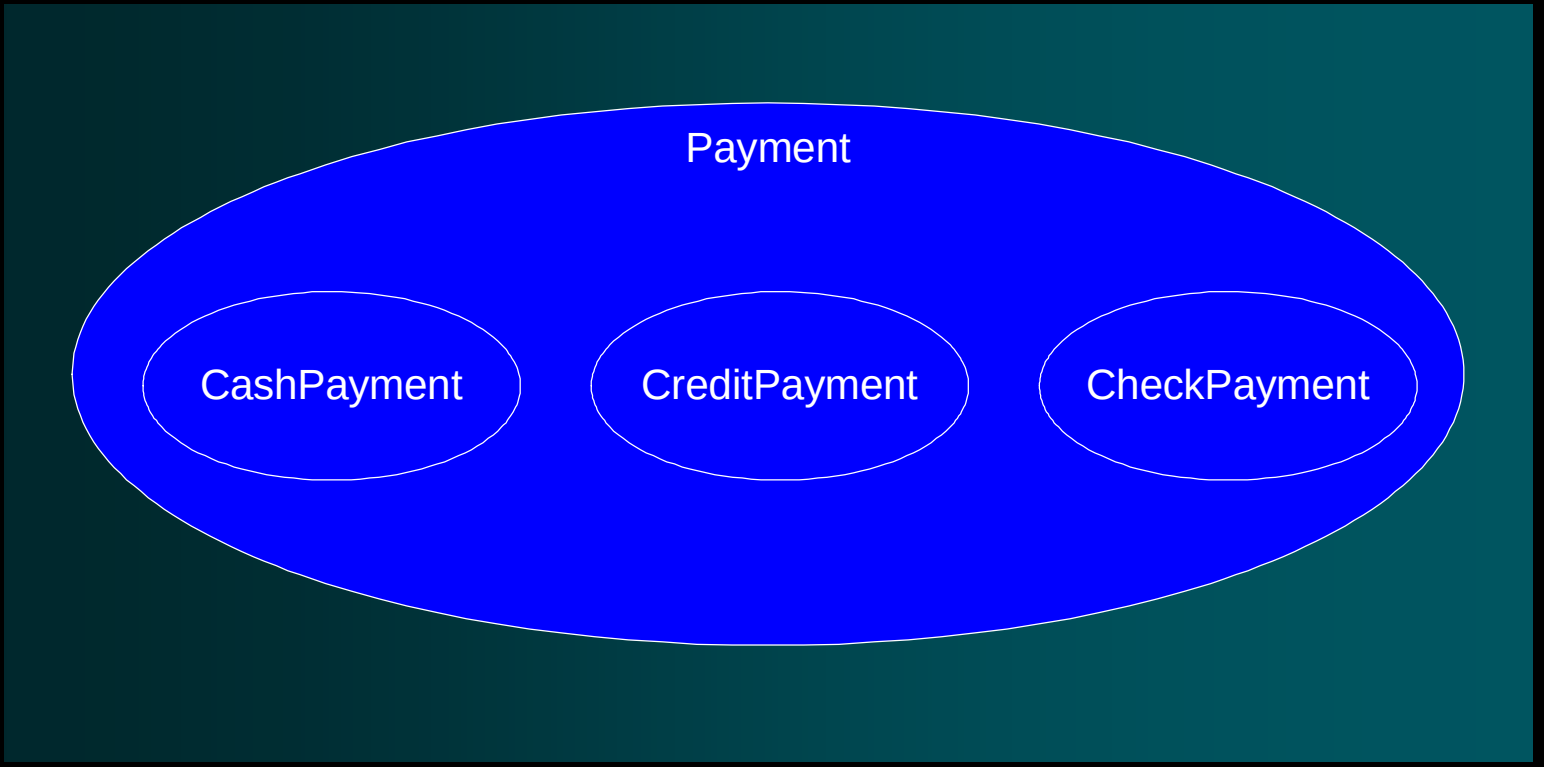
w Generalization and Conceptual Class Definition

- § A conceptual superclass definition is more general or encompassing than a subclass definition.



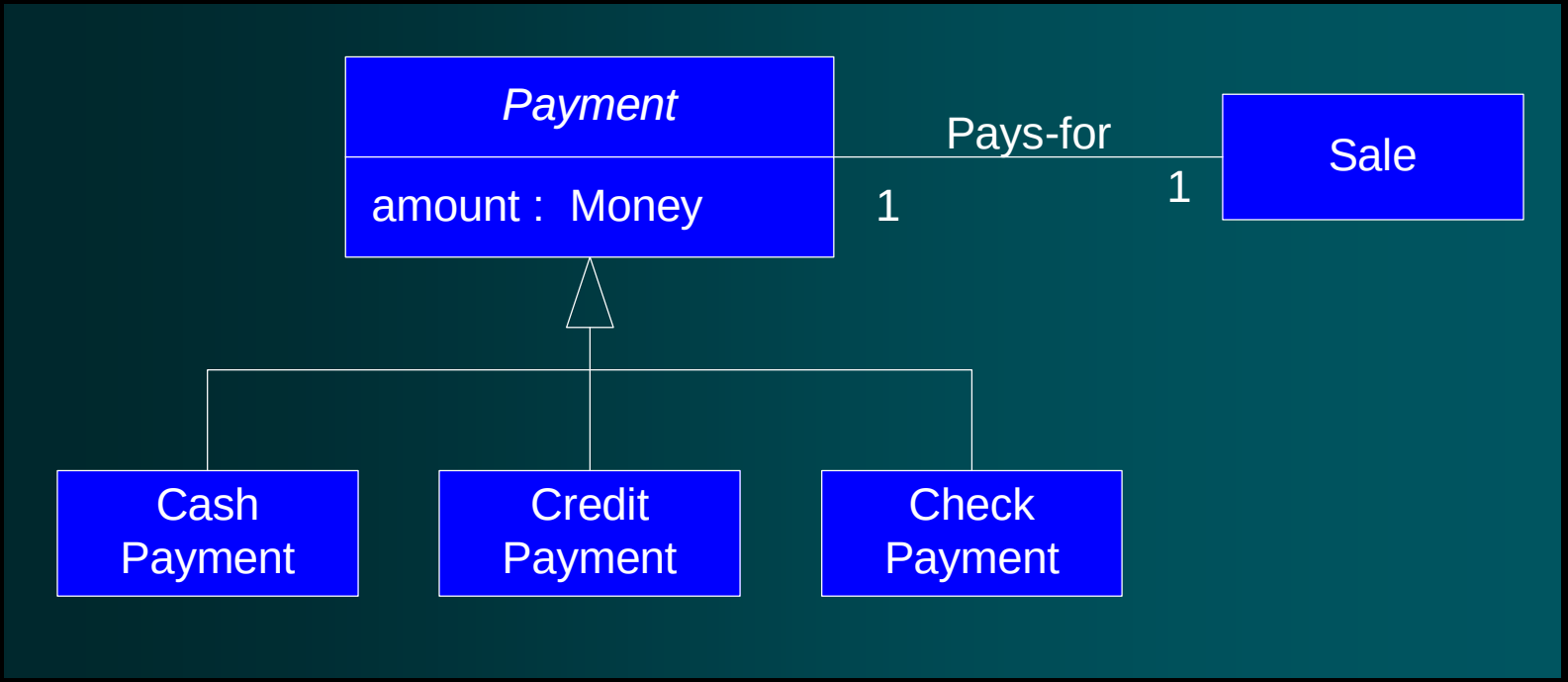
31.3. Defining Conceptual Superclasses and Subclasses

- Generalization and Class Sets
  - All members of a conceptual subclass set are members of their superclass set.



31.3. Defining Conceptual Superclasses and Subclasses

- Conceptual Subclass Definition Conformance
  - statements about superclasses are also true to subclasses

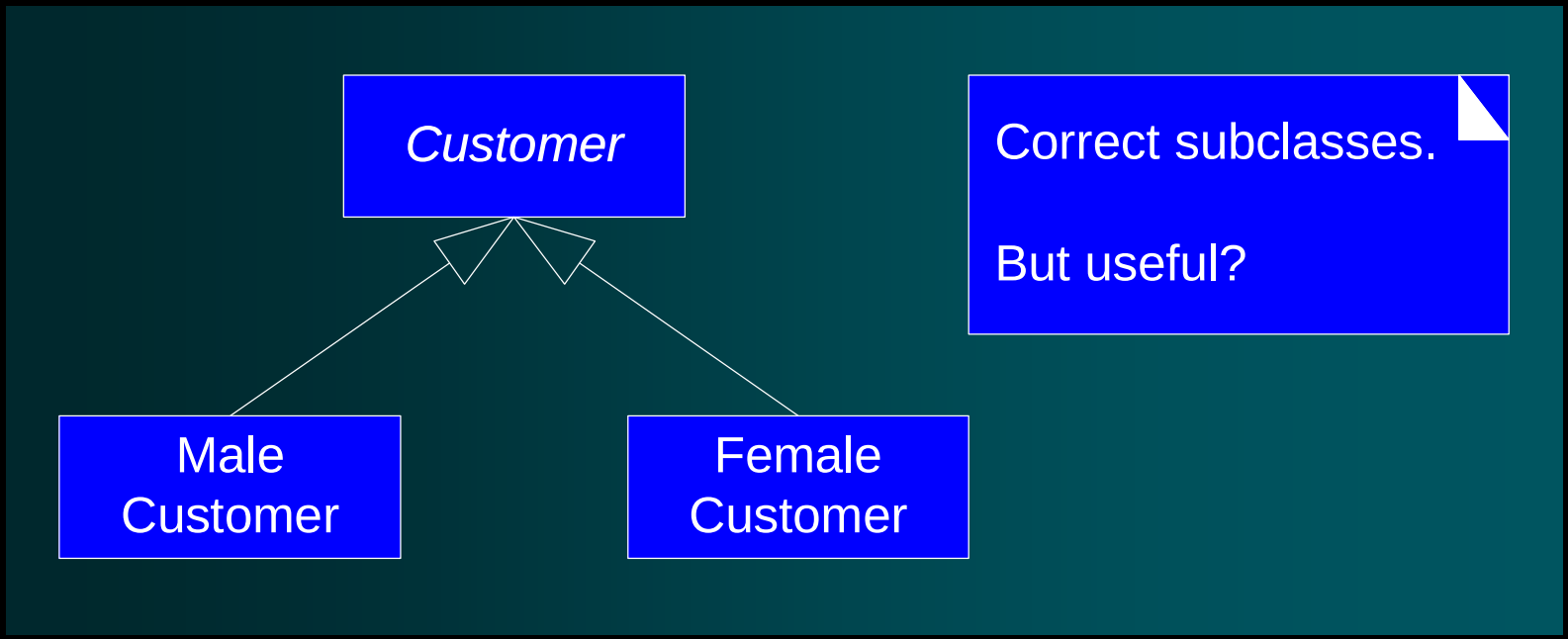


31.3. Defining Conceptual Superclasses and Subclasses

- Conceptual Subclass Set Conformance
  - A conceptual subclass should be a member of the set of the superclass.
- **What Is a Correct Conceptual Subclass?**
  - A potential subclass should conform to the:
    - 100% Rule (definition conformance)
    - Is-a Rule (set membership conformance)

31.4. When to Define a Conceptual Subclass?

- A conceptual class partition is a division of a conceptual class into disjoint subclasses



31.4. When to Define a Conceptual Subclass?

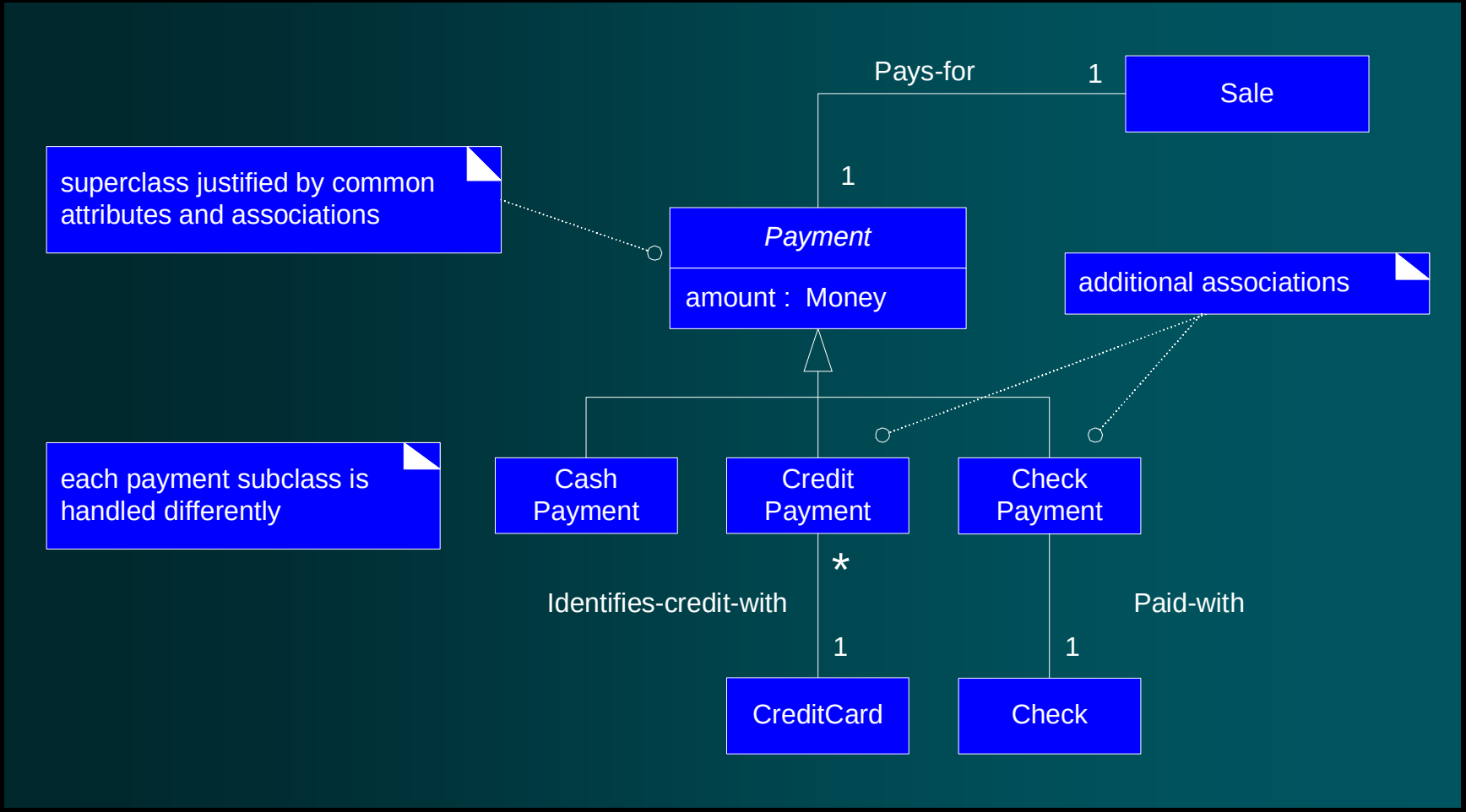
- Create a conceptual subclass of a superclass when:
  - The subclass has additional attributes of interest.
  - The subclass has additional associations of interest.
  - The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses.
  - The subclass concept represents an animate thing that behaves differently than the superclass or other subclasses.

31.5. When to Define a Conceptual Superclass?

- Create a superclass in a generalization relationship to subclasses when:
  - 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.

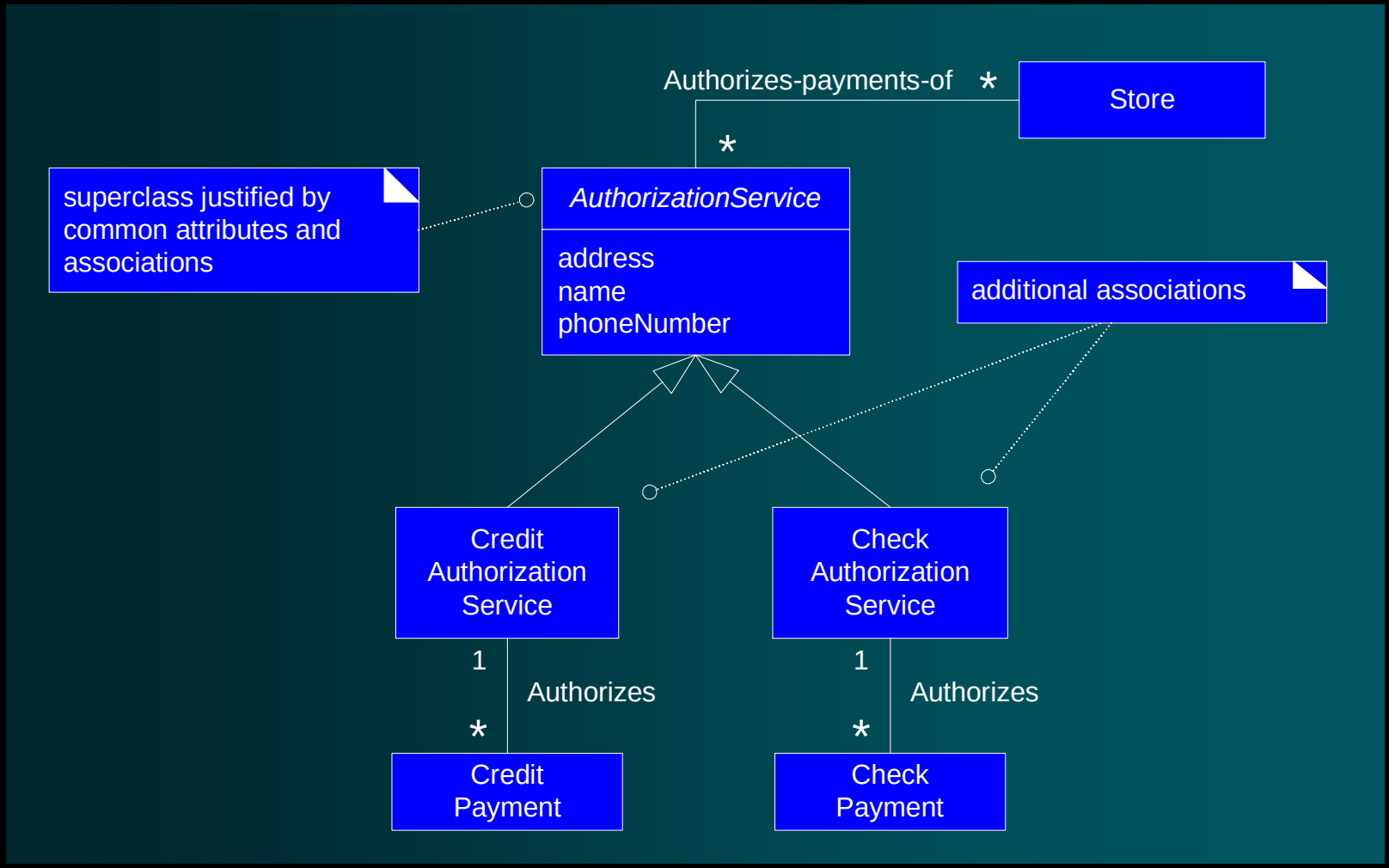
31.6. NextGen POS Conceptual Class Hierarchies

w Payment Classes



31.6. NextGen POS Conceptual Class Hierarchies

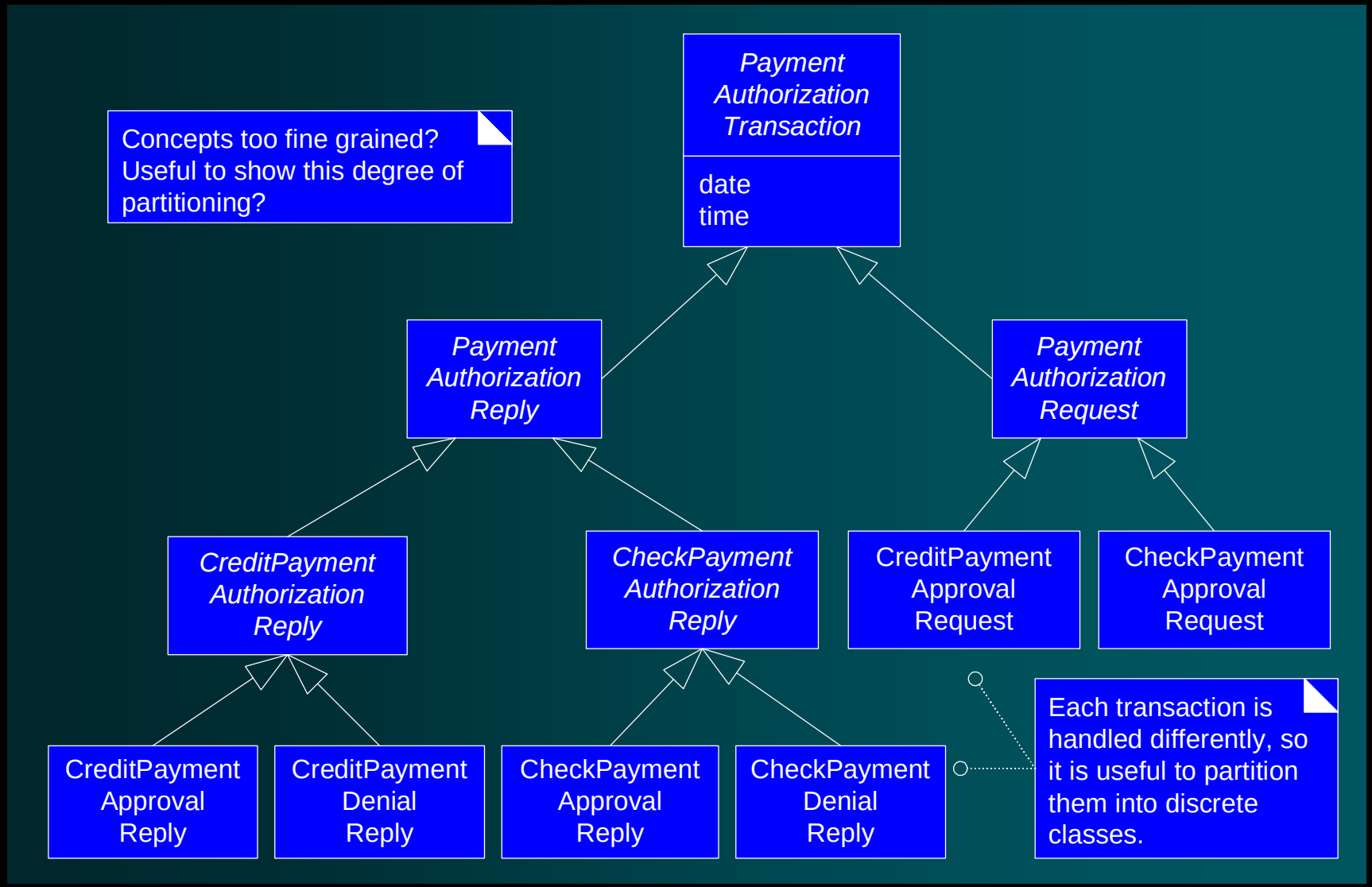
w Authorization Service Classes



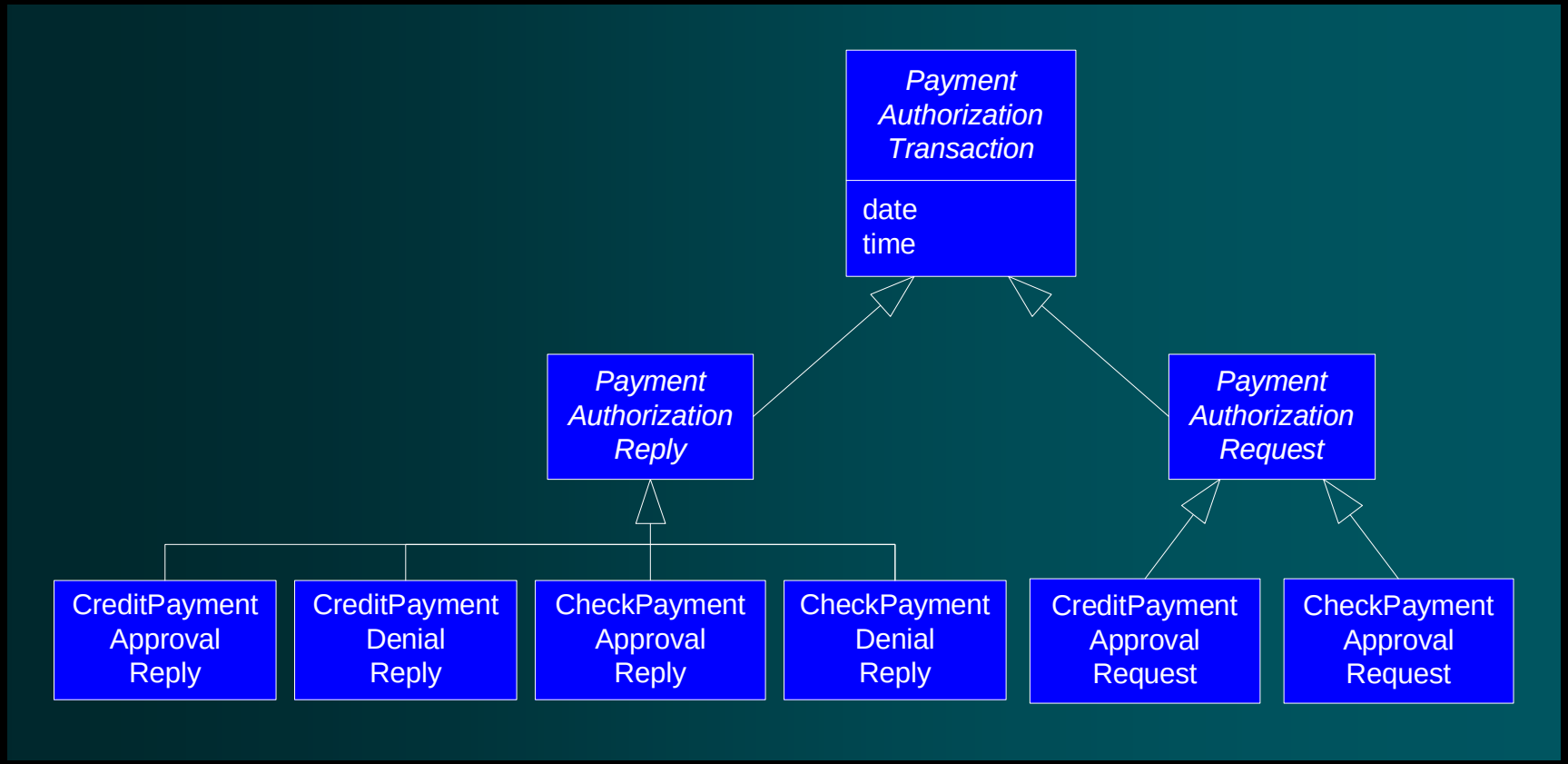
31.6. NextGen POS Conceptual Class Hierarchies

- Authorization Transaction Classes
  - Should the modeler illustrate every variation of an external service transaction?
  - the degree of generalization that is useful to show in the model

31.6. NextGen POS Conceptual Class Hierarchies

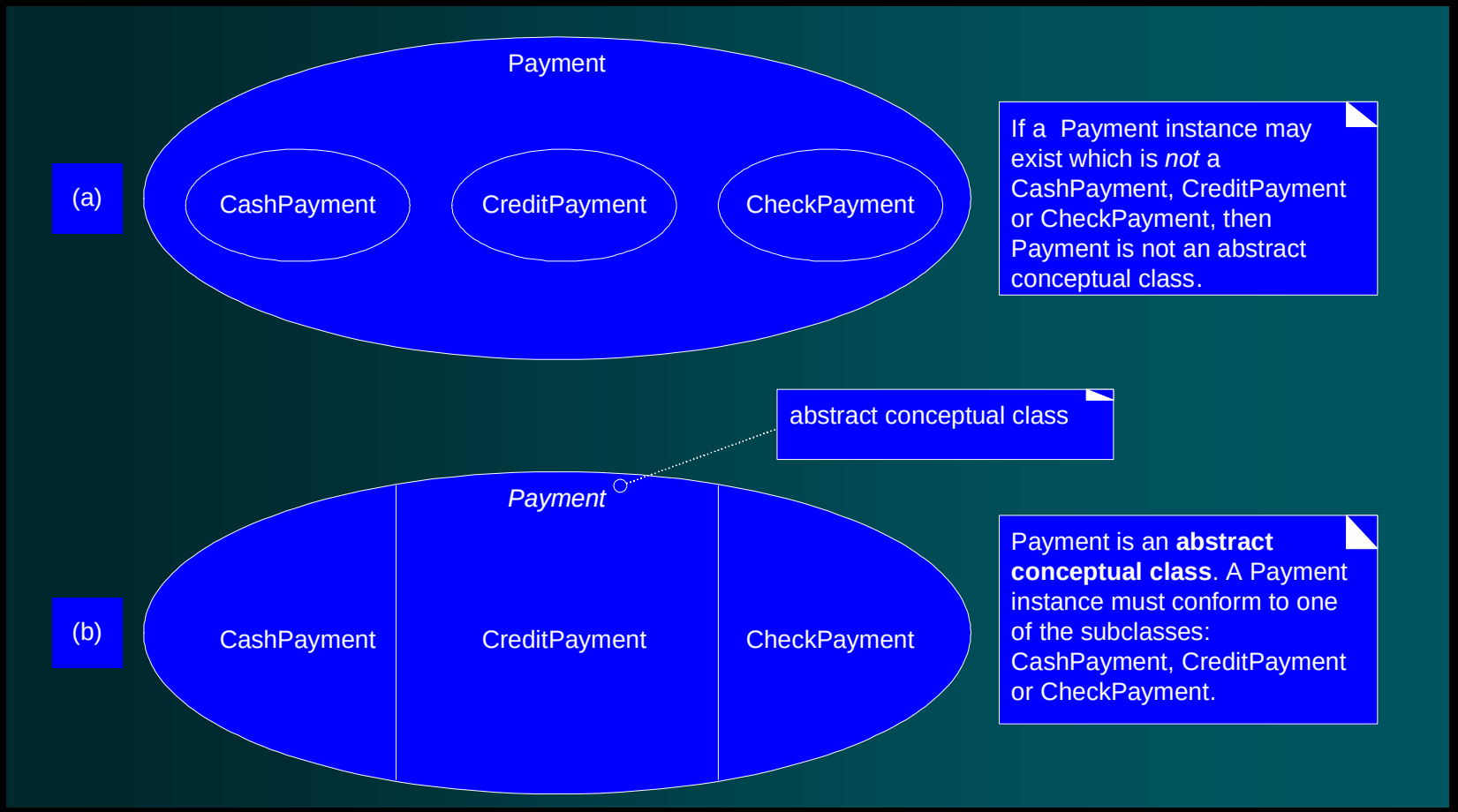


31.6. NextGen POS Conceptual Class Hierarchies



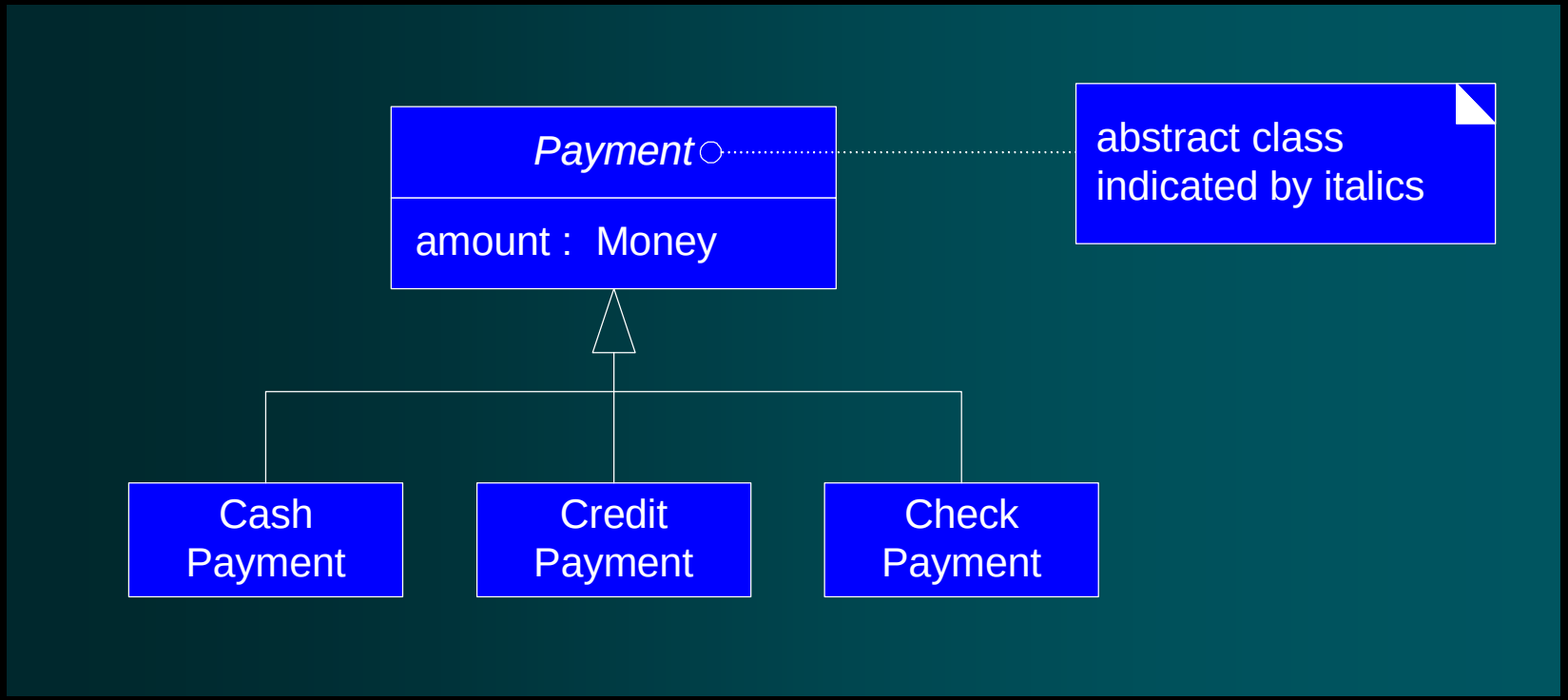
31.7. Abstract Conceptual Classes

- Every member of a class C must also be a member of a subclass



31.7. Abstract Conceptual Classes

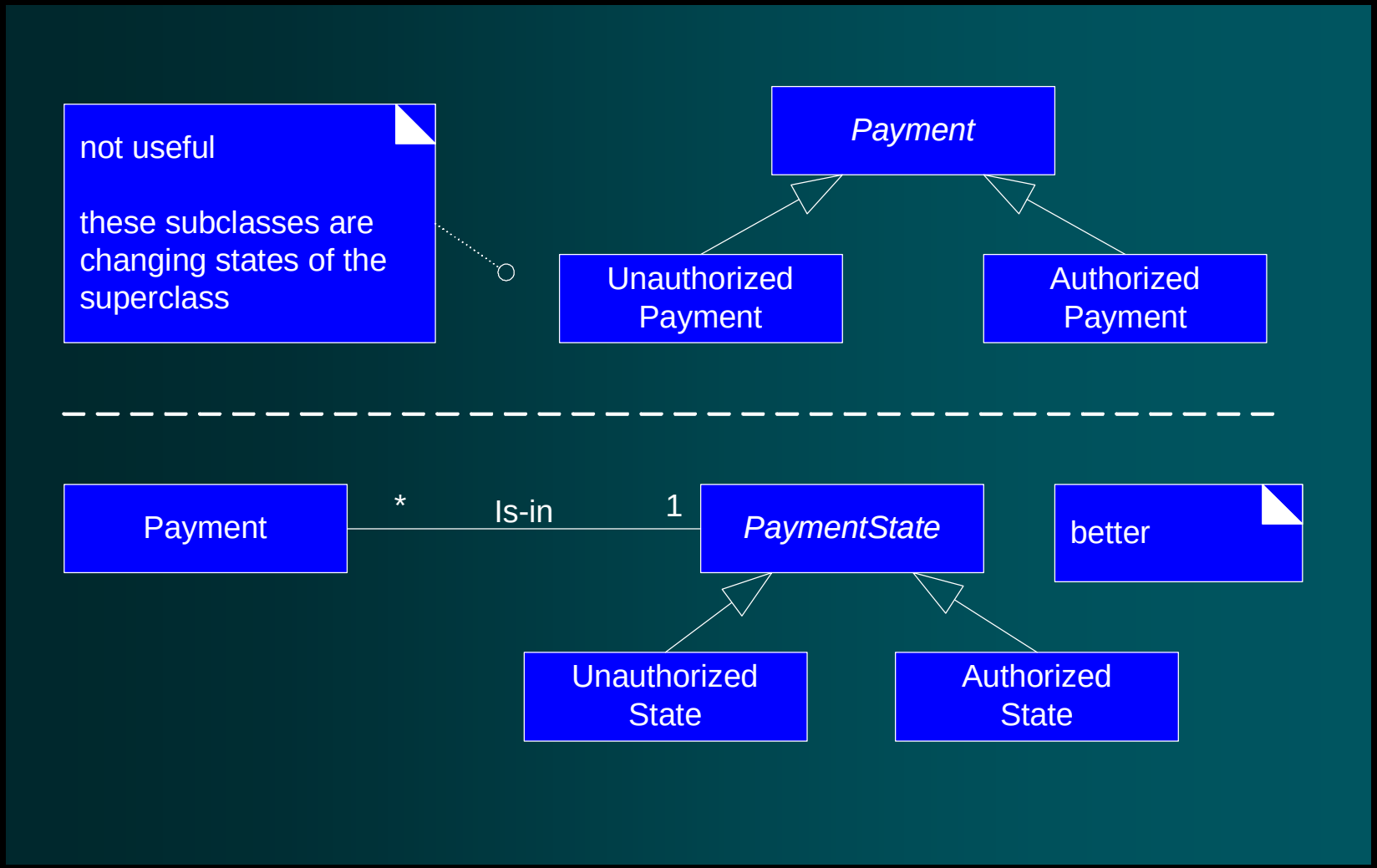
w Abstract Class Notation in the UML



31.8. Modeling Changing States

- Define a state hierarchy and associate the states with the class,
- Ignore showing the states of a concept in the domain model;
  - show the states in state diagrams instead.

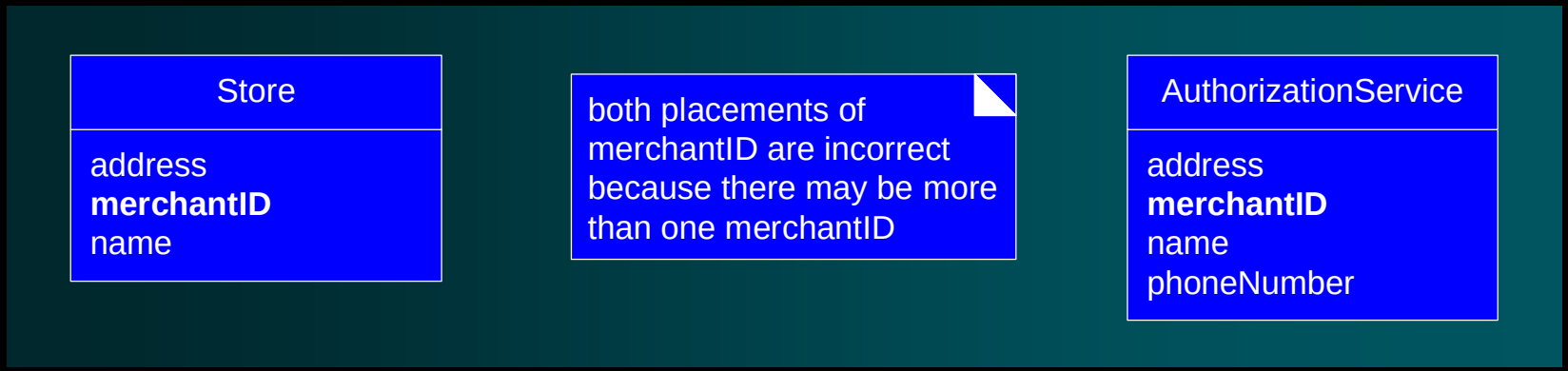
31.8. Modeling Changing States



31.10. Association Classes

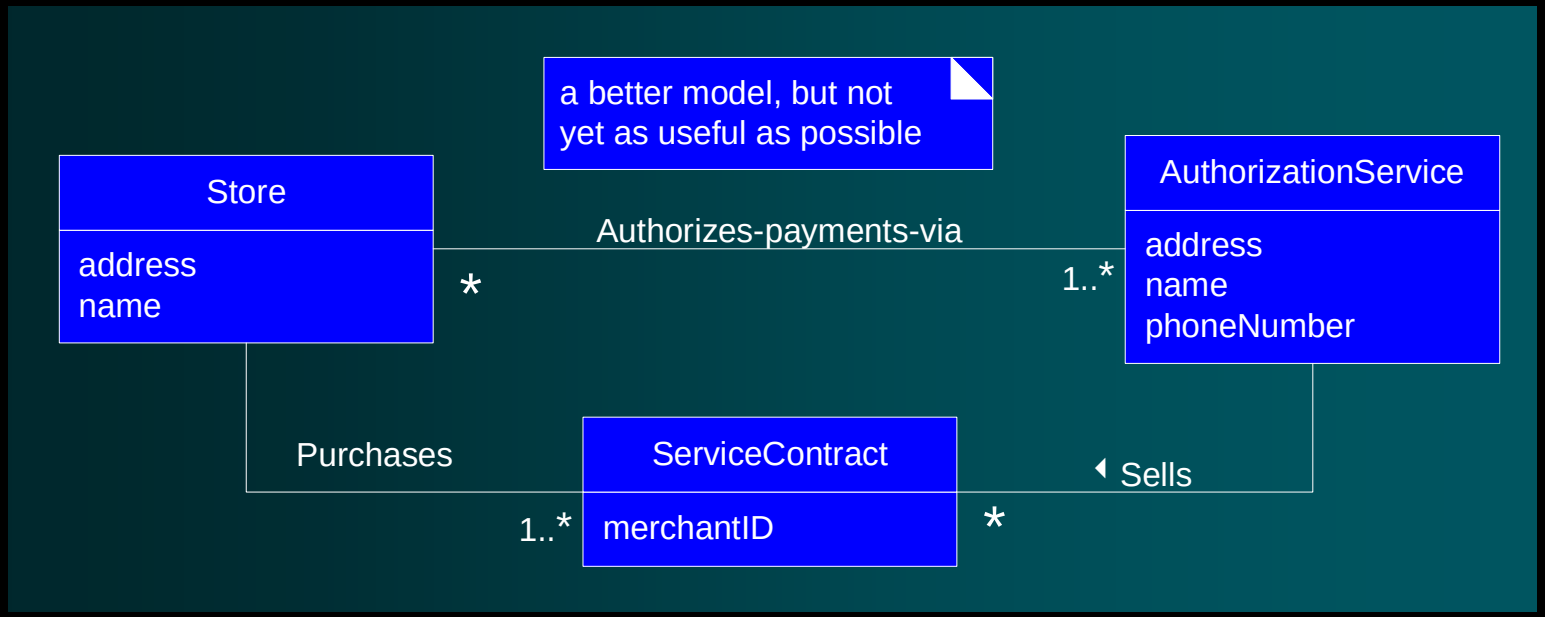
- Some domain requirements:
  - Authorization services assign a merchant ID to each store for identification during communications.
  - A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.
  - Furthermore, a store has a different merchant ID for each service.
- Where in the UP Domain Model should the merchant ID attribute reside?

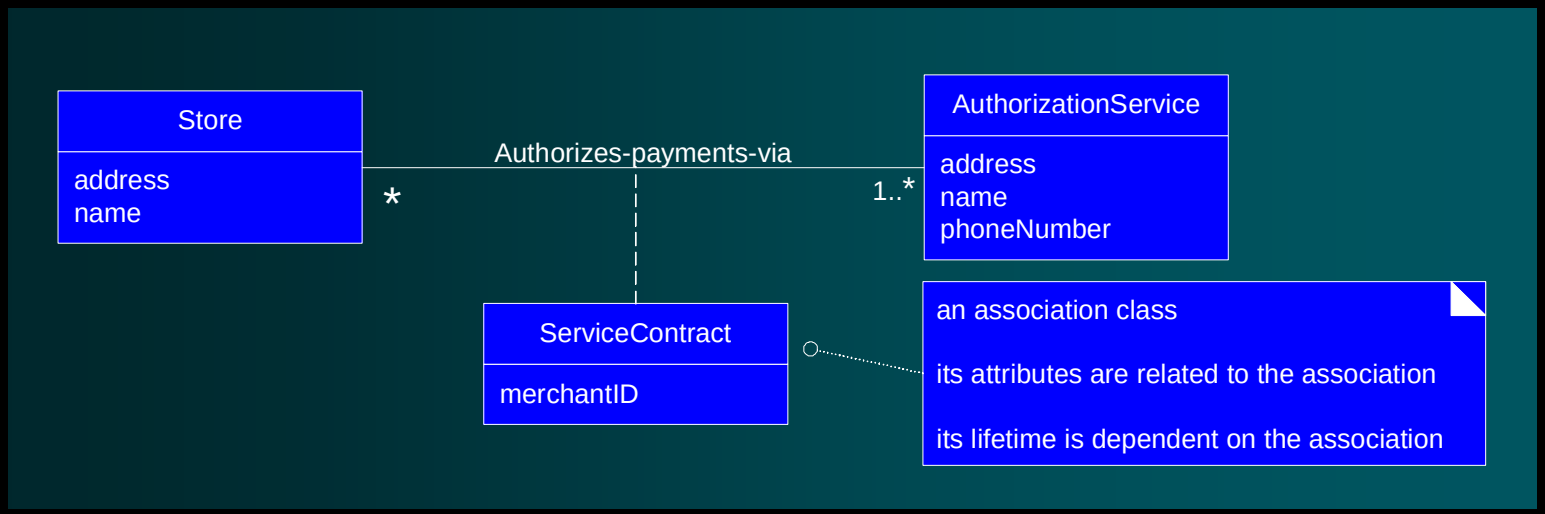
31.10. Association Classes



- When a class C can simultaneously have many values for the same kind of attribute A,
  - do not place attribute A in C.
  - Place attribute A in another class that is associated with C.

31.10. Association Classes

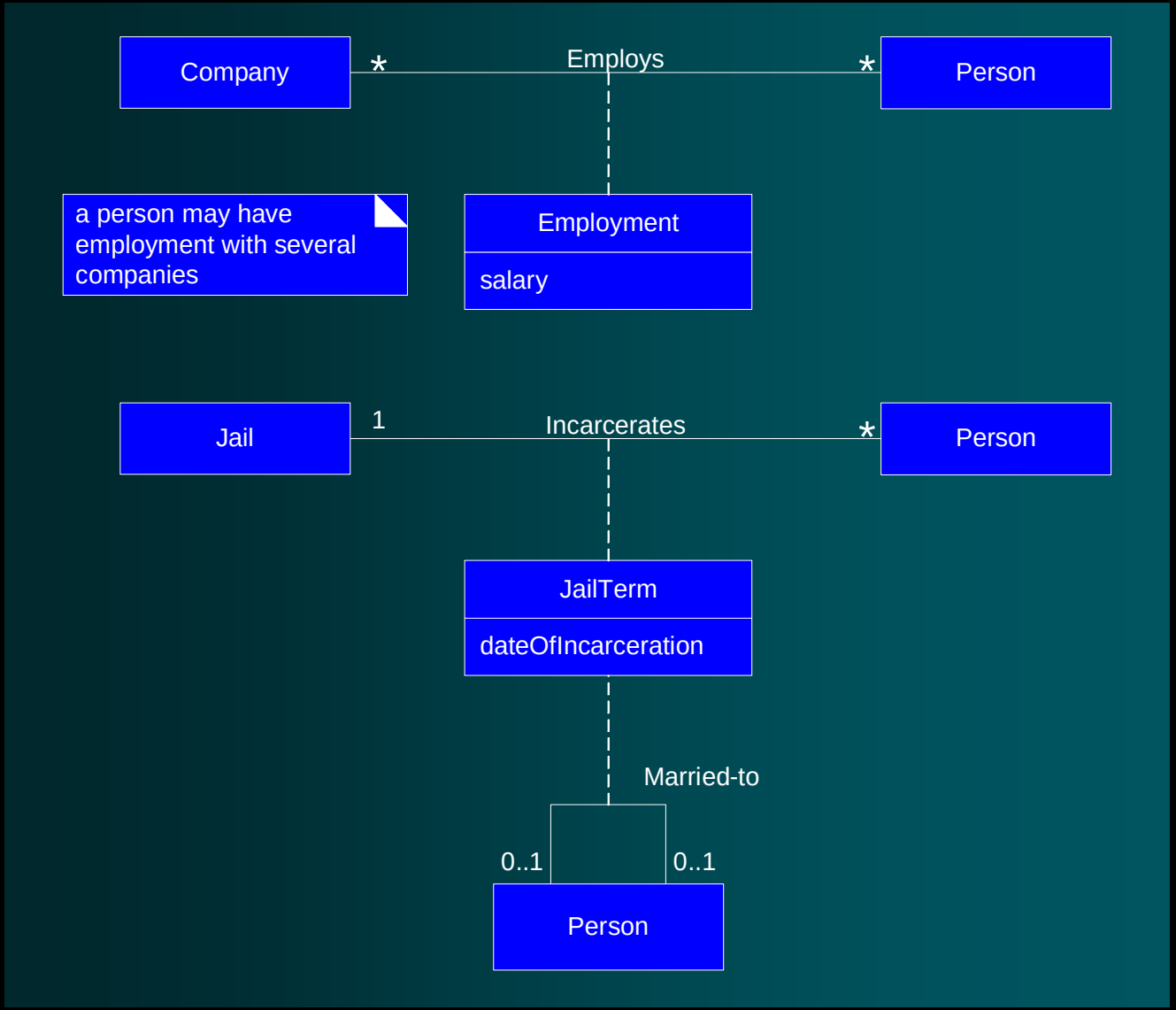




31.10. Association Classes

- Clues that an association class might be useful in a domain model:
  - An attribute is related to an association.
  - Instances of the association class have a lifetime dependency on the association.
  - There is a many-to-many association between two concepts and information associated with the association itself.

31.10. Association Classes



31.11. Aggregation and Composition

- Aggregation
  - Loosely suggests whole-part relationships
  - A plain association in UML
- Composition
  - an instance of the part at a time
  - the part must always belong to a composite
  - the composite is responsible for the creation and deletion of its parts.

31.11. Aggregation and Composition

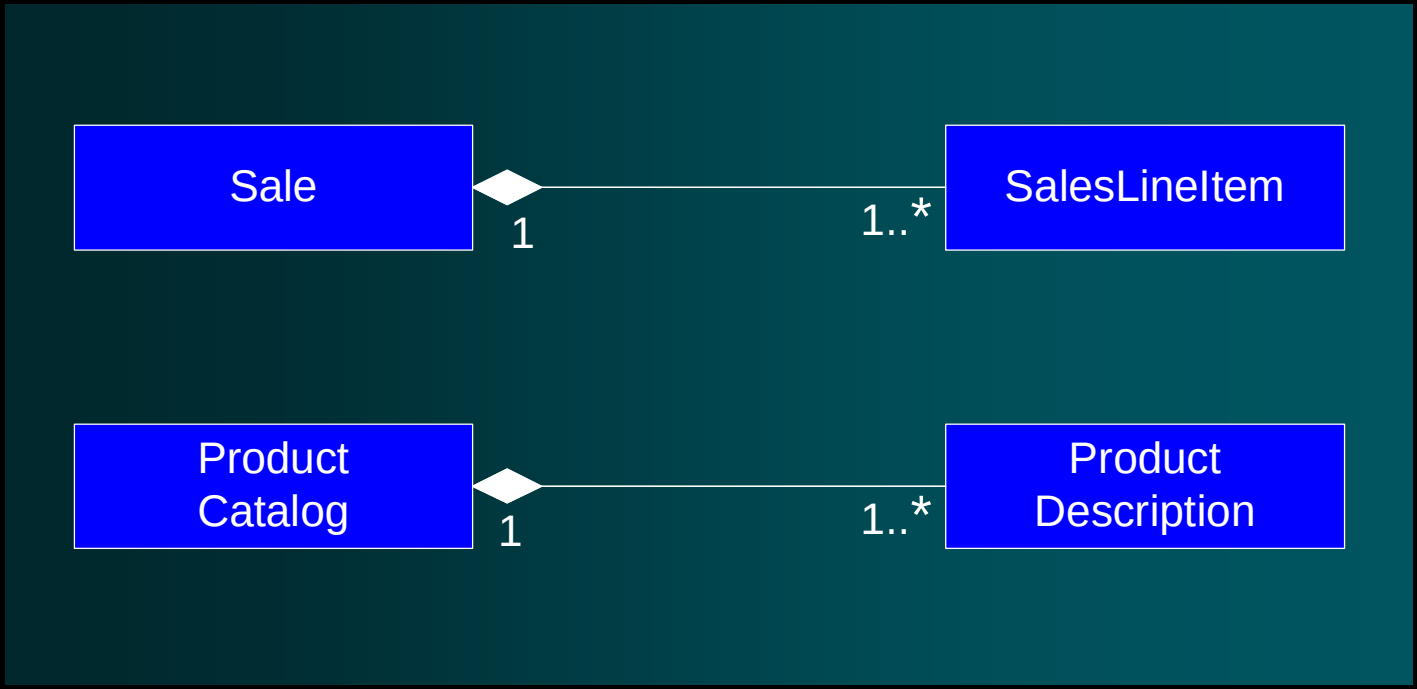
- How to Identify Composition
  - Consider showing composition when:
    - The lifetime of the part is bound within the lifetime of the composite.
    - There is an obvious whole-part physical or logical assembly.
    - Some properties of the composite propagate to the parts, such as the location.
    - Operations applied to the composite propagate to the parts, such as destruction, movement, recording.

31.11. Aggregation and Composition

- A Benefit of Showing Composition
  - It clarifies the domain constraints regarding the eligible existence of the part independent of the whole.
  - It assists in the identification of a creator using the GRASP Creator pattern.
  - Operations applied to the whole often propagate to the parts.

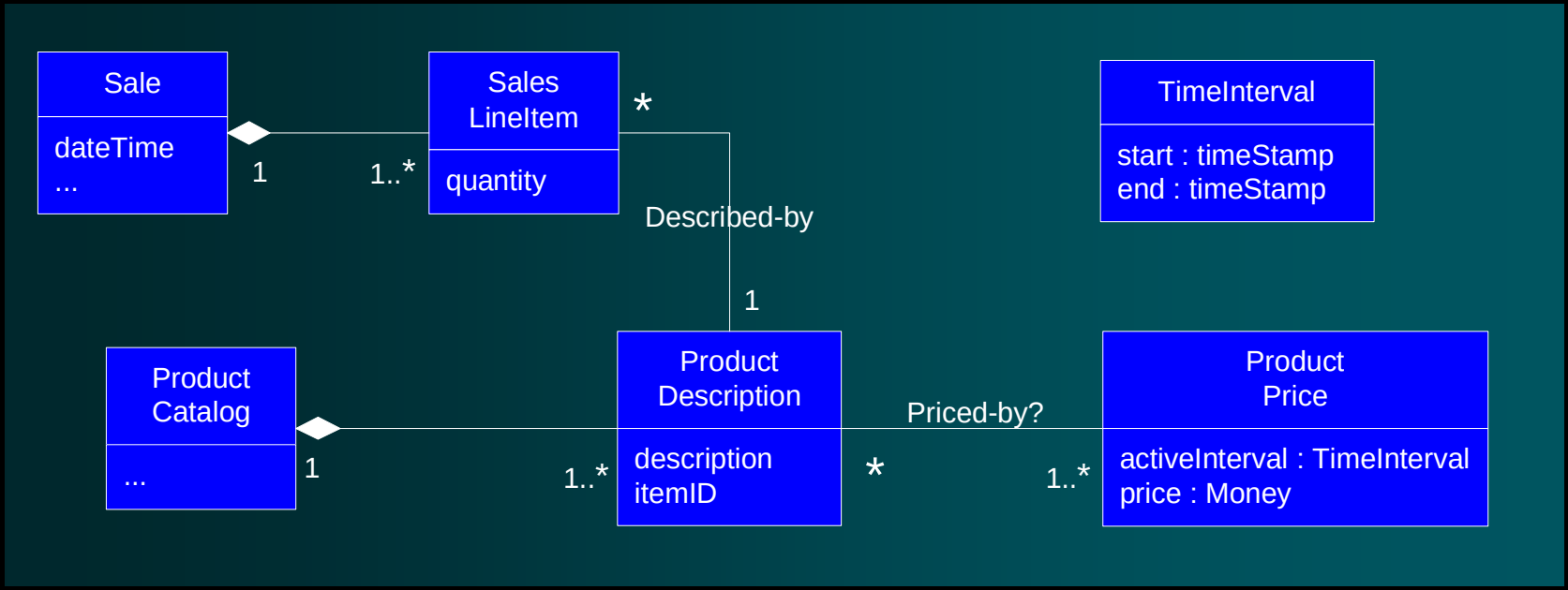
31.11. Aggregation and Composition

- Composition in the NextGen Domain Model

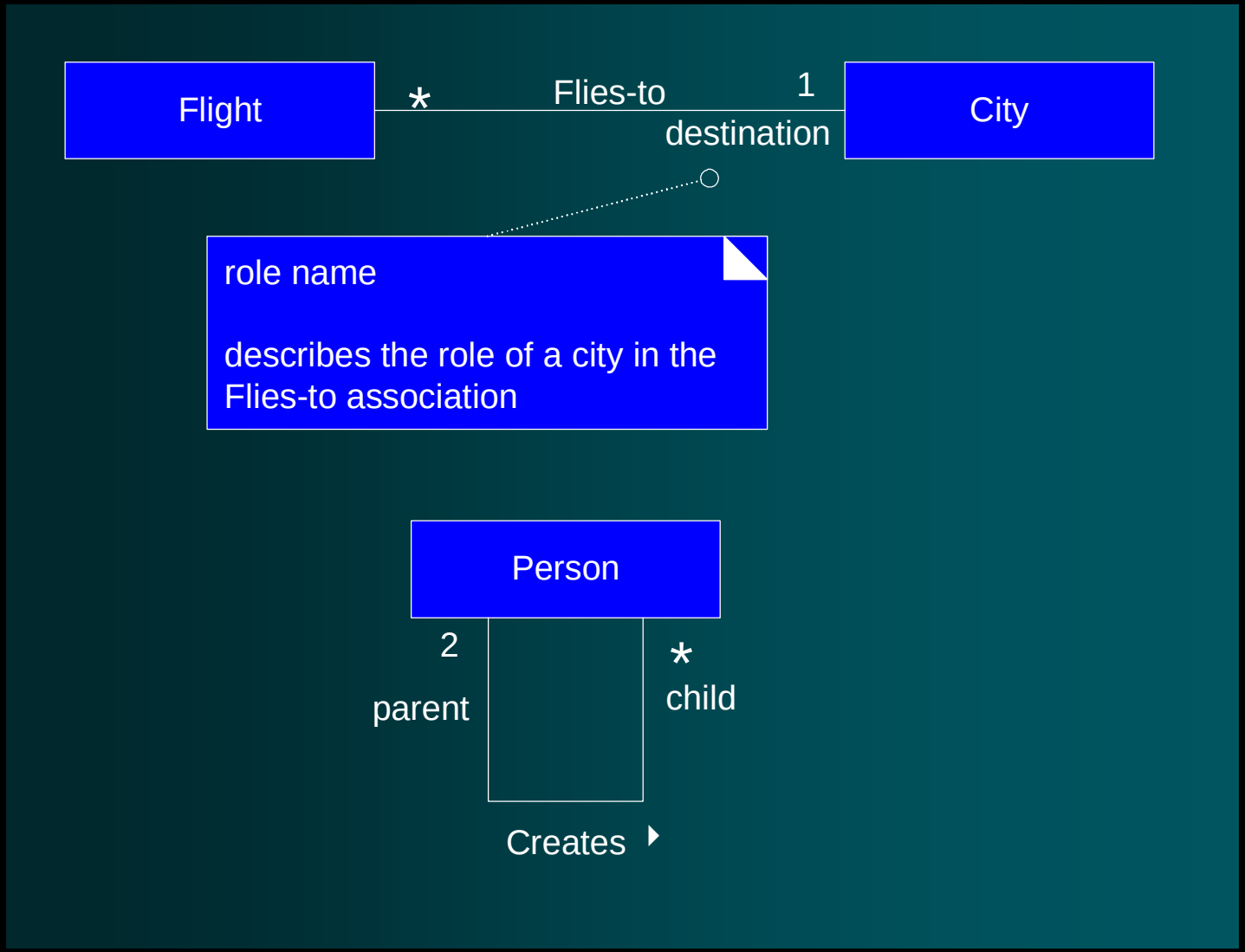


31.12. Time Intervals and Product

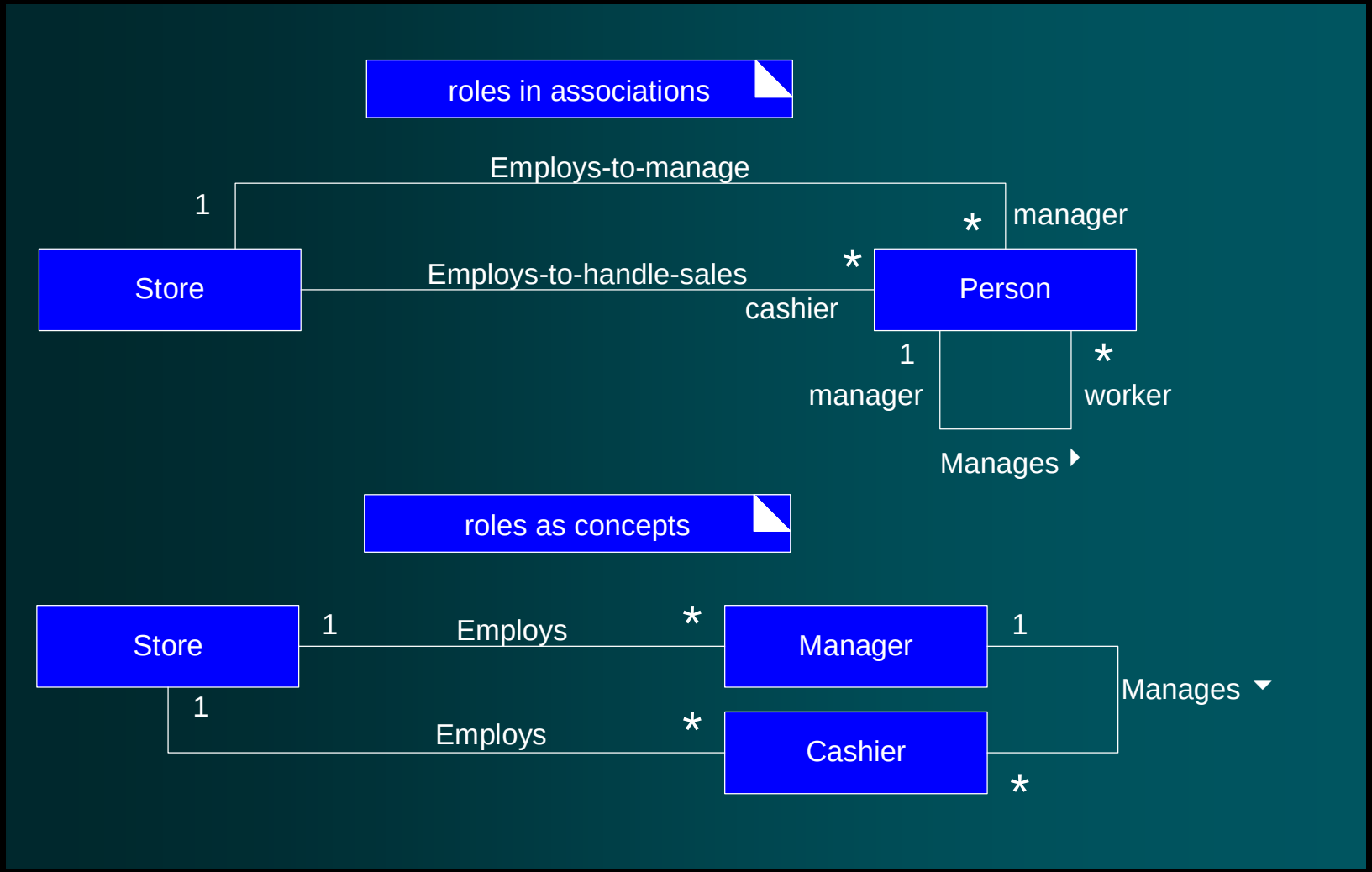
- In the first iteration,
  - SalesLineItems were associated with ProductDescriptions, that recorded the price of an item.
- What shall we do when price is changed?



31.13. Association Role Names



31.14. Roles as Concepts versus Roles in Associations

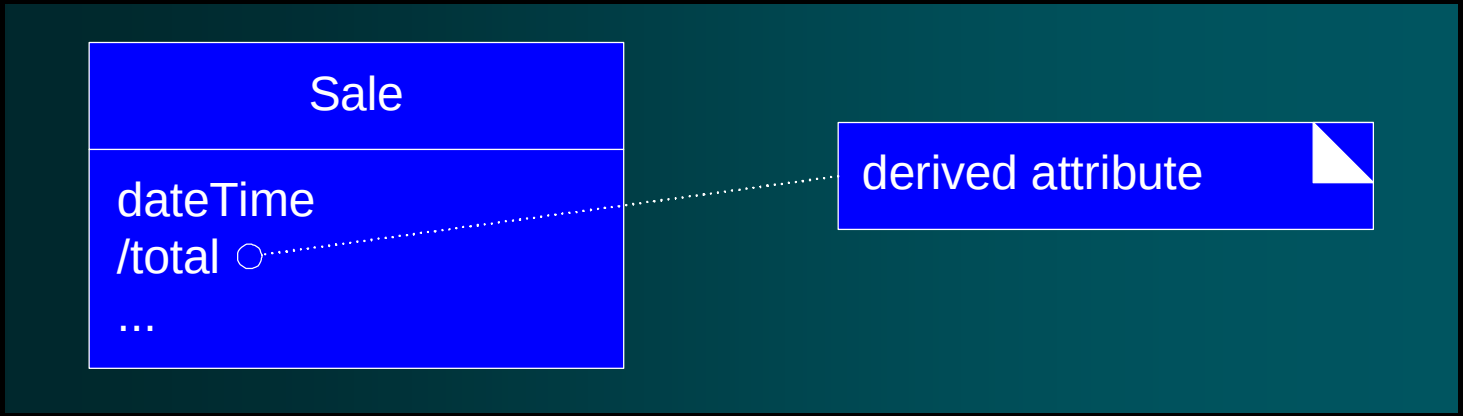


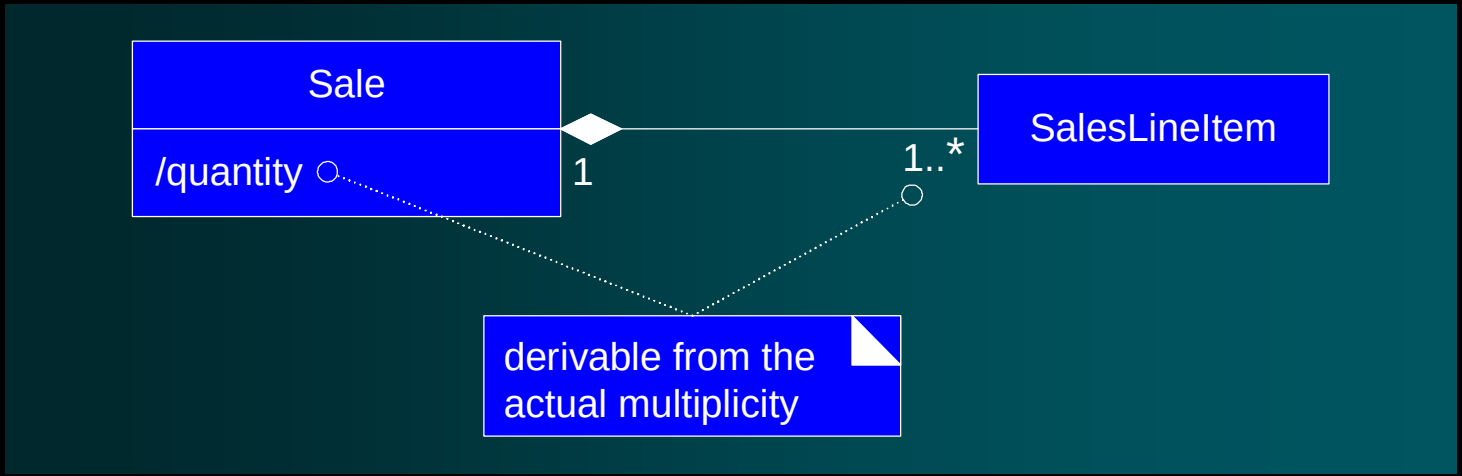
31.14. Roles as Concepts versus Roles in Associations

- Roles in associations
  - the same instance of a person takes on multiple roles in various associations
- Roles as concepts
  - provides ease and flexibility in adding unique attributes, associations, and additional semantics.
  - easier to implement

31.15. Derived Elements

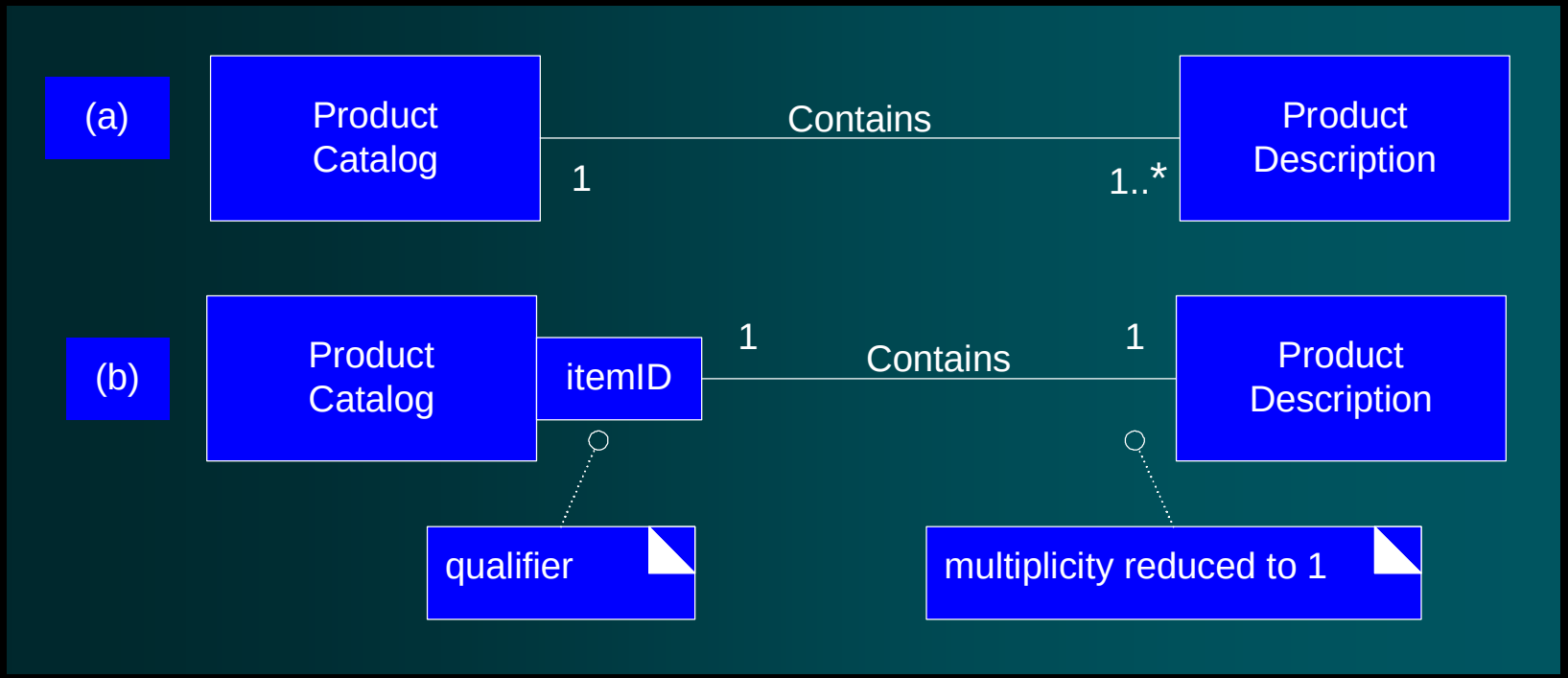
- A derived element can be determined from others.





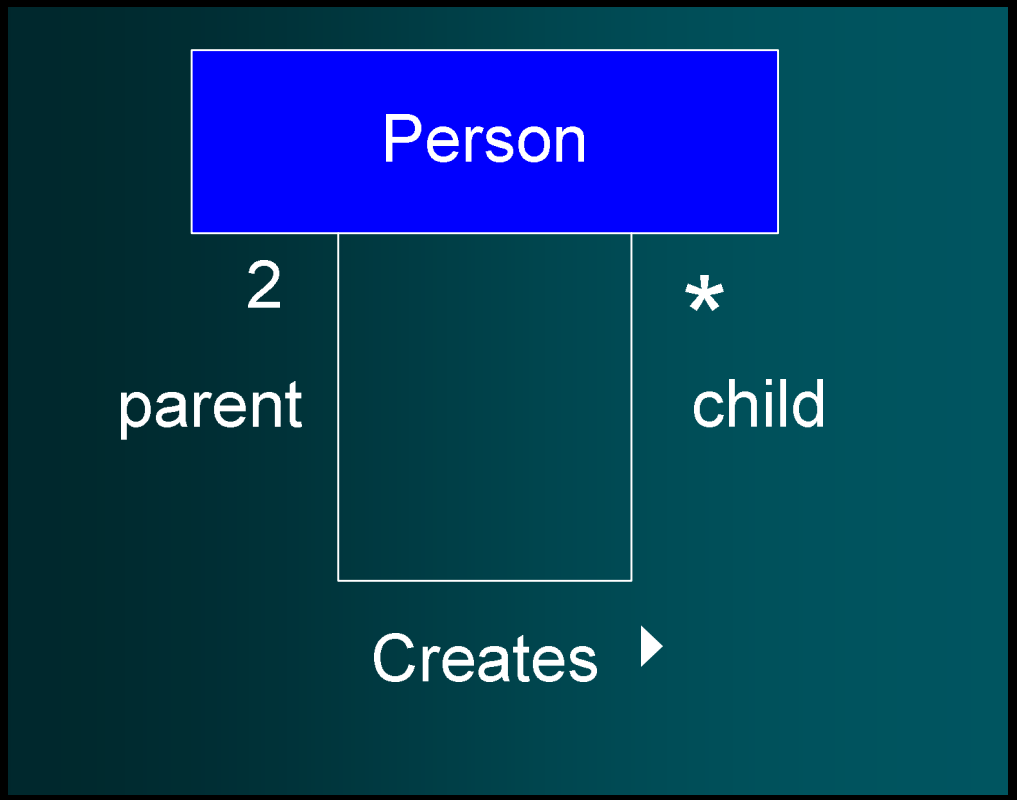
31.16. Qualified Associations

- Qualifier
  - used in an association;
  - distinguishes the set of objects at the far end of the association based on the qualifier value

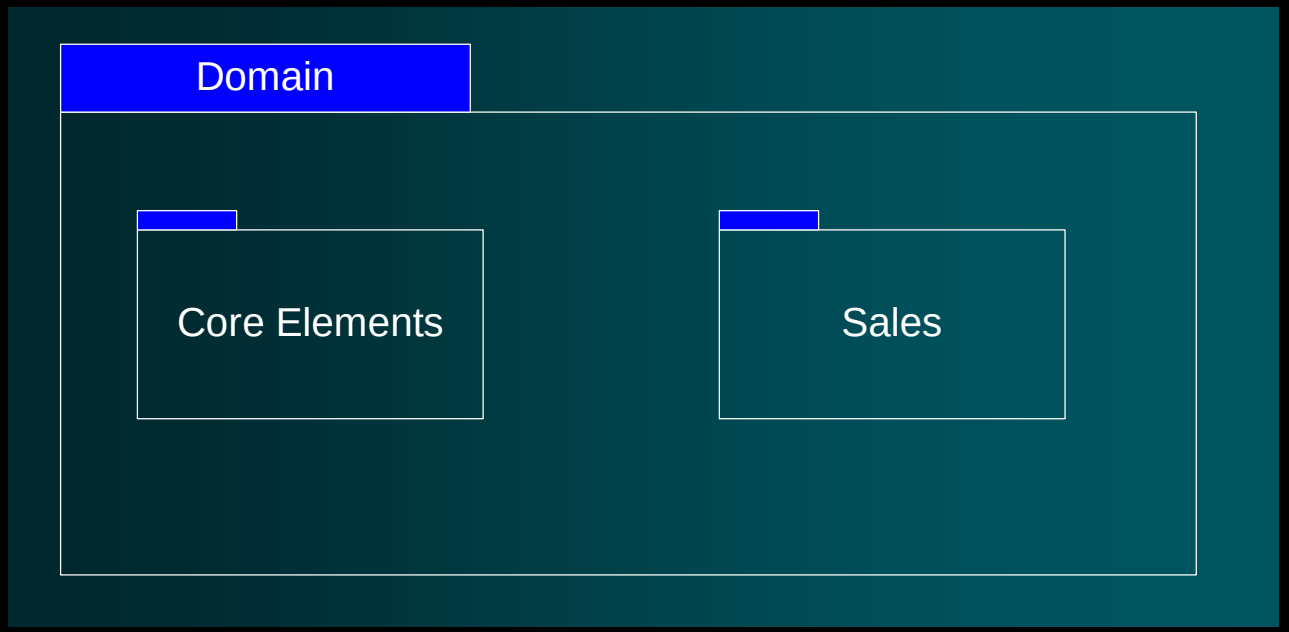


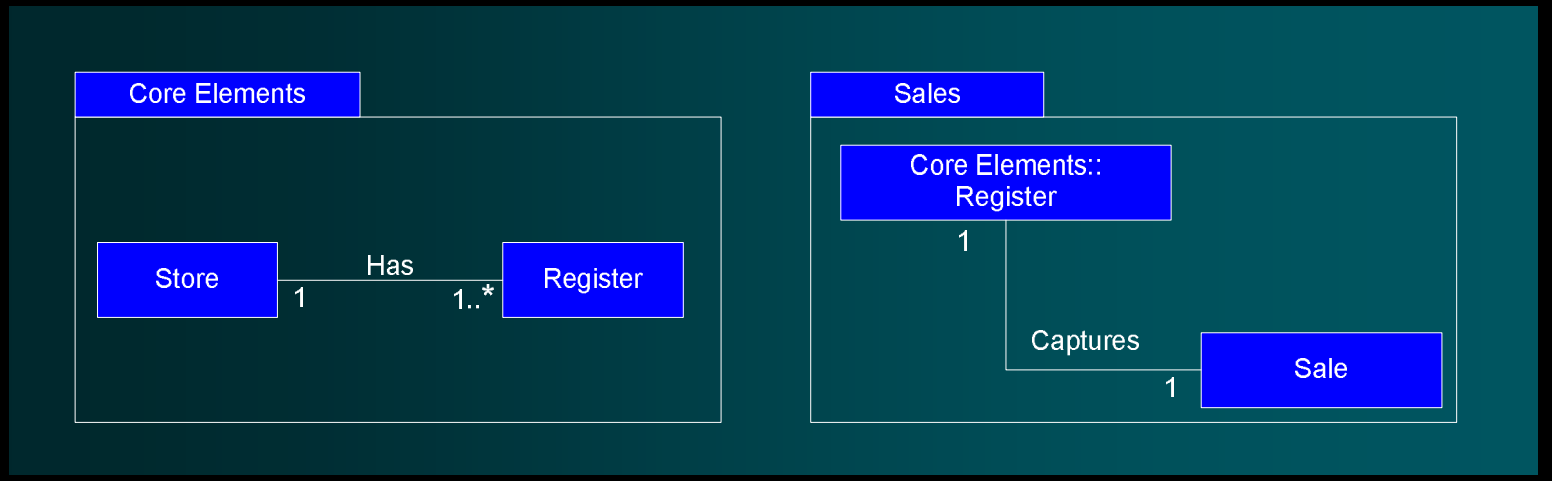
31.17. Reflexive Associations

- A concept may have an association to itself



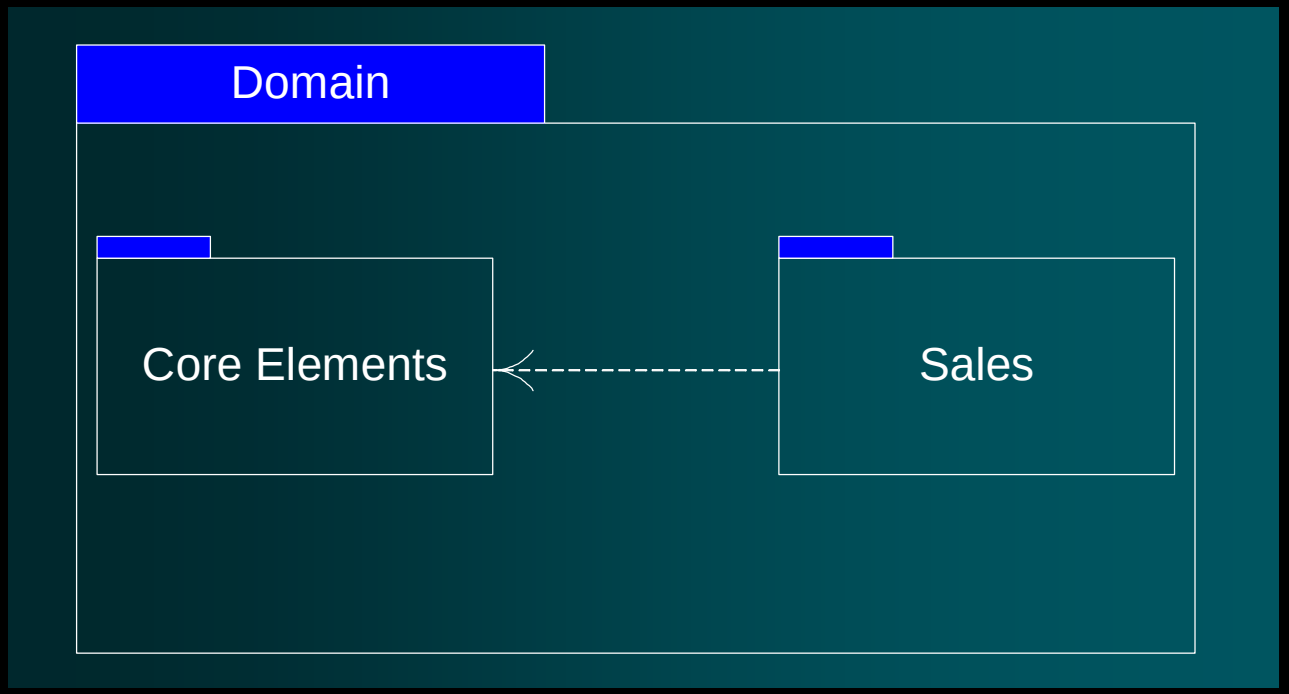
31.18. Using Packages to Organize the Domain Model





31.18. Using Packages to Organize the Domain Model

- Package Dependencies

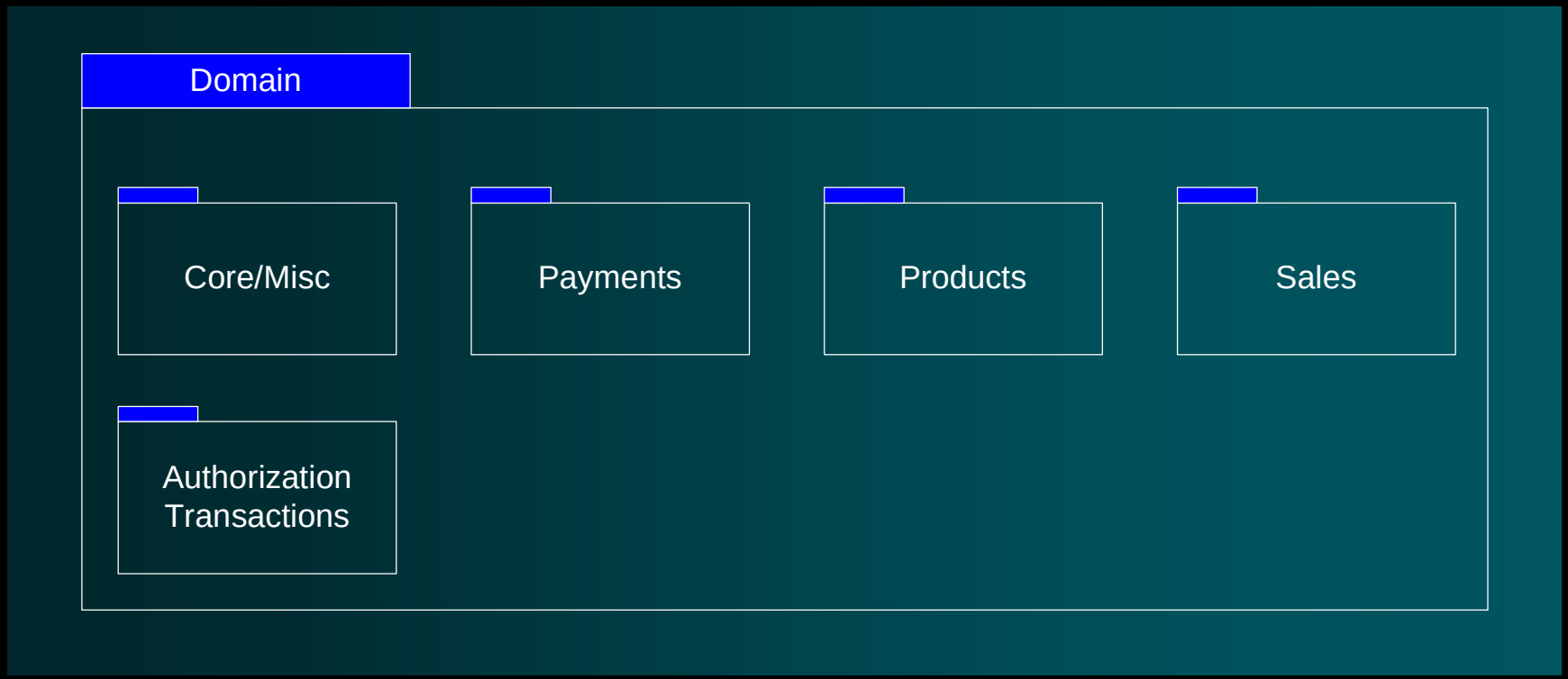


31.18. Using Packages to Organize the Domain Model

- How to Partition the Domain Model
  - Place elements together that:
    - are in the same subject area
    - are in a class hierarchy together
    - participate in the same use cases
    - are strongly associated

31.18. Using Packages to Organize the Domain Model

w POS Domain Model Packages

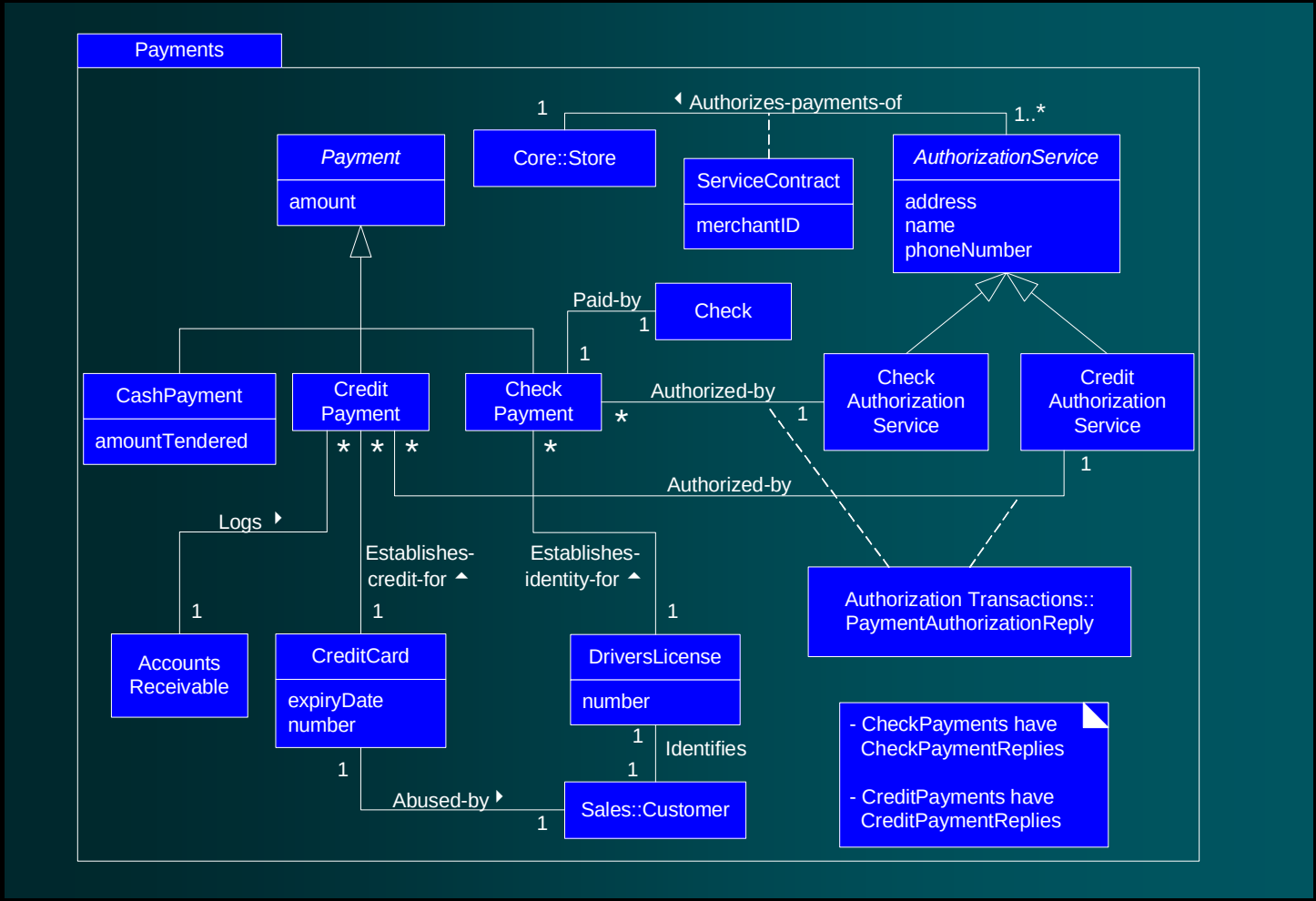


31.18. Using Packages to Organize the Domain Model

- Core/Misc Package
  - to own widely shared concepts or those without an obvious home.

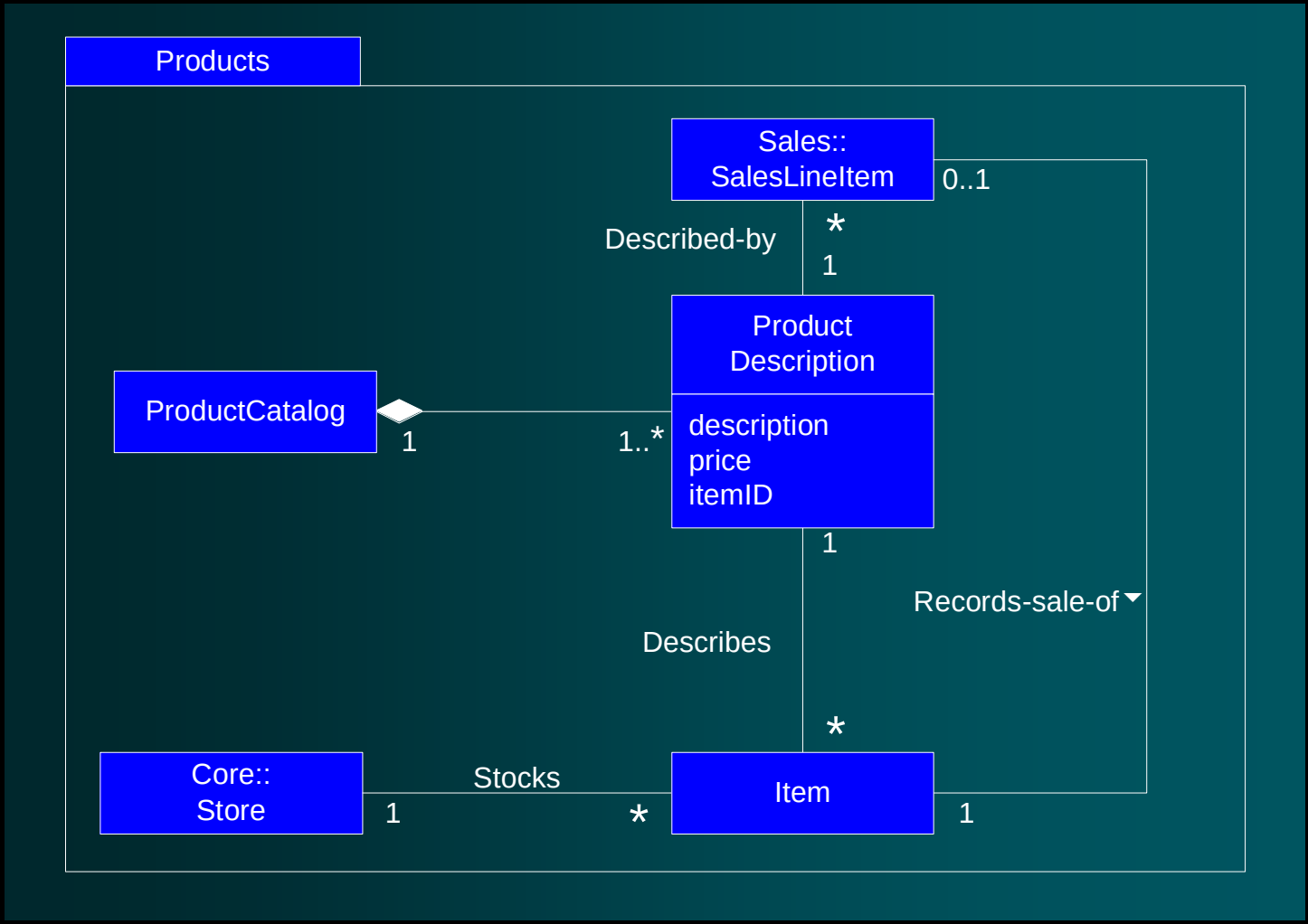
31.18. Using Packages to Organize the Domain Model

w Payments



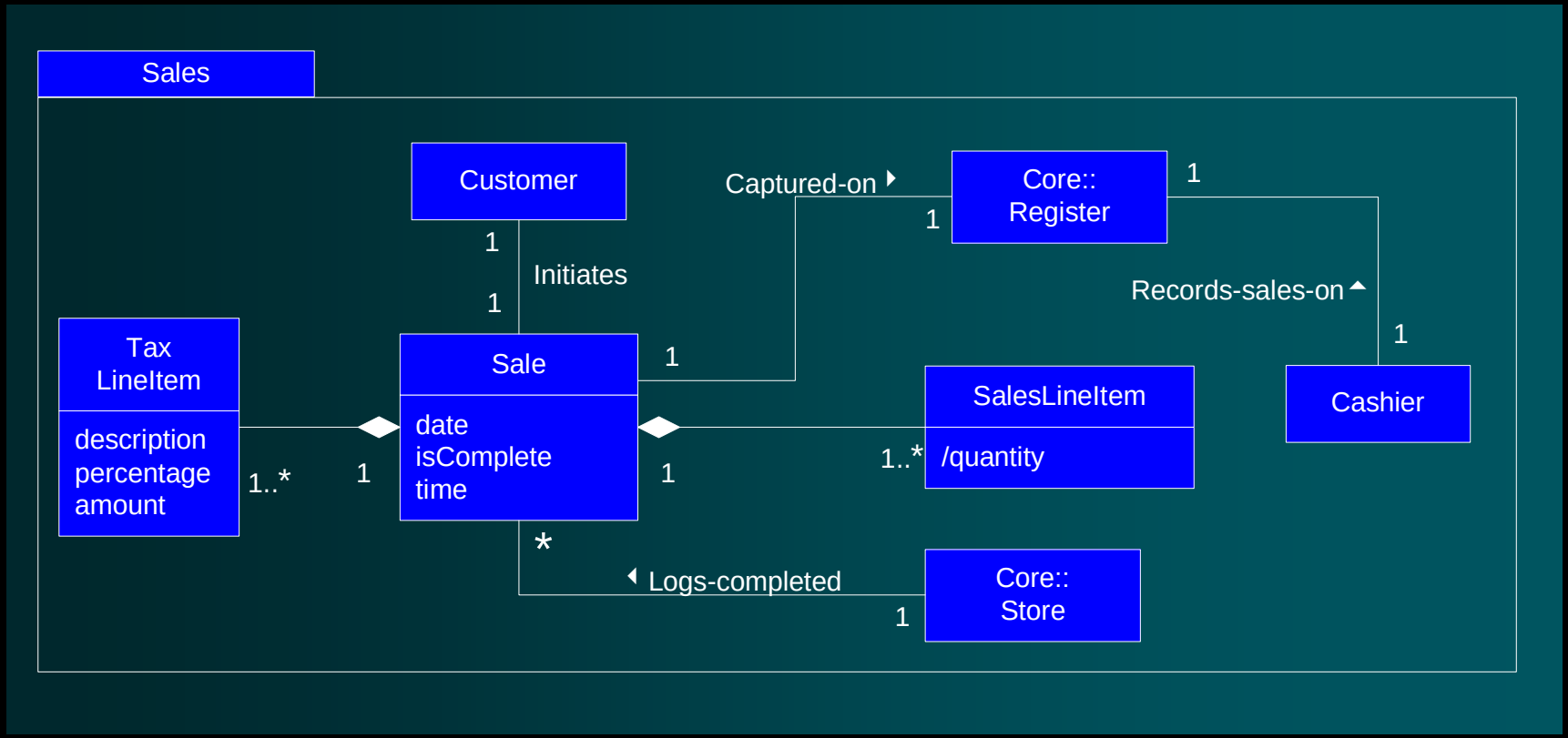
31.18. Using Packages to Organize the Domain Model

w Products



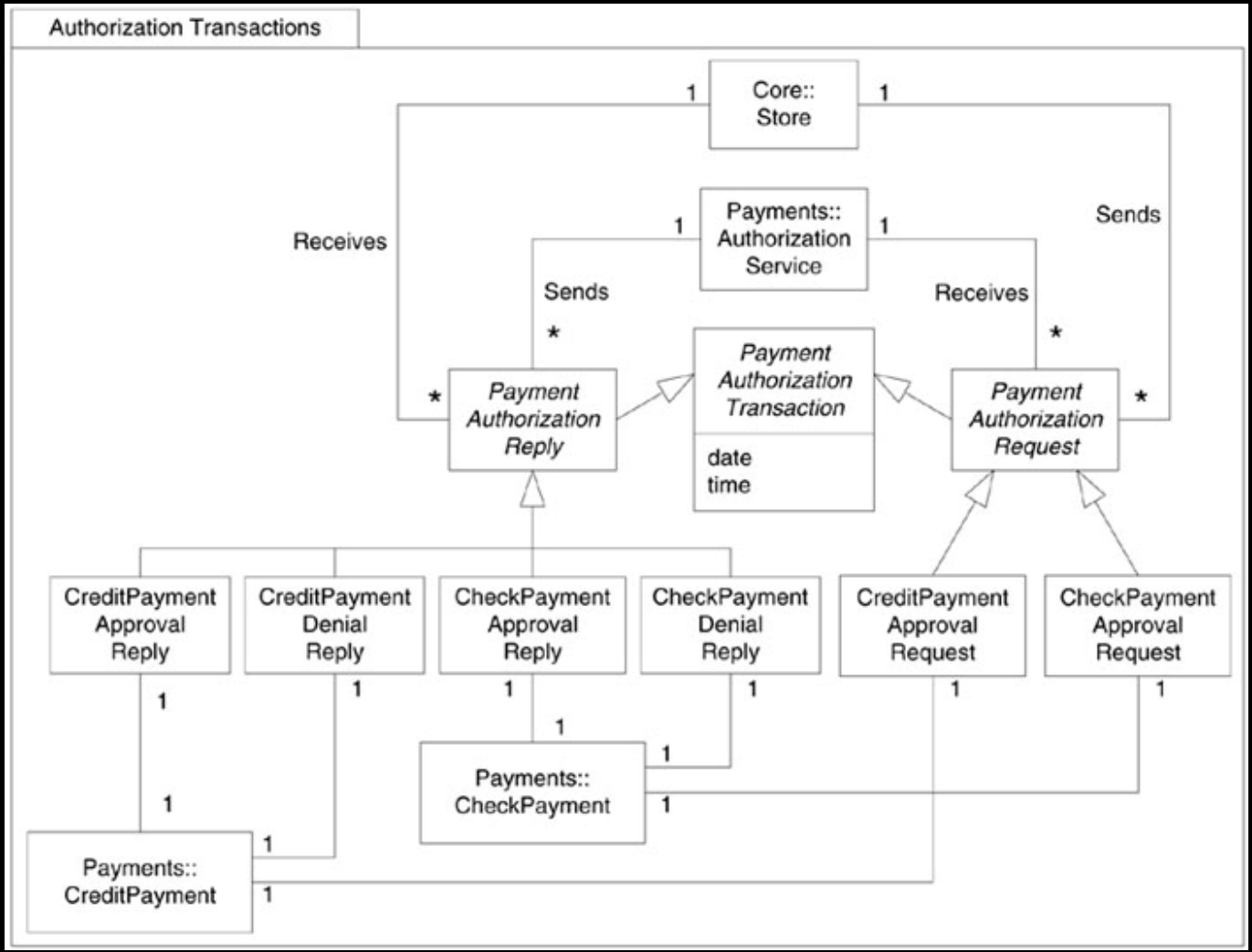
31.18. Using Packages to Organize the Domain Model

w Sales

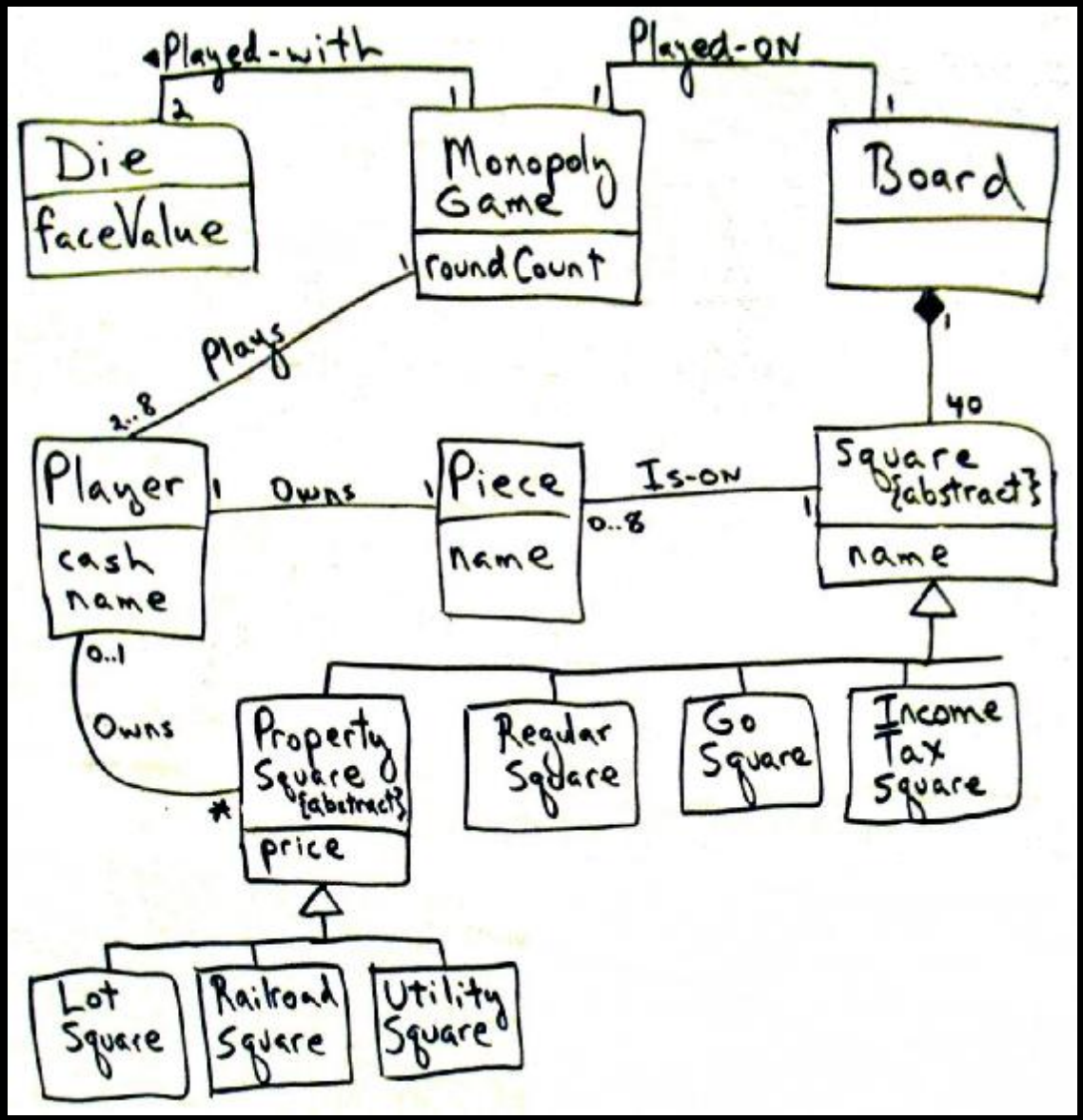


31.18. Using Packages to Organize the Domain Model

w Authorization Transactions



31.19. Example: Monopoly Domain Model Refinements



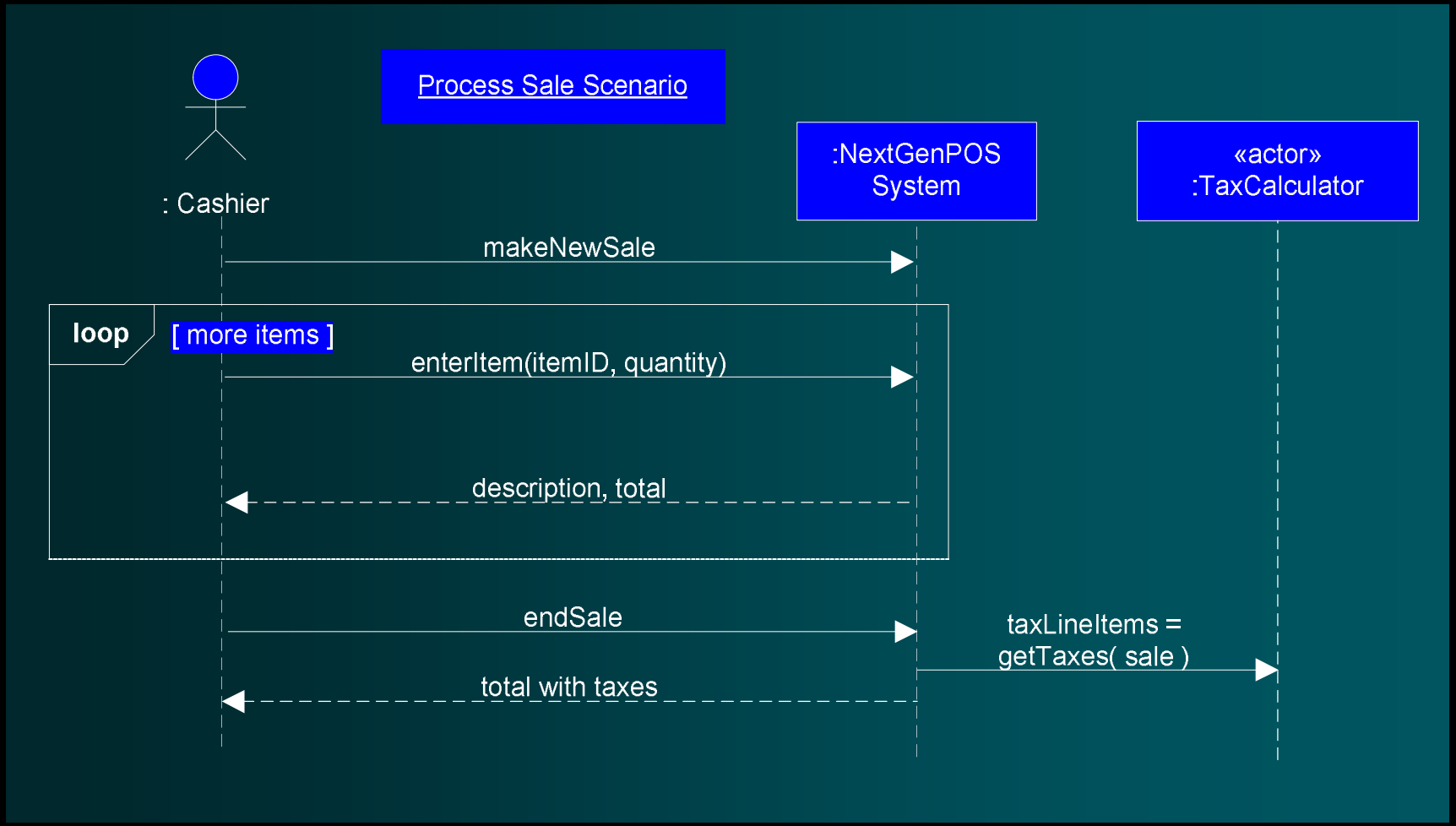
Chapter 32: More SSDs and Contracts

Objective

- Define SSDs and operation contracts for the current iteration.

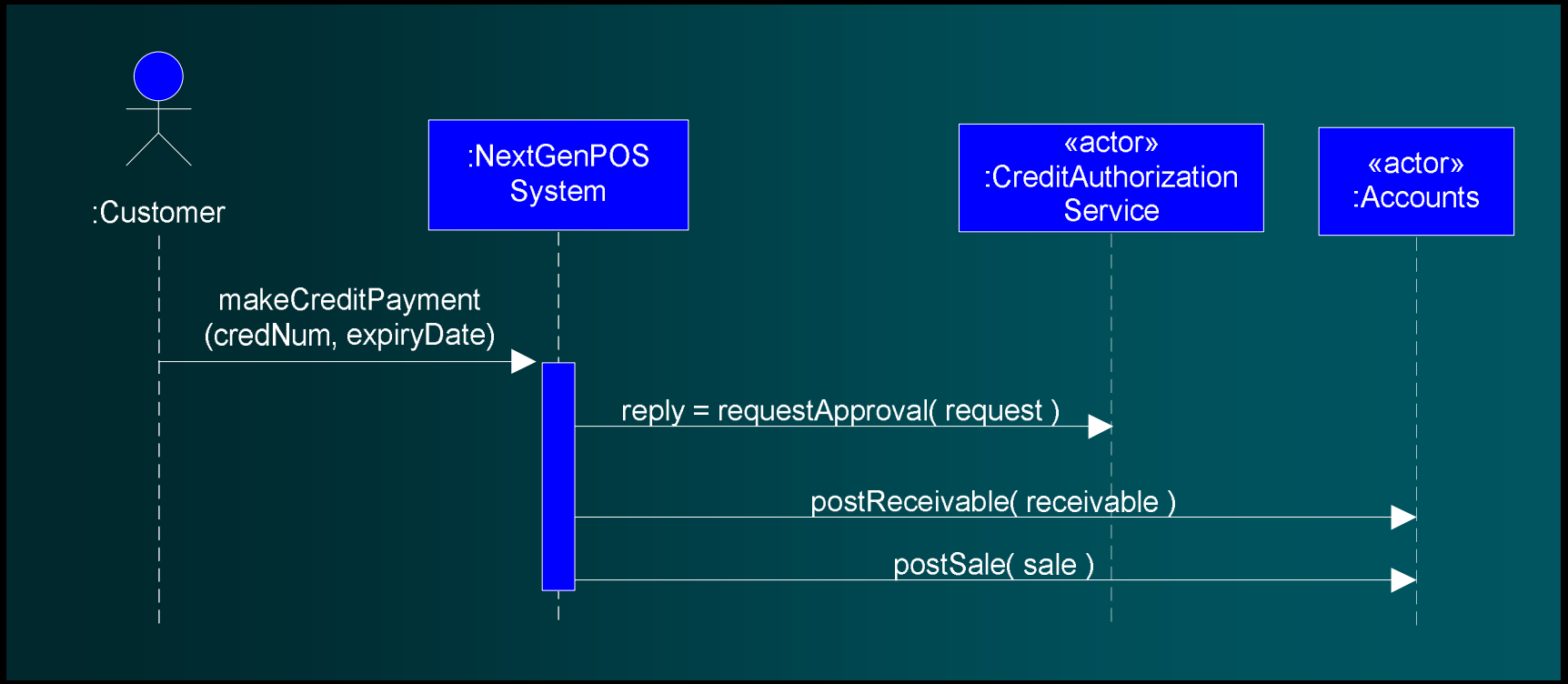
32.1. NextGen POS

w New System Sequence Diagrams



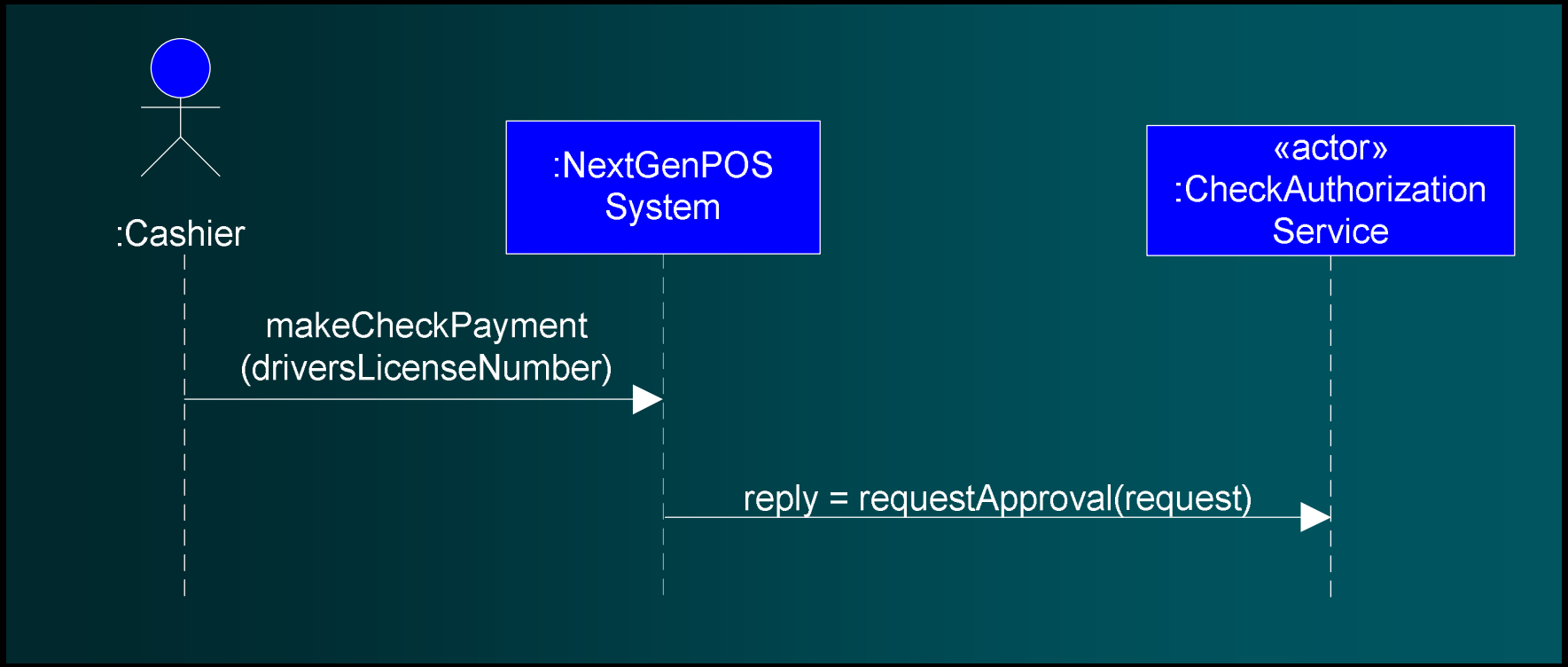
32.1. NextGen POS

w Credit Payment



32.1. NextGen POS

w Check Payment



32.1. NextGen POS

- **New System Operations**
  - makeCreditPayment
  - makeCheckPayment

Chapter 35: Package Design

Objective

- Organize packages to reduce the impact of changes.
- Know alternative UML package structure notation.

35.1 Package Organization Guidelines

- Package Functionally Cohesive Vertical and Horizontal Slice
  - Relational Cohesion

$$RC = \frac{NumberOfInternalRelations}{NumberOfTypes}$$

- A package of 6 types with 12 internal relations has RC=2.
- A package of 6 types with 3 intra-type relations has RC=0.5.

35.1 Package Organization Guidelines

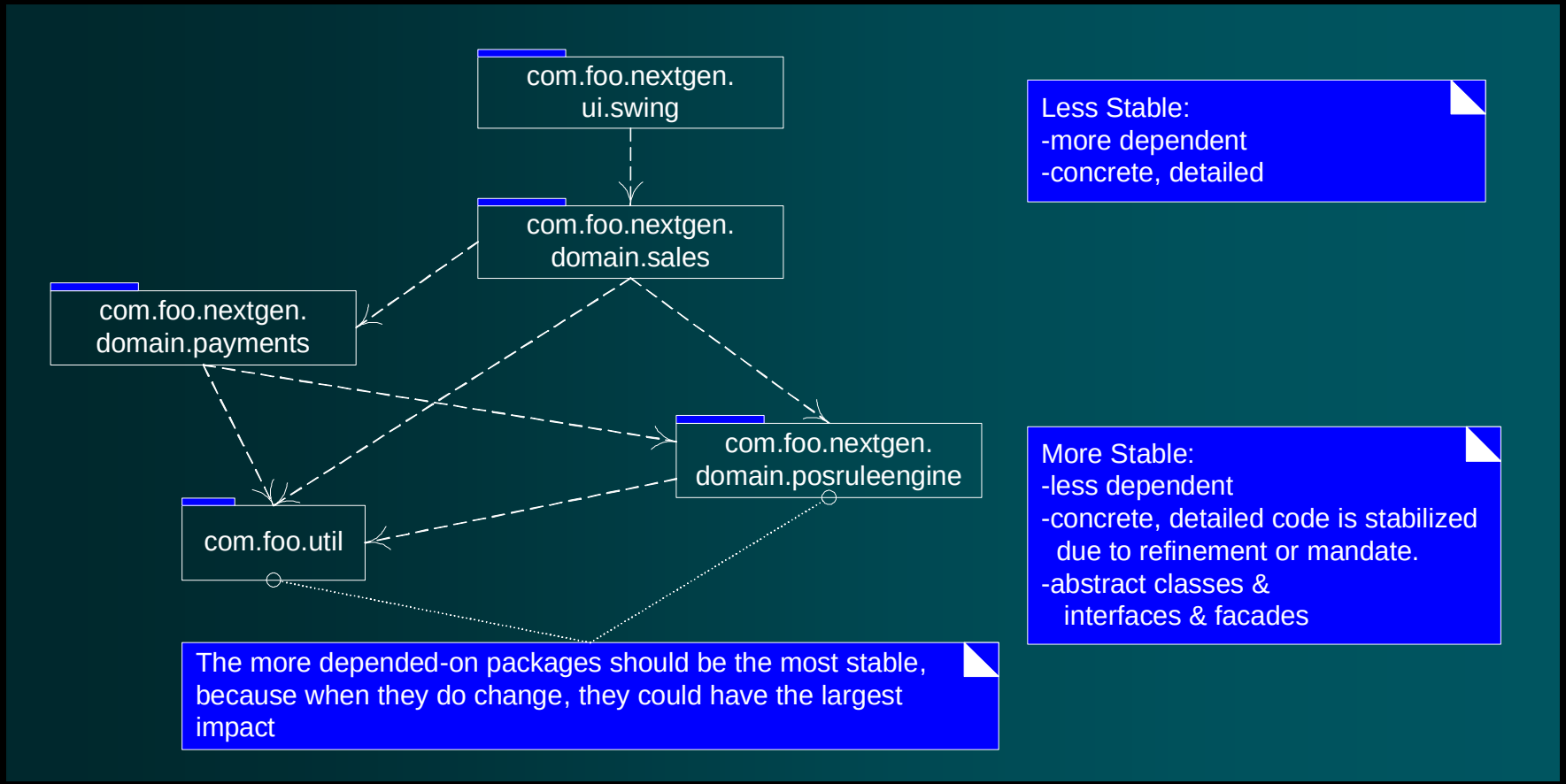
- A very low RC value suggests
  - The package contains unrelated things and is not factored well.
  - The package contains unrelated things and the designer deliberately does not care.
  - It contains one or more subset clusters with high RC, but overall does not.

35.1 Package Organization Guidelines

- Package a Family of Interfaces
  - javax.ejb
- Package by Work and by Clusters of Unstable Classes
  - Reduce widespread dependency on unstable packages.

35.1 Package Organization Guidelines

w Most Responsible Are Most Stable

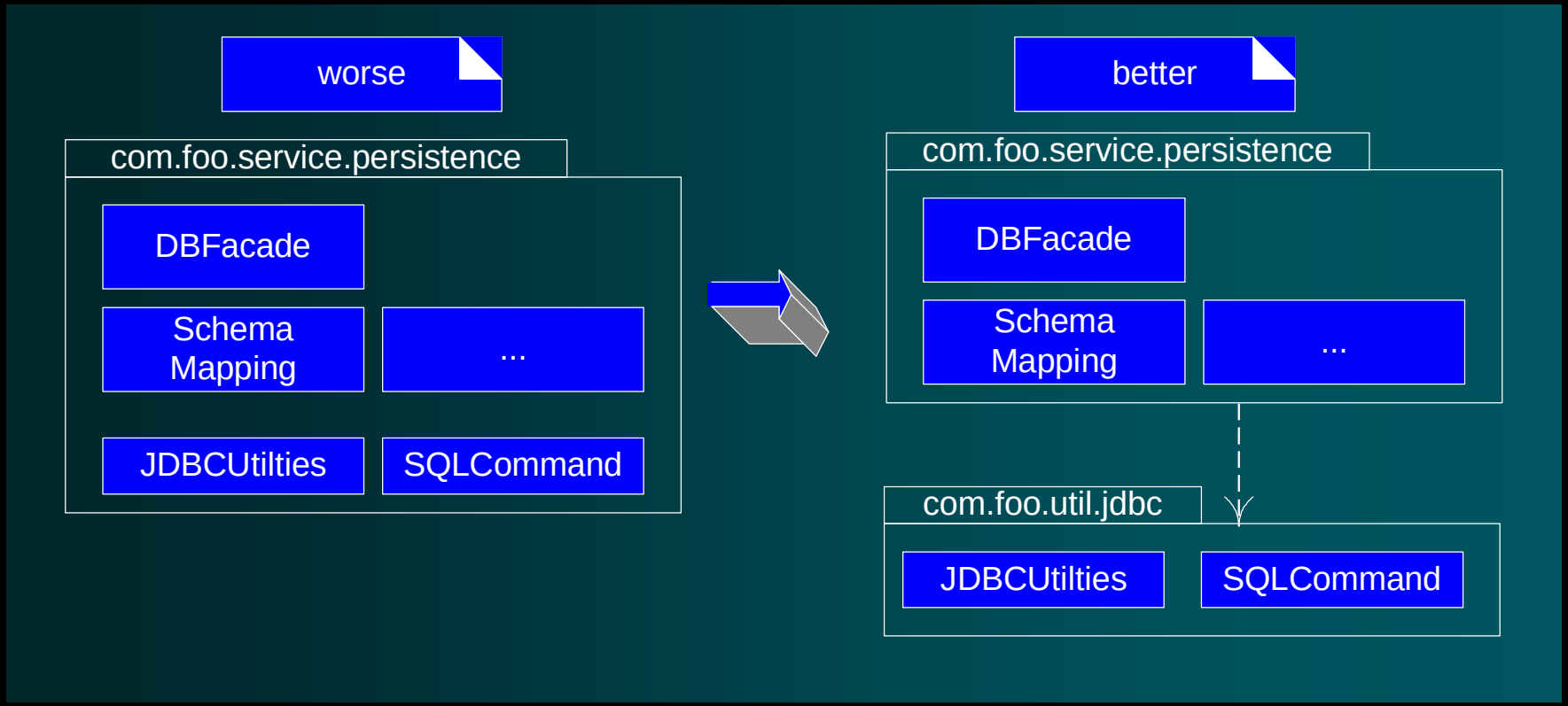


35.1 Package Organization Guidelines

- Different ways to increase stability in a package:
  - It contains only or mostly interfaces and abstract classes.
    - java.sql
  - it is independent, or it depends on other very stable packages, or it encapsulates its dependencies such that dependents are not affected.
    - com.foo.necxtgen.domain.postruleengine
  - It contains relatively stable code because it was well-exercised and refined before release.
    - java.util
  - It is mandated to have a slow change schedule.
    - java.lang

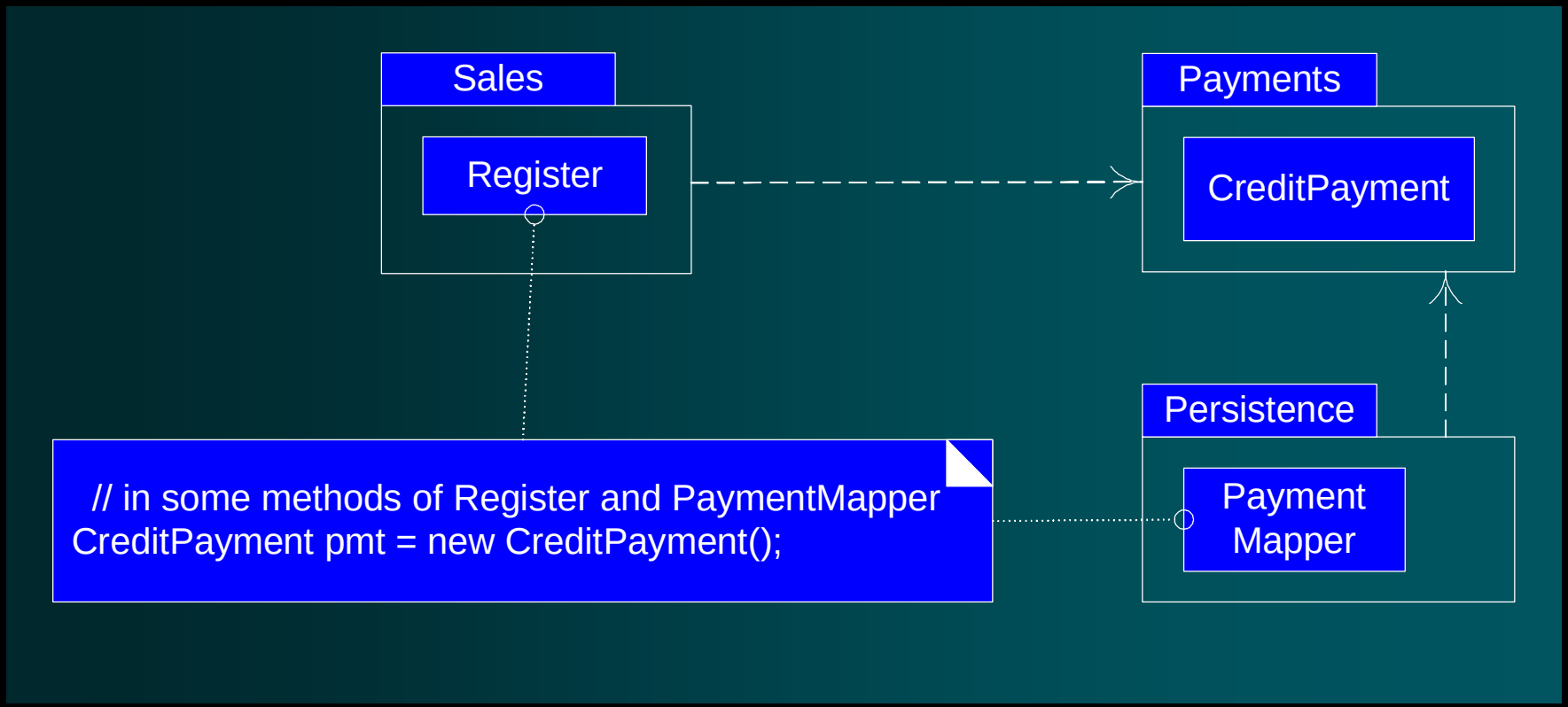
35.1 Package Organization Guidelines

w Factor out Independent Types

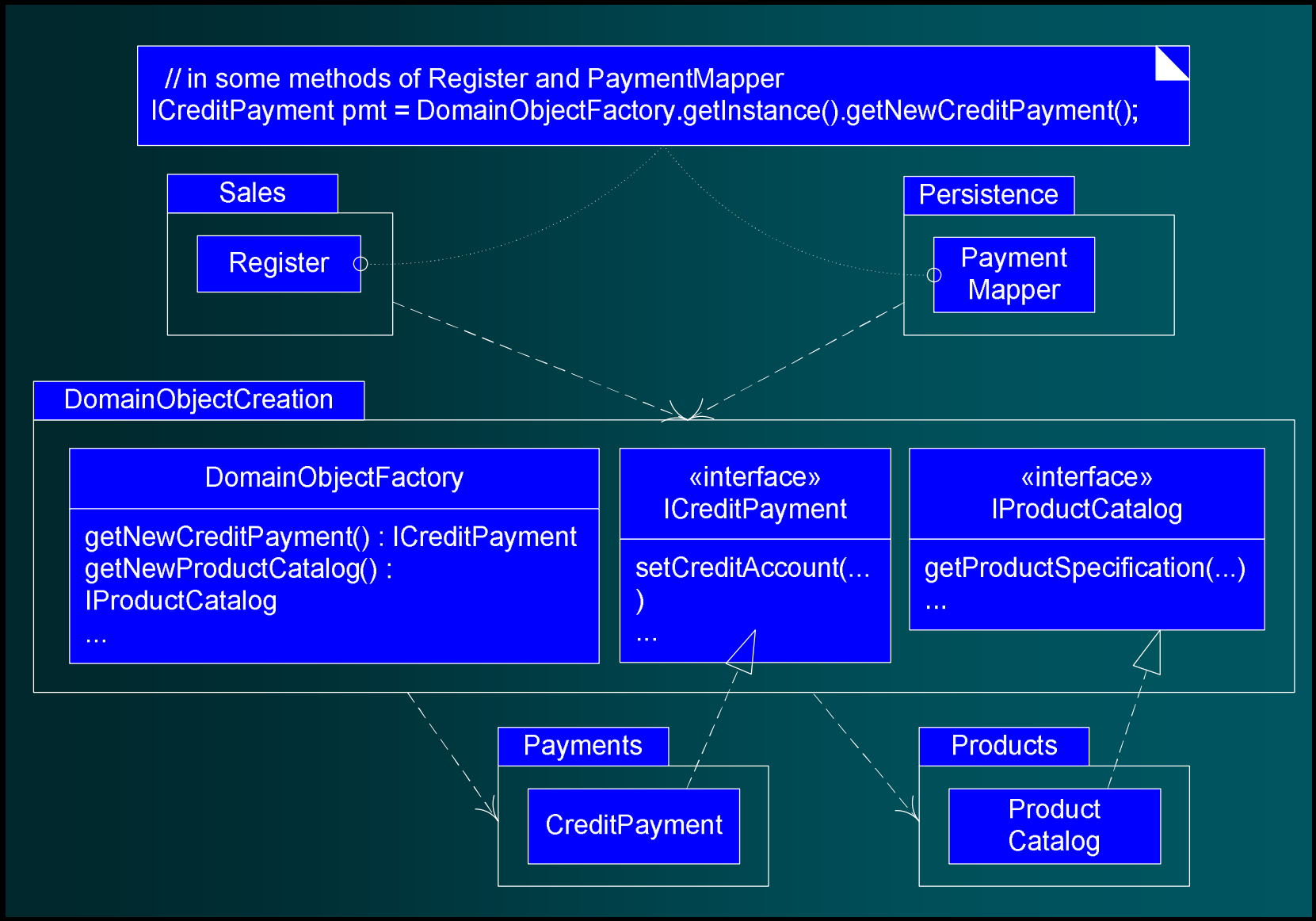


35.1 Package Organization Guidelines

- Use Factories to Reduce Dependency on Concrete Packages



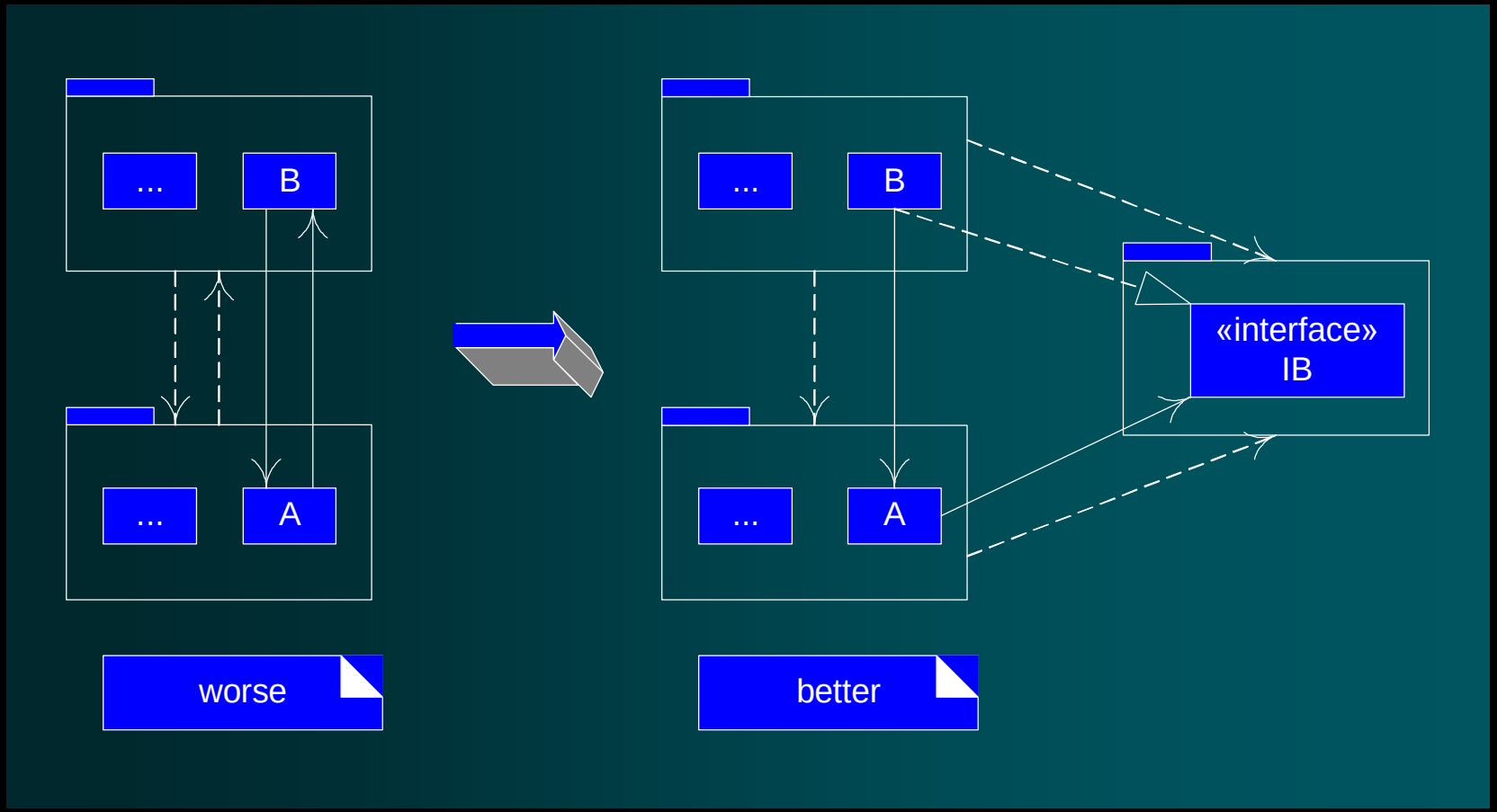
35.1 Package Organization Guidelines



35.1 Package Organization Guidelines

- No Cycles in Packages
  - Need to be treated as one larger package in terms of a release unit.
  - Factor out the types participating in the cycle into a new smaller package.
  - Break the cycle with an interface.
    - Redefine the depended-on classes in one of the packages to implement new interfaces.
    - Define the new interfaces in a new package.
    - Redefine the dependent types to depend on the interfaces in the new package, rather than the original classes

35.1 Package Organization Guidelines



Chapter 36. More Object Design with GoF Patterns

Objective

- Apply GoF and GRASP in the design of the use-case realizations.

36.1. Example: NextGen POS

- 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

36.2. Failover to Local Services; Performance with Local Caching

- 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
  - Offer local implementations of remote services.
    - use constant tax rates.
  - Cache a small set of the most common products.

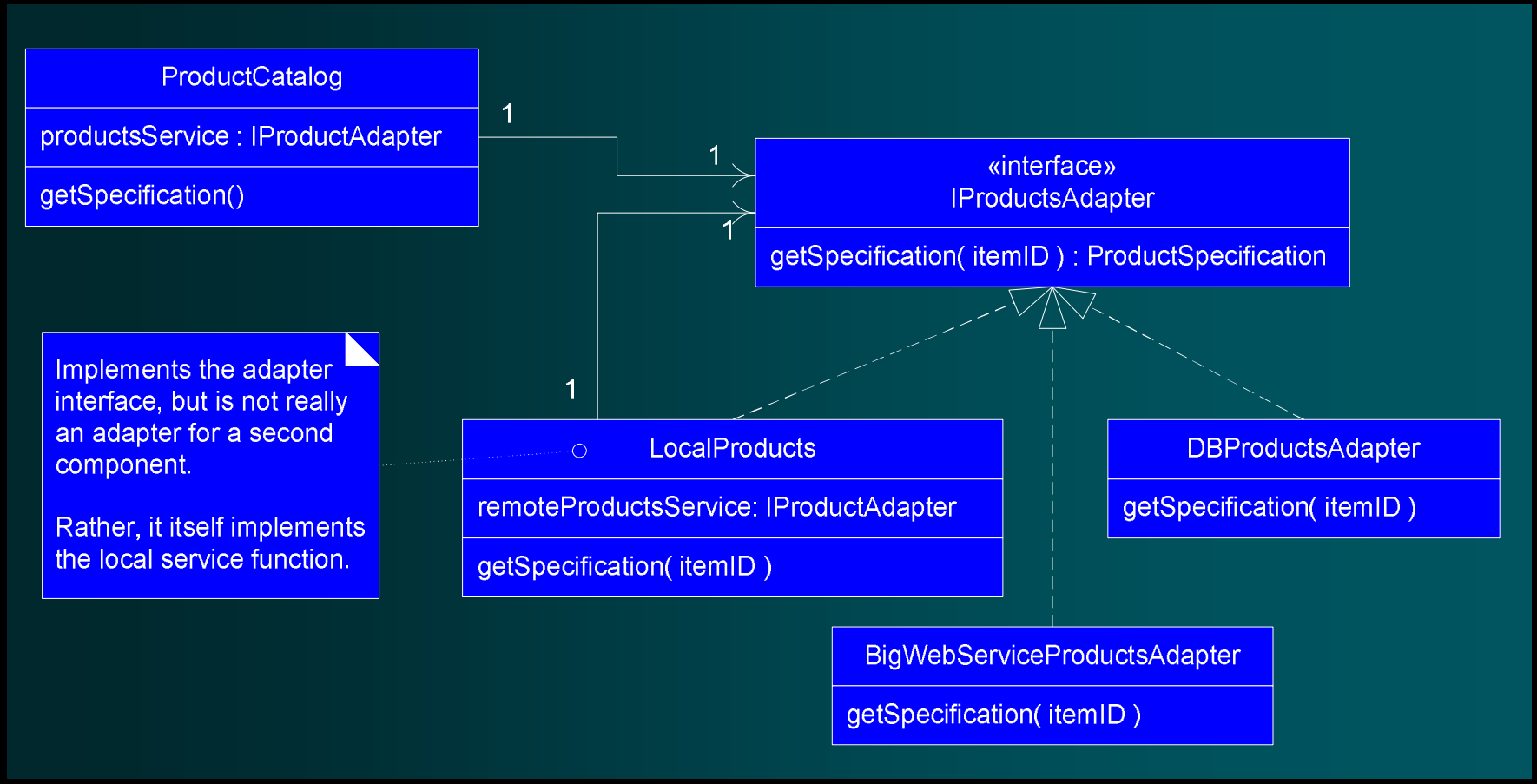
36.2. Failover to Local Services; Performance with Local Caching

- Can be achieved with our existing adapter and factory design:
  - The ServicesFactory will always return an adapter to a local product information service.
  - The local products "adapter" itself implement the responsibilities of the local service.
  - The local service is initialized to a reference to a second adapter to the true remote product service.
  - If the local service finds the data in its cache, it returns it; otherwise, it forwards the request to the adapter for the external service.

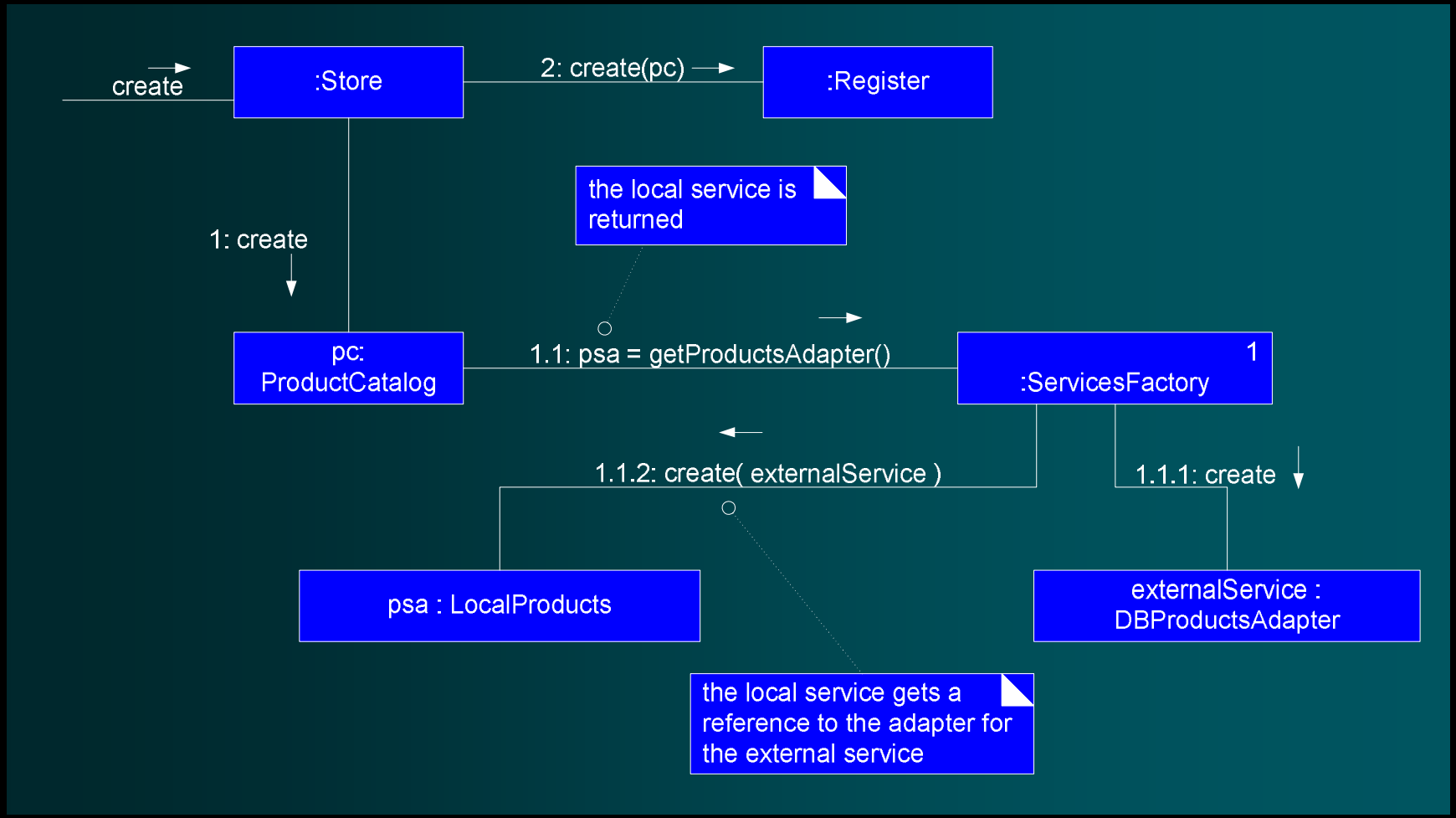
36.2. Failover to Local Services; Performance with Local Caching

- Two levels of client-side cache:
  - The in-memory ProductCatalog object
    - maintain an in-memory collection (such as a Java HashMap) of some (for example, 1,000) ProductDescription objects.
  - A larger persistent (hard disk based) cache
    - maintains some quantity of product information (such as 1 or 100MB of file space).

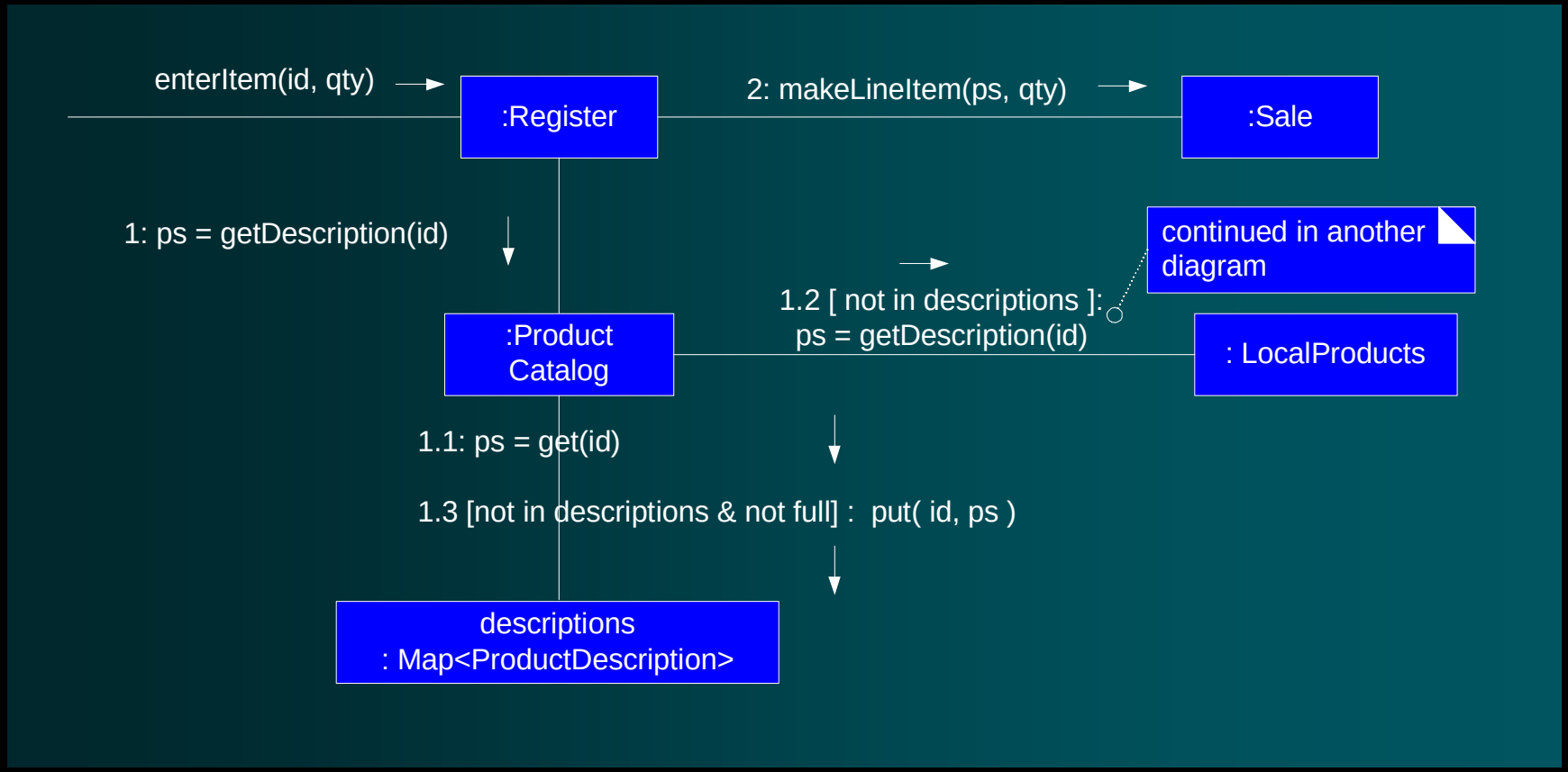
36.2. Failover to Local Services; Performance with Local Caching



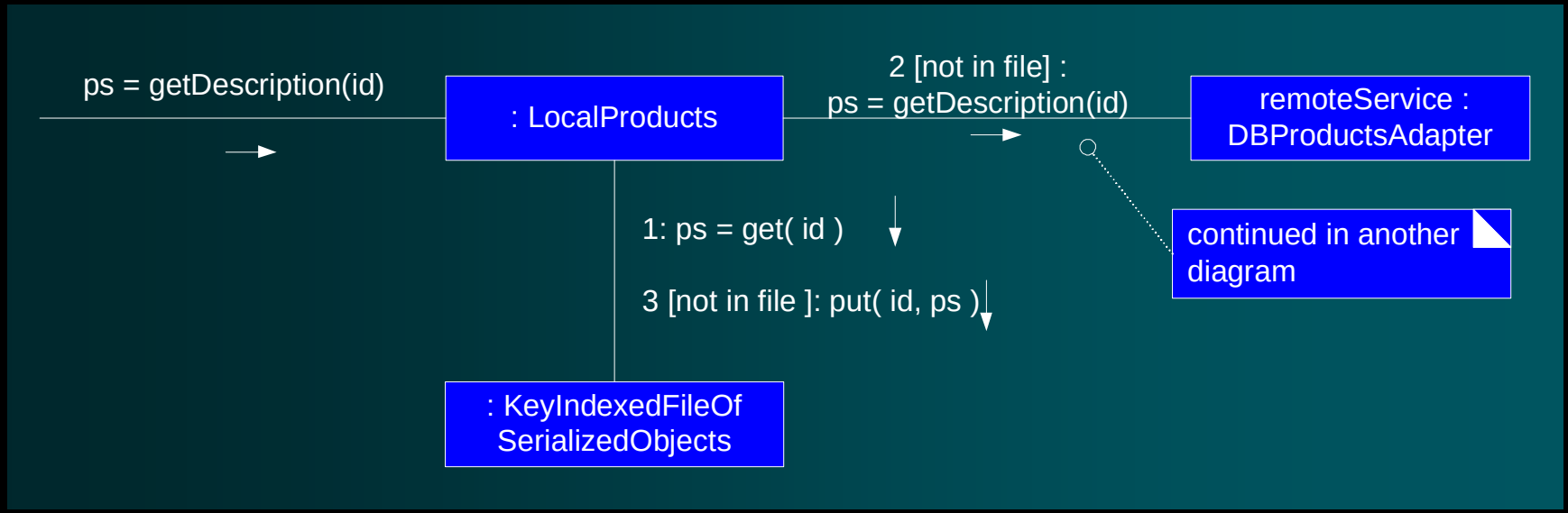
36.2. Failover to Local Services; Performance with Local Caching



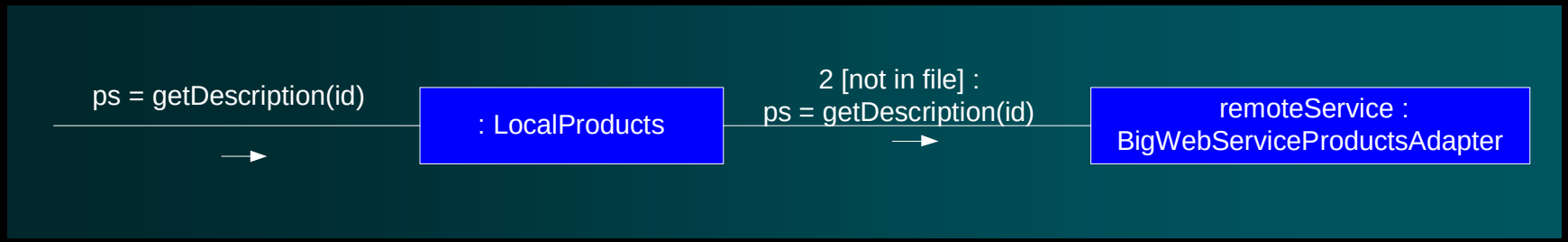
36.2. Failover to Local Services; Performance with Local Caching



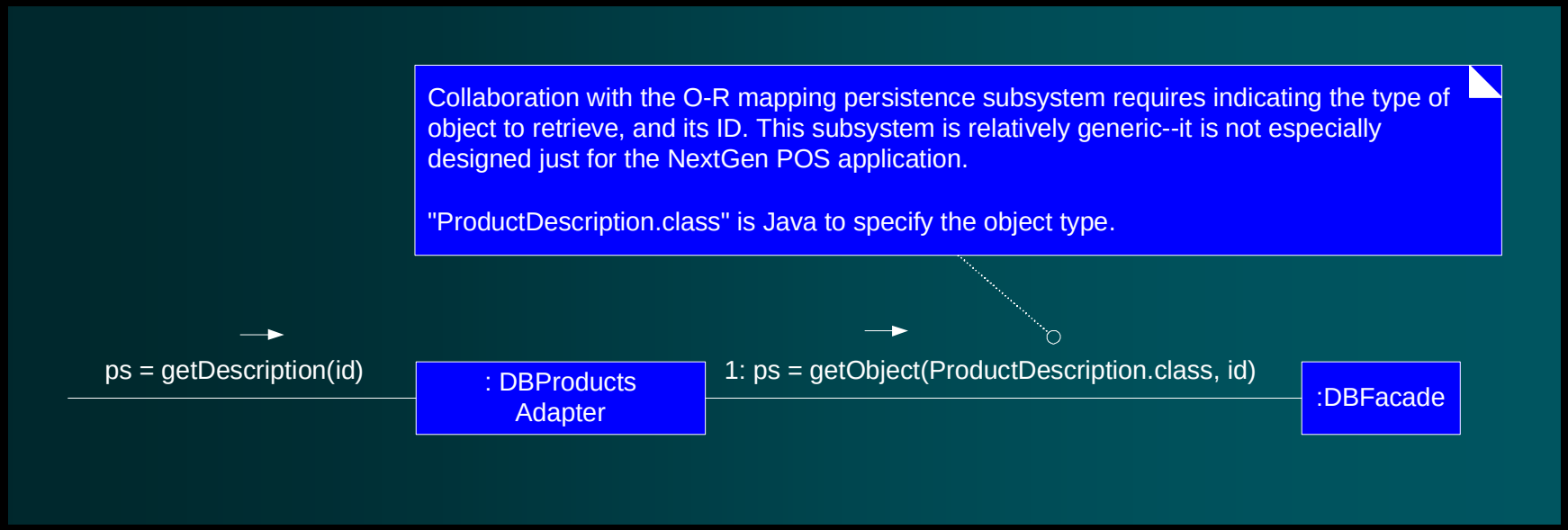
36.2. Failover to Local Services; Performance with Local Cachings



If the true external service was changed from a database to a new Web service



36.2. Failover to Local Services; Performance with Local Cachings

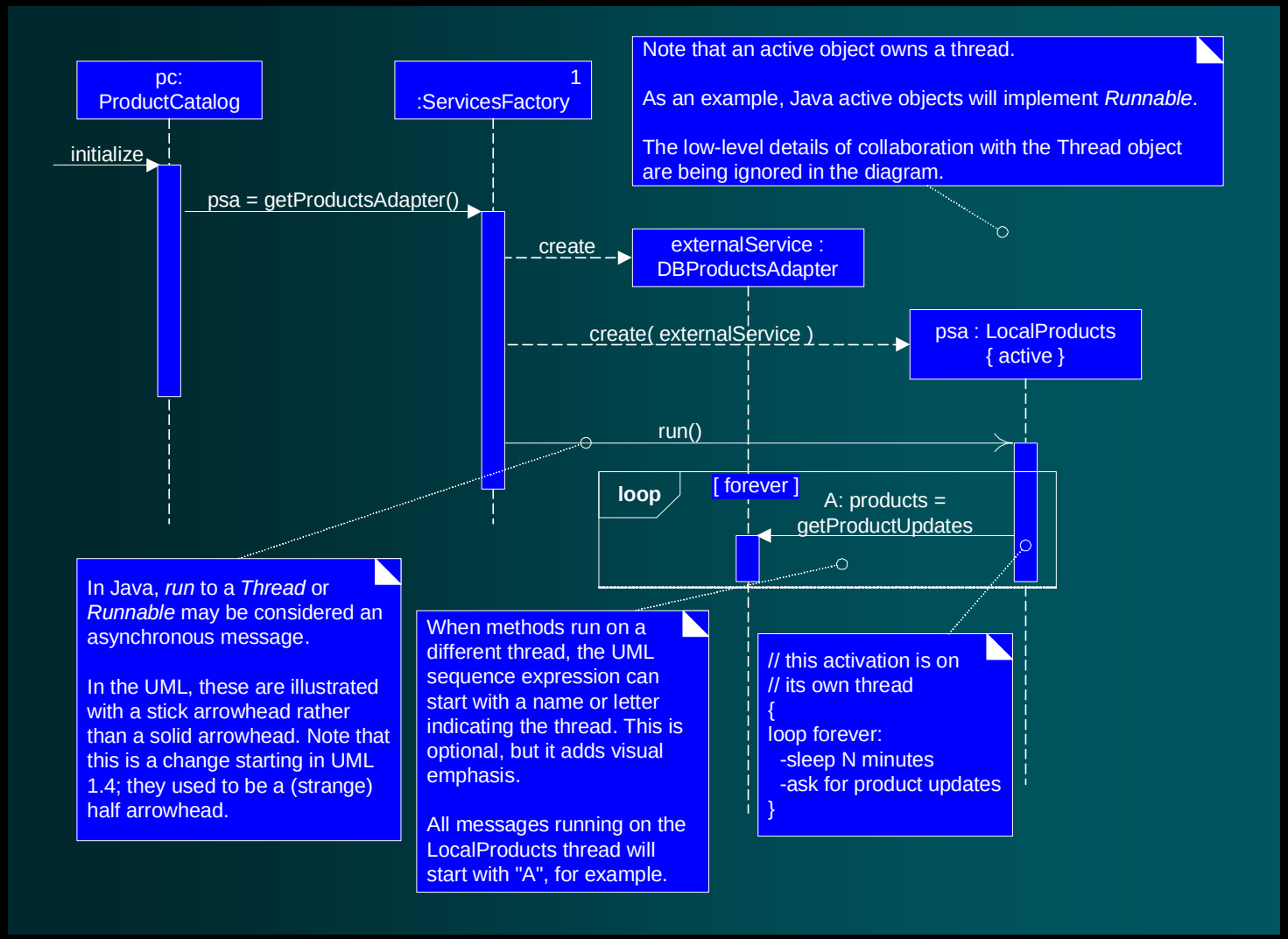


36.2. Failover to Local Services; Performance with Local Cachings

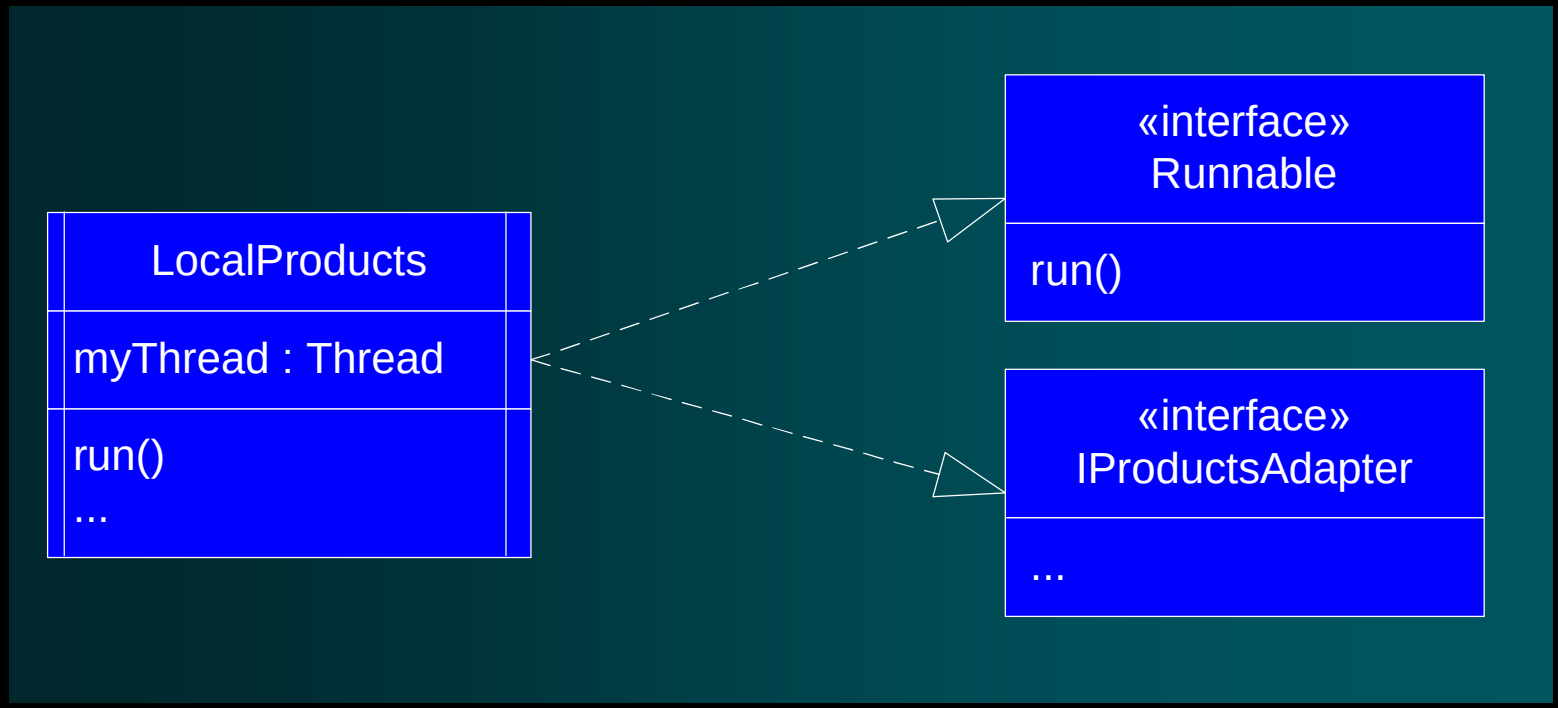
- Caching Strategies
  - Lazy initialization
    - the caches fill slowly as external product information is retrieved
  - Eager initialization
    - the caches are loaded during the StartUp use case
- **Stale Cache**
  - add a remote service operation that answers today's current changes;
  - the LocalProducts object queries it every n minutes and updates its cache.

36.2. Failover to Local Services; Performance with Local Cachings

w Threads in the UML



36.2. Failover to Local Services; Performance with Local Cachings



36.3. Handling Failure

- What to do in the case where there isn't a local cache hit and access to the external products service fails?
  - signals the cashier to manually enter the price and description, or cancel the line item entry.

36.3. Handling Failure

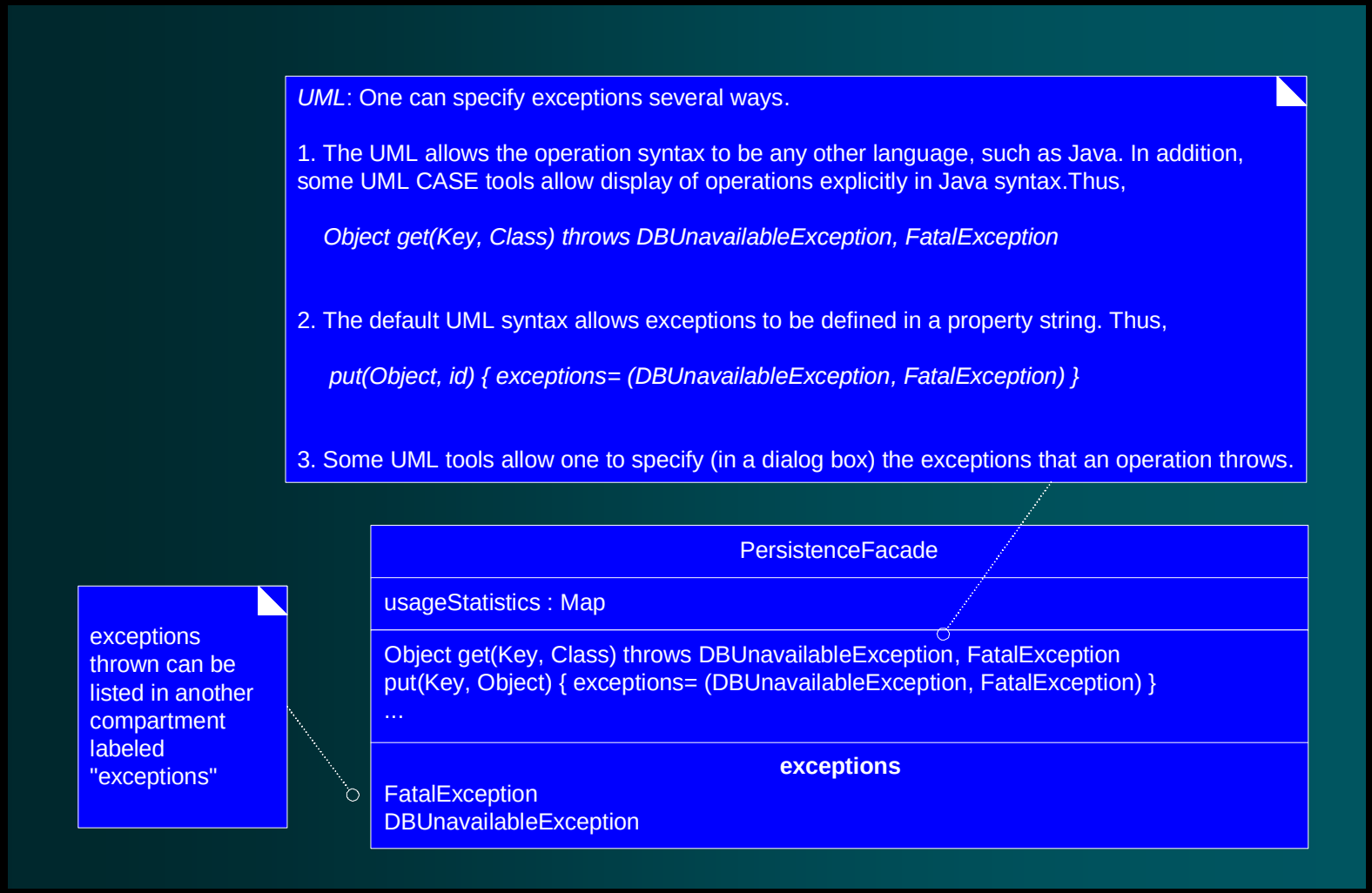
- Error
  - A manifestation of the fault in the running system. Errors are detected (or not).
    - When calling the naming service to obtain a reference to the database (with the misspelled name), it signals an error.
- Failure
  - A denial of service caused by an error.
    - The Products subsystem (and the NextGen POS) fails to provide a product information service.

36.3. Handling Failure

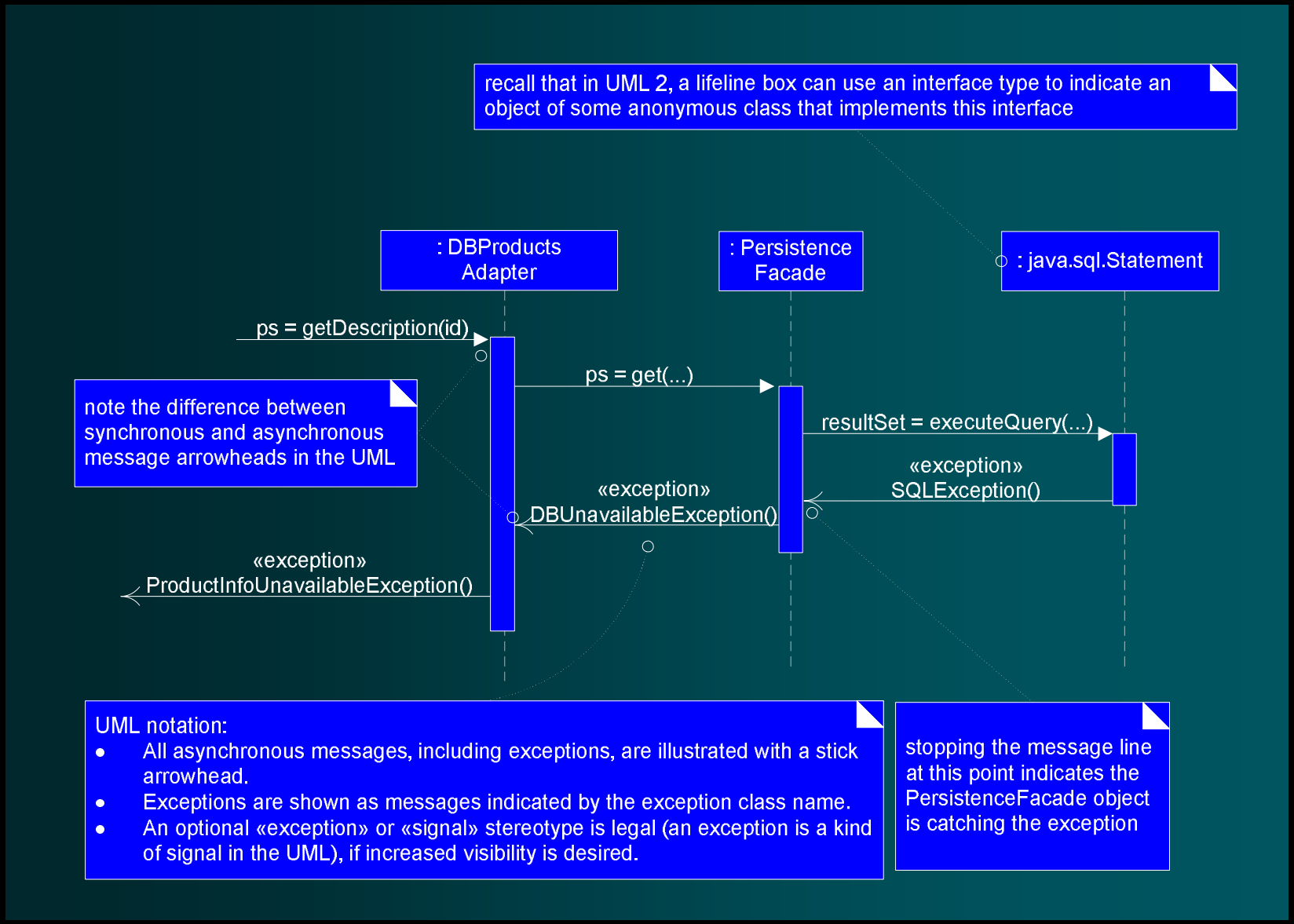
- Throwing Exceptions
  - Should the original exception be thrown all the way up to the presentation layer?
- Some common exception handling patterns
  - Convert Exceptions
    - Convert the lower level exception into high level exception
      - SQLException->DBUnavailableException->ProductInfoUnavailableException
  - Name The Problem Not The Thrower
    - Assign a name that describes why the exception is being thrown

36.3. Handling Failure

w Exceptions in the UML



36.3. Handling Failure

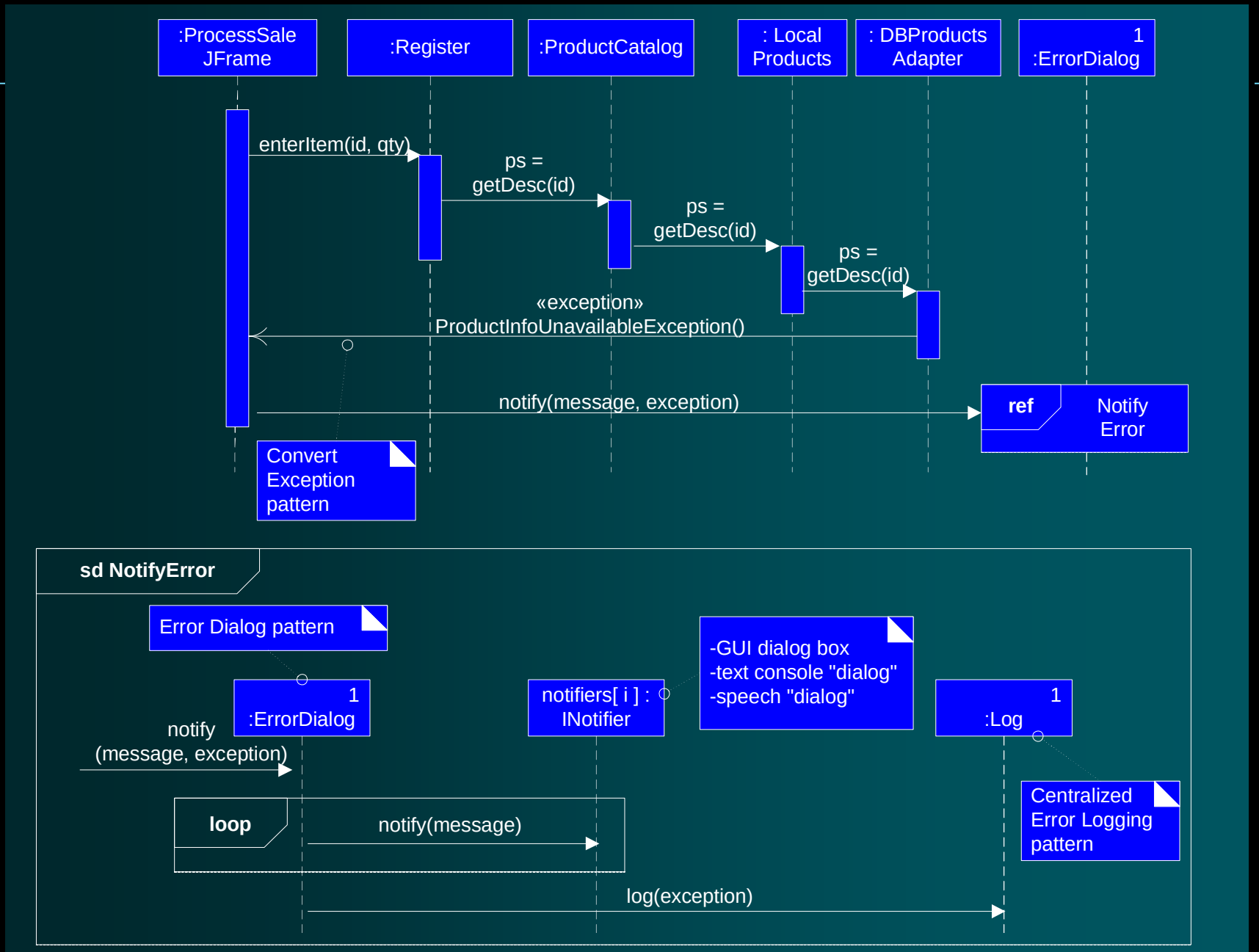


36.3. Handling Failure

- Handling Errors
  - Centralized Error Logging
    - Use a Singleton-accessed central error logging object and report all exceptions to it.
      - Consistency in reporting.
      - Flexible definition of output streams and format

36.3. Handling Failure

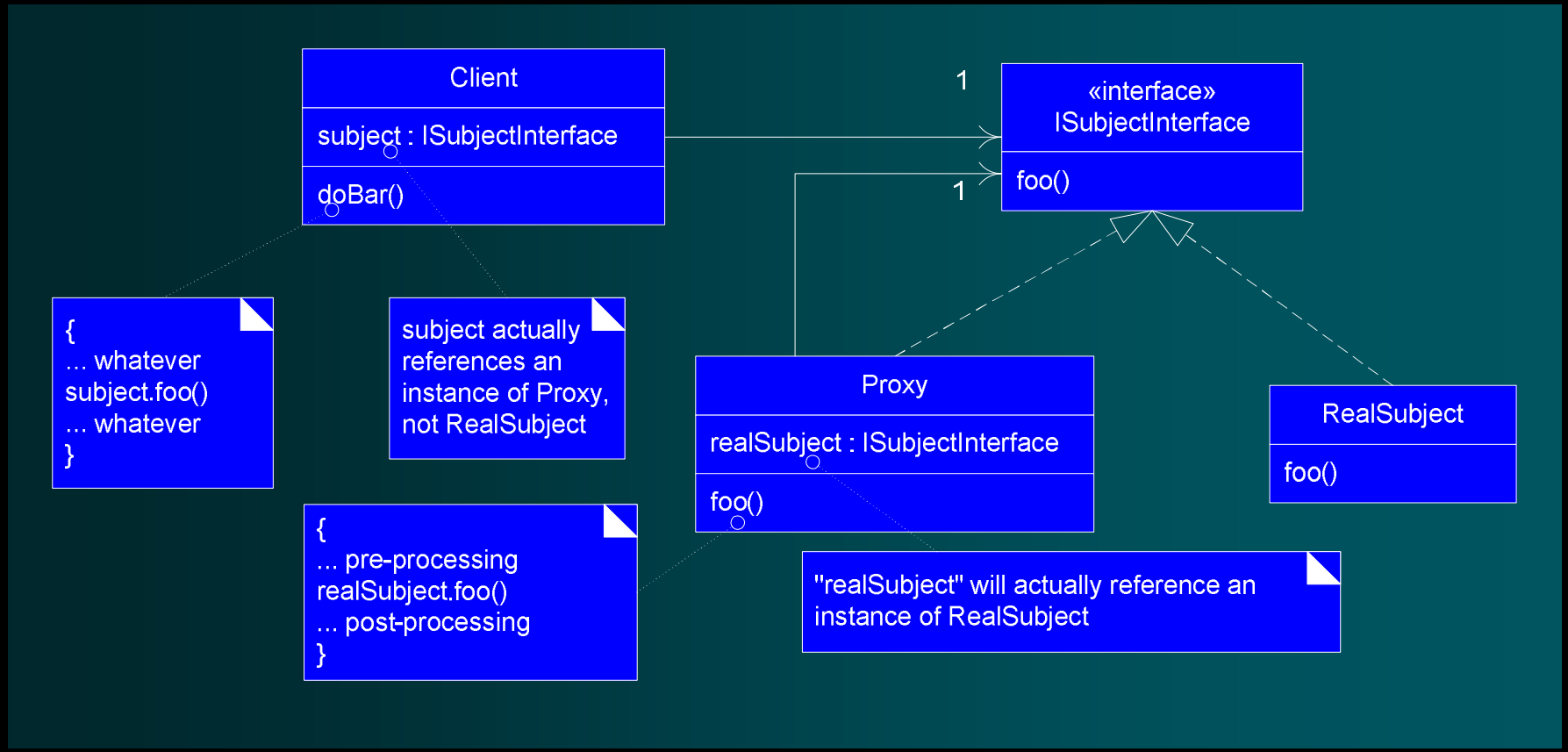
- Error Dialog
  - Use a standard Singleton-accessed, application-independent, non-UI object to notify users of errors.
    - Protected Variations with respect to changes in the output mechanism.
    - Consistent style of error reporting;
    - Centralized control of the common strategy for error notification.
    - Minor performance gain;
      - hide and cache a Error Dialog for recycled use, rather than recreate a dialog for each error.


:ProcessSale JFrame
:Register
:ProductCatalog
: Local Products
: DBProducts Adapter
1 :ErrorDialog
enterItem(id, qty)
ps = getDesc(id)
ps = getDesc(id)
ps = getDesc(id)
«exception» ProductInfoUnavailableException()
notify(message, exception)
ref
Notify Error
Convert Exception pattern
sd NotifyError
Error Dialog pattern
1 :ErrorDialog
notifiers[i] : INotifier
-GUI dialog box
-text console "dialog"
-speech "dialog"
1 :Log
notify (message, exception)
loop
notify(message)
Centralized Error Logging pattern
log(exception)

36.4. Failover to Local Services with a Proxy (GoF)

- Failover to a local service for the product information
  - achieved by inserting the local service in front of the external service
- Sometimes the external service should be tried first, and a local version second.
  - Posting of sales to the accounting service

36.4. Failover to Local Services with a Proxy (GoF)

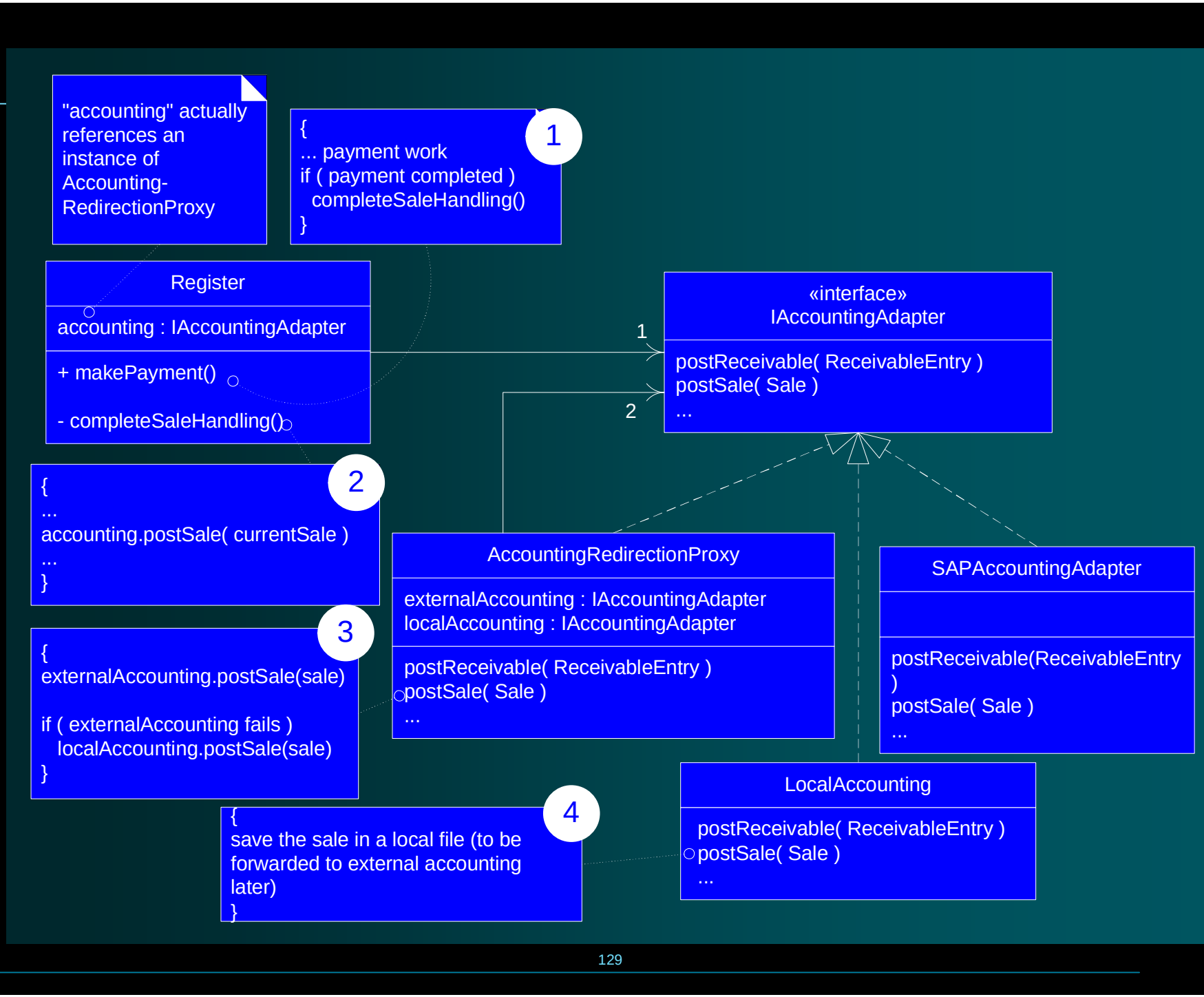


36.4. Failover to Local Services with a Proxy (GoF)

- **Proxy**
  - 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.

36.4. Failover to Local Services with a Proxy (GoF)

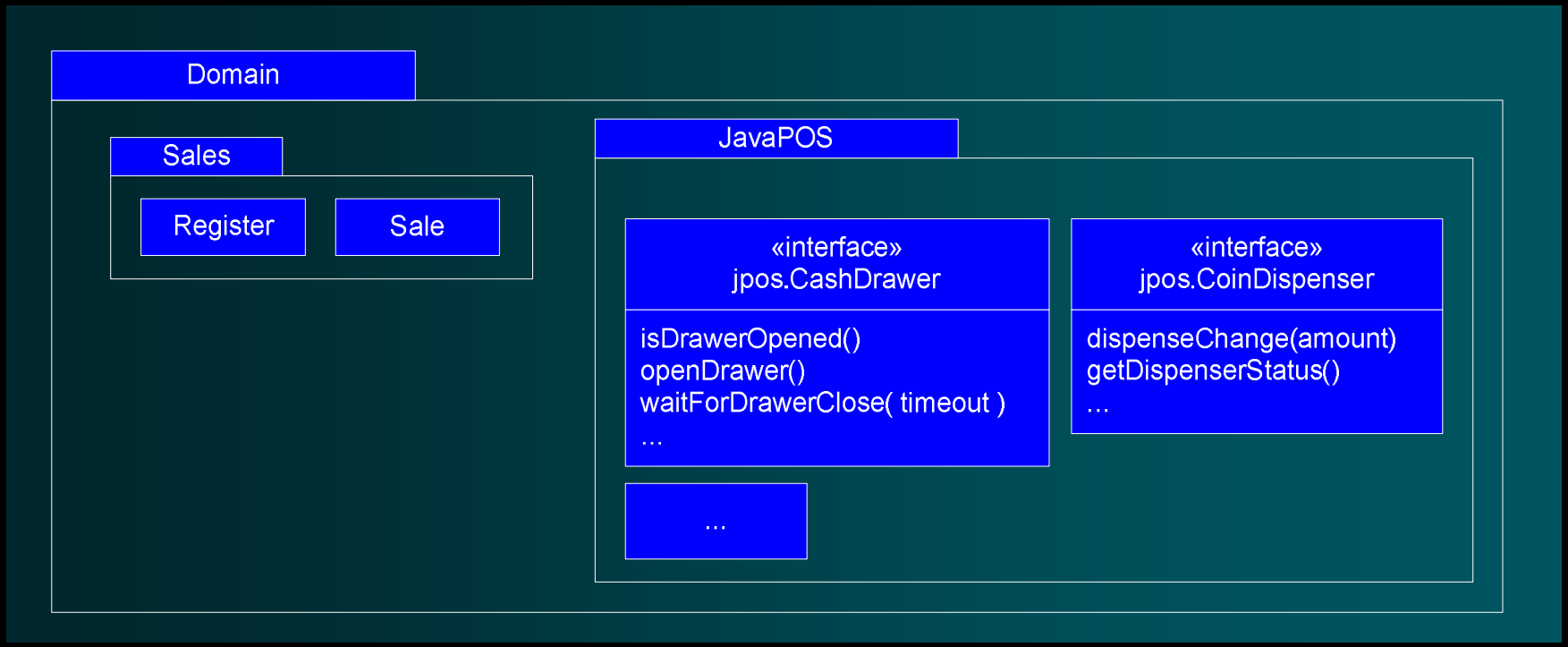
- A redirection proxy is used in NextGen as follows:
  - Send a postSale message to the redirection proxy.
  - If the redirection proxy fails to make contact with the external service, then it redirects the postSale message to a local service.


"accounting" actually references an instance of Accounting-RedirectionProxy
{
... payment work
if (payment completed)
completeSaleHandling()
}
1
Register
accounting : IAccountingAdapter
+ makePayment()
- completeSaleHandling()
«interface»
IAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
1
2
{
...
accounting.postSale(currentSale)
...
}
2
AccountingRedirectionProxy
externalAccounting : IAccountingAdapter
localAccounting : IAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
SAPAccountingAdapter
postReceivable(ReceivableEntry)
postSale(Sale)
...
{
externalAccounting.postSale(sale)
if (externalAccounting fails)
localAccounting.postSale(sale)
}
3
LocalAccounting
postReceivable(ReceivableEntry)
postSale(Sale)
...
{
save the sale in a local file (to be forwarded to external accounting later)
}
4

36.6. Accessing External Physical Devices with Adapters

- How to interact with physical devices that comprise a POS terminal, such as
  - opening a cash drawer,
  - dispensing change from the coin dispenser,
  - capturing a signature from the digital signature device.
- UnifiedPOS
  - defines standard object-oriented interfaces for all common POS devices.
- JavaPOS
  - a Java mapping of the UnifiedPOS.

36.6. Accessing External Physical Devices with Adapters



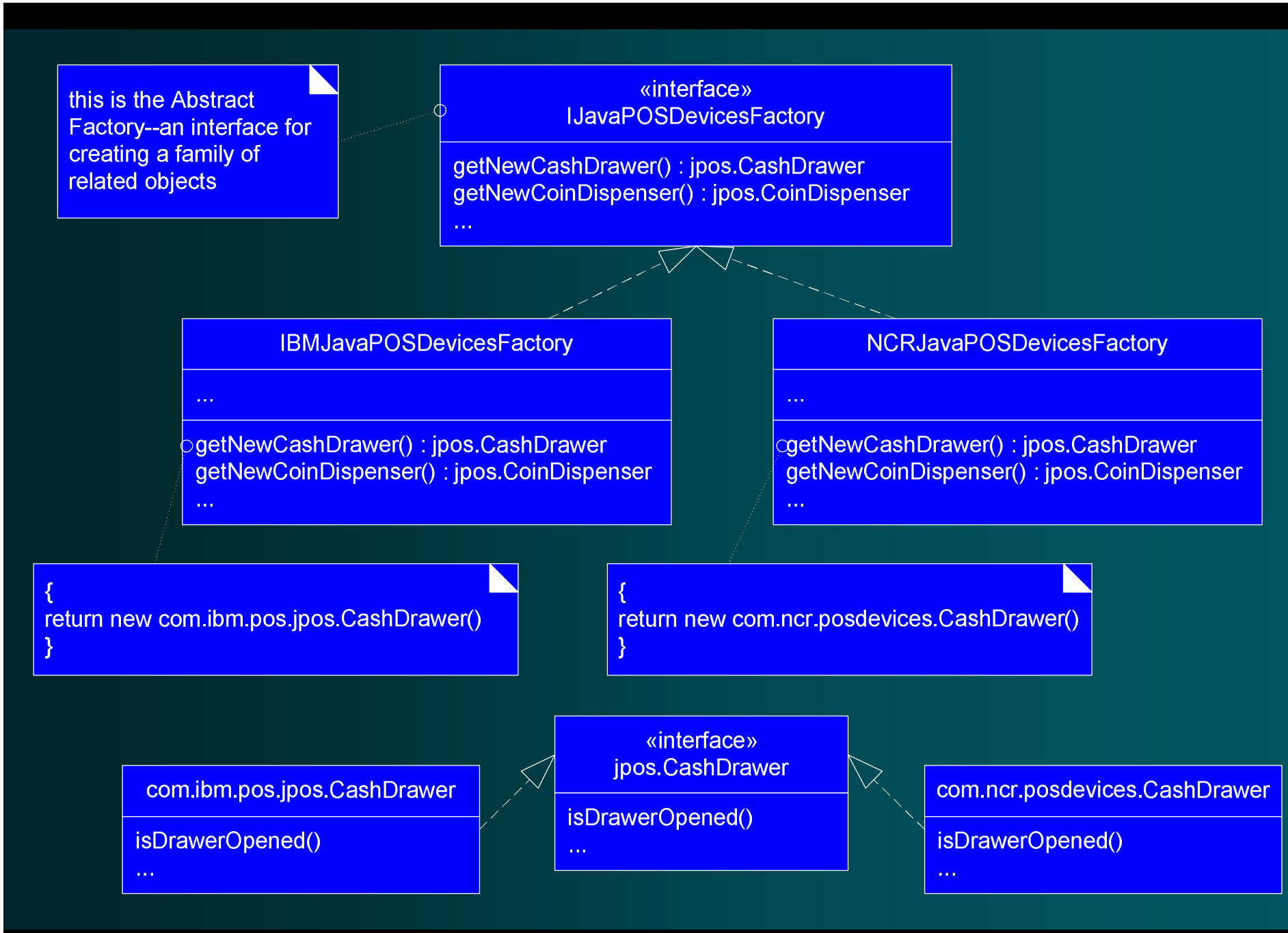
36.7. Abstract Factory (GoF) for Families of Related Objects

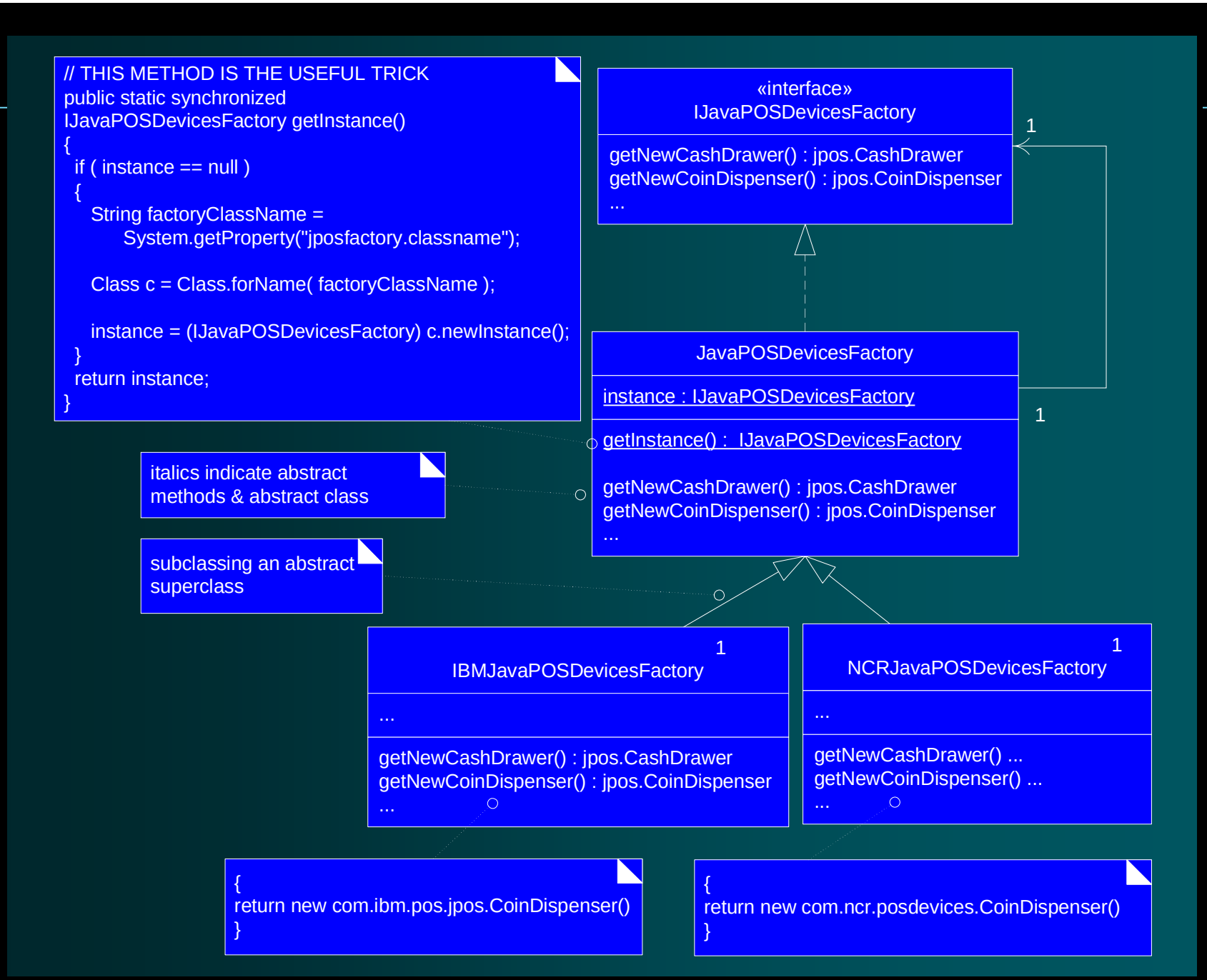
- How to design the NextGen POS application to use the IBM Java drivers if IBM hardware is used, NCR drivers if appropriate, and so forth?
  - families of classes that need to be created, and each family implements the same interfaces.

36.7. Abstract Factory (GoF) for Families of Related Objects

w Abstract Factory

- § Problem
  - How to create families of related classes that implement a common interface?
- § Solution
  - Define a factory interface. Define a concrete factory class for each family of things to create.

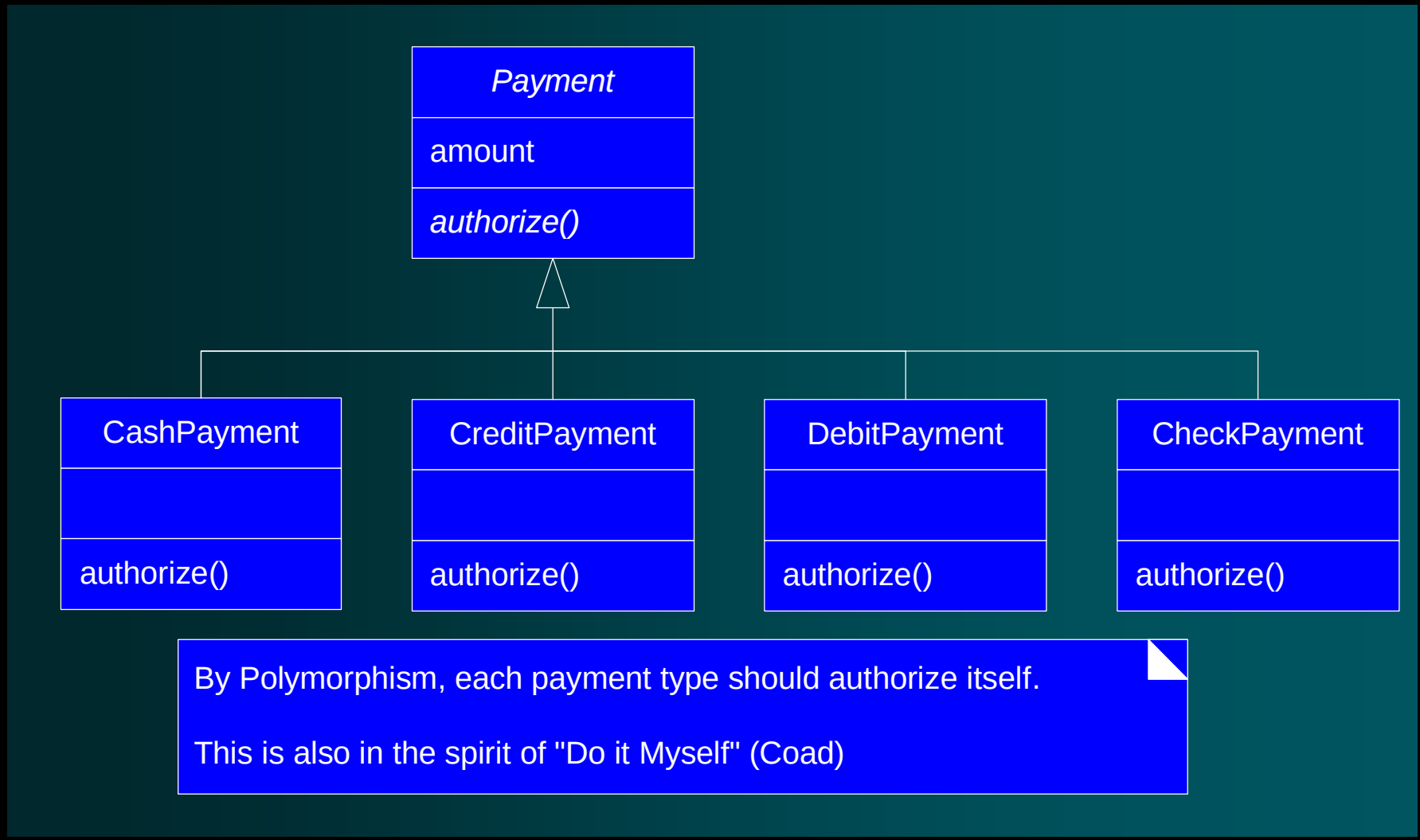

this is the Abstract Factory--an interface for creating a family of related objects
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
IBMJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
NCRJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
{
return new com.ibm.pos.jpos.CashDrawer()
}
{
return new com.ncr.posdevices.CashDrawer()
}
«interface»
jpos.CashDrawer
isDrawerOpened()
...
com.ibm.pos.jpos.CashDrawer
isDrawerOpened()
...
com.ncr.posdevices.CashDrawer
isDrawerOpened()
...


// THIS METHOD IS THE USEFUL TRICK
public static synchronized
IJavaPOSDevicesFactory getInstance()
{
if (instance == null)
{
String factoryClassName =
System.getProperty("jposfactory.classname");
Class c = Class.forName(factoryClassName);
instance = (IJavaPOSDevicesFactory) c.newInstance();
}
return instance;
}
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
1
JavaPOSDevicesFactory
instance : IJavaPOSDevicesFactory
getInstance() : IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
1
italics indicate abstract
methods & abstract class
subclassing an abstract
superclass
IBMJavaPOSDevicesFactory
1
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
NCRJavaPOSDevicesFactory
1
...
getNewCashDrawer() ...
getNewCoinDispenser() ...
...
{
return new com.ibm.pos.jpos.CoinDispenser()
}
{
return new com.ncr.posdevices.CoinDispenser()
}

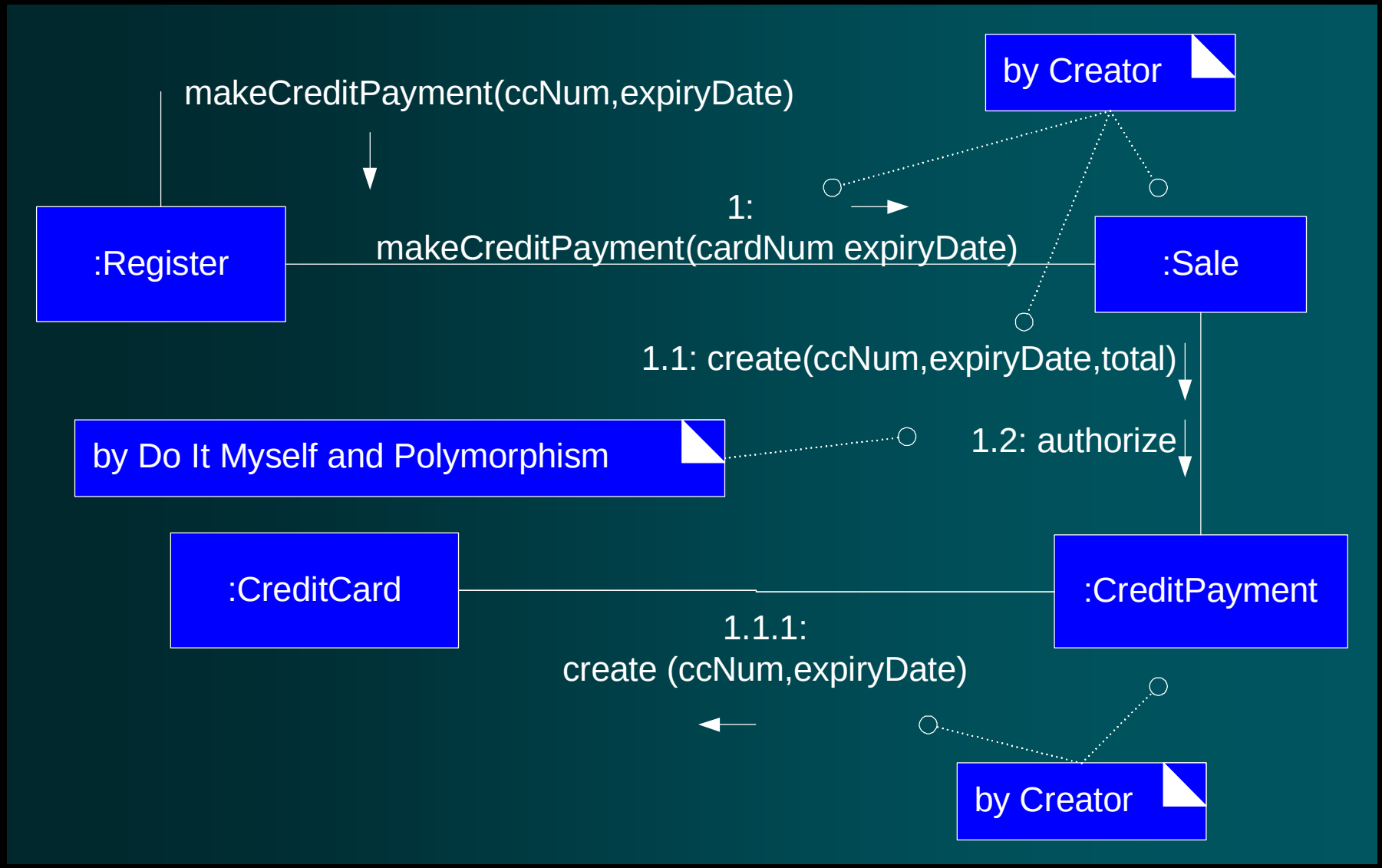
36.8. Handling Payments with Polymorphism and Do It Myself

- Do It Myself and Information Expert usually lead to the same choice.
  - Circle objects draw themselves,
  - Square objects draw themselves,
  - Text objects spell-check themselves
- Do It Myself and Polymorphism usually lead to the same choice.
  - When related alternatives vary by type, assign responsibility using polymorphic operations to the types for which the behavior varies.

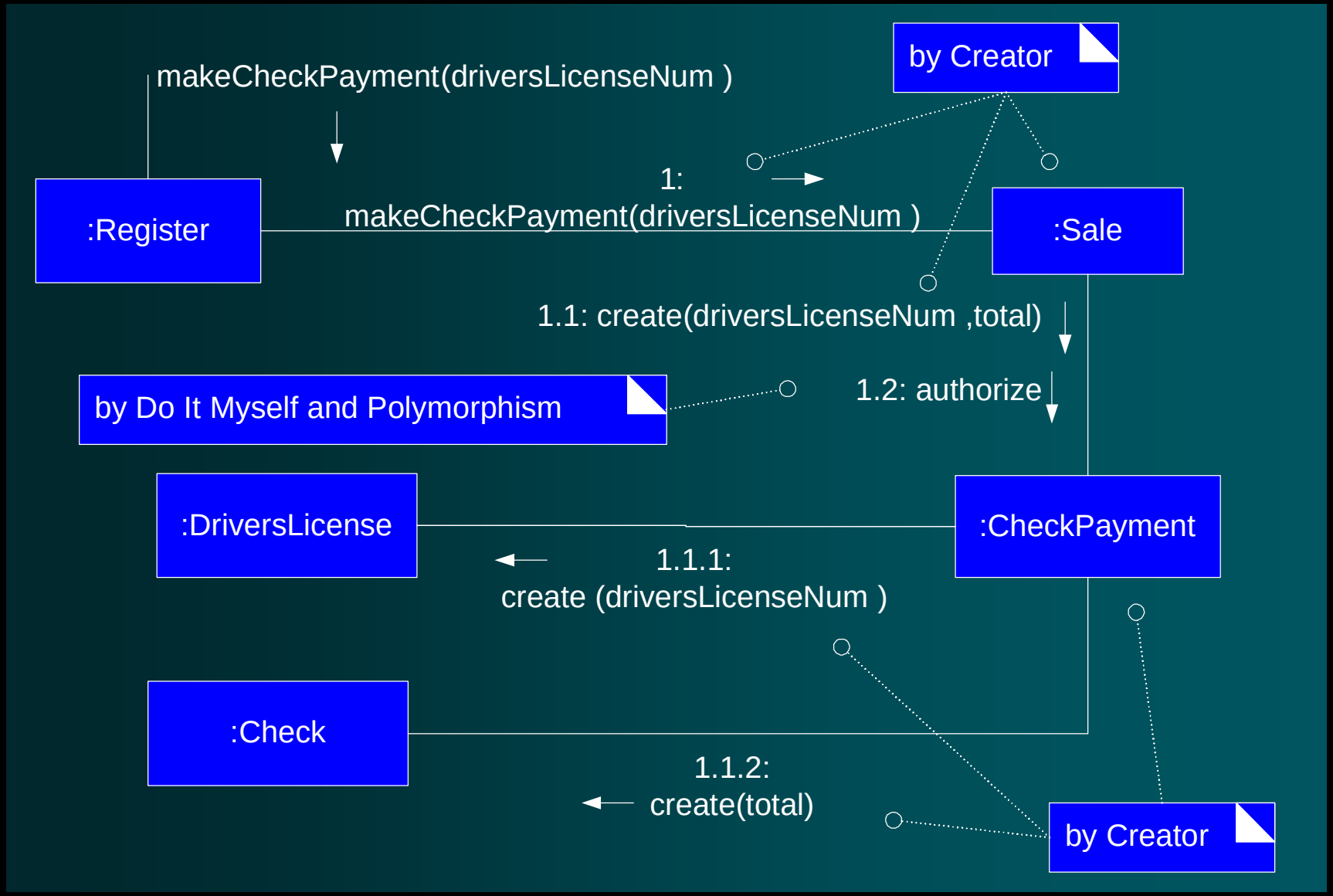
36.8. Handling Payments with Polymorphism and Do It Myself



36.8. Handling Payments with Polymorphism and Do It Myself

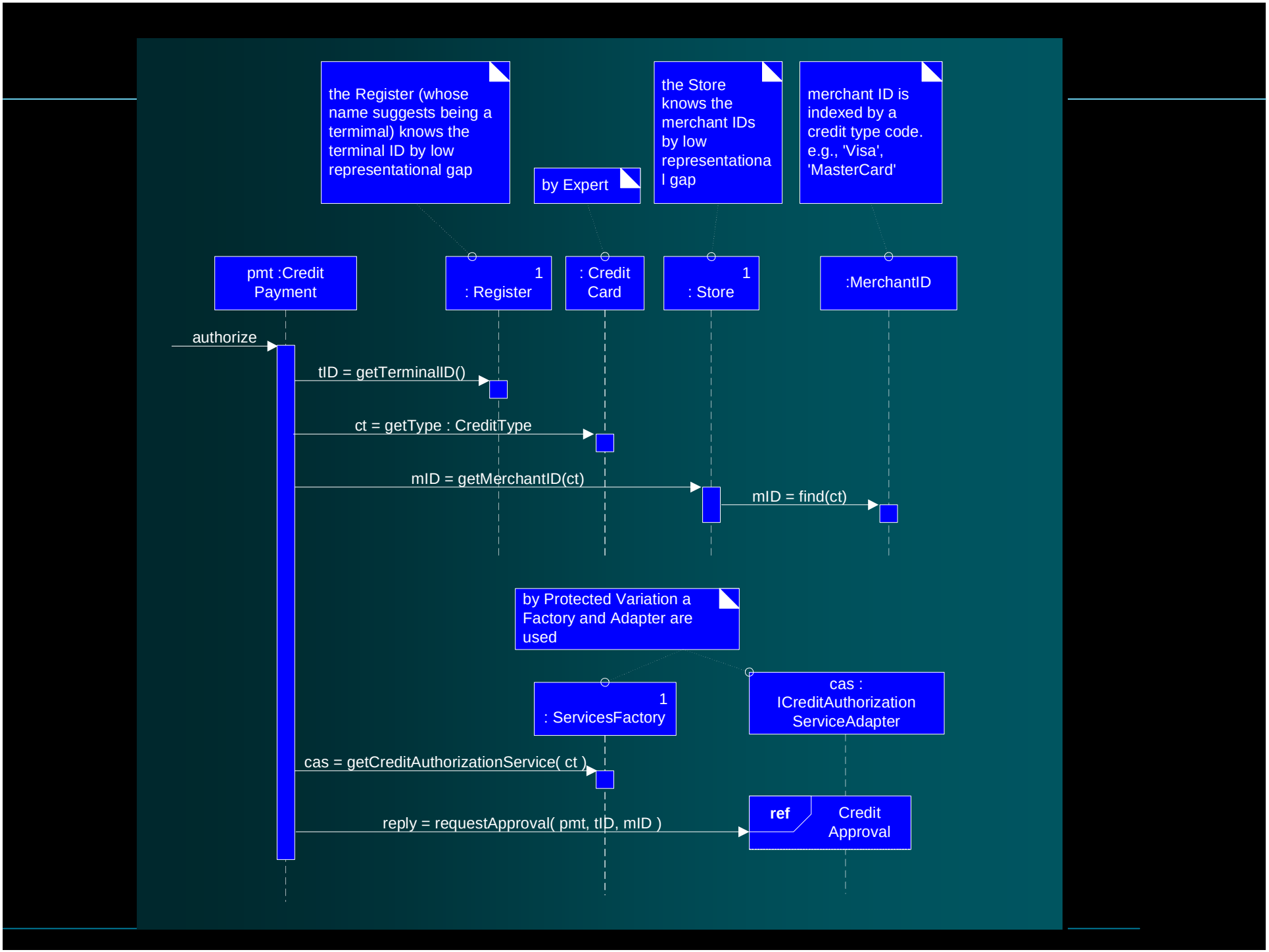


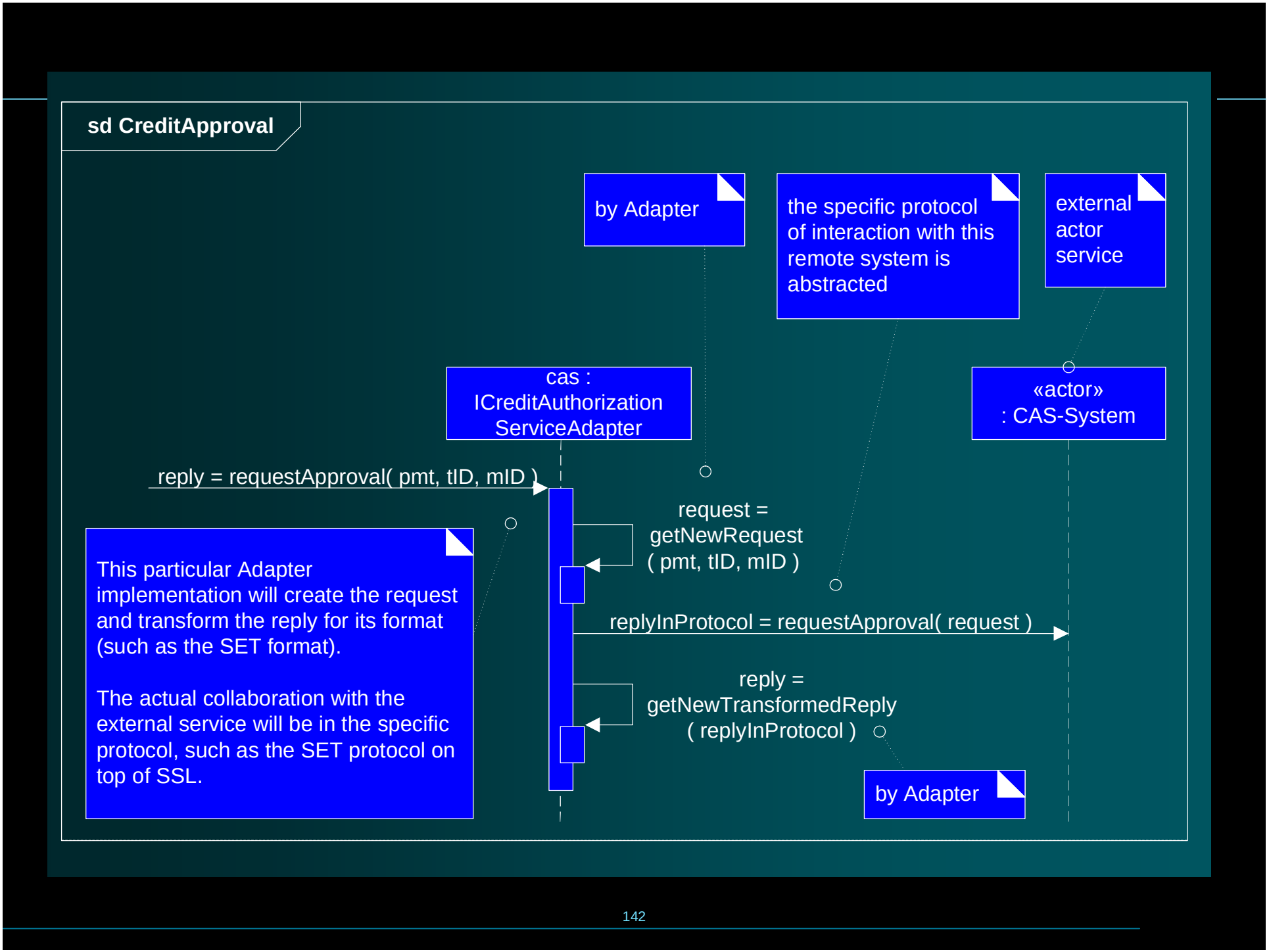
36.8. Handling Payments with Polymorphism and Do It Myself



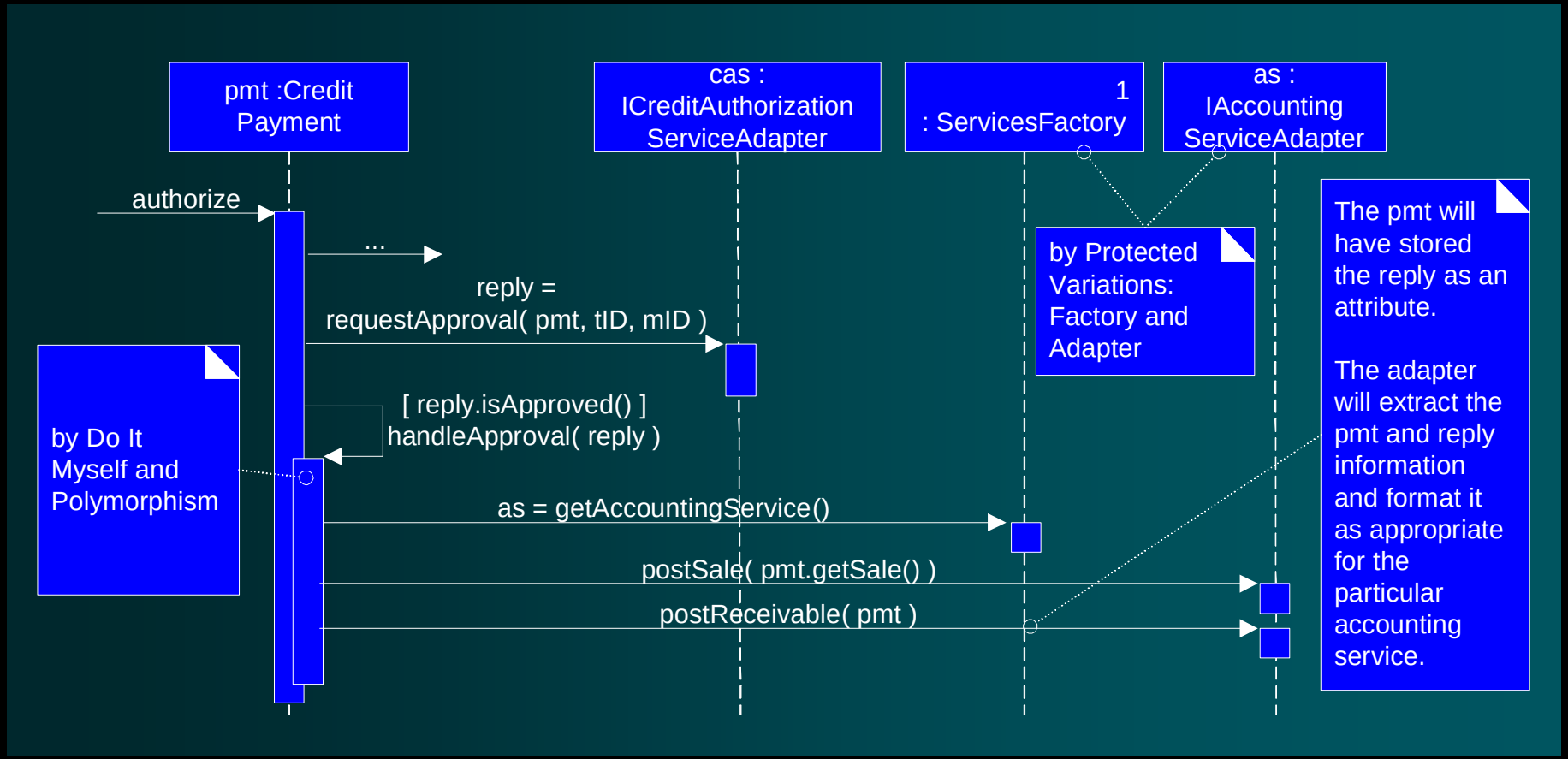
36.8. Handling Payments with Polymorphism and Do It Myself

- **Relevant Credit Payment Domain Information**
  - POS systems are physically connected with external authorization services in several ways,
    - including phone lines and always-on broadband Internet connections.
  - Different application-level protocols and associated data formats are used,
    - such as Secure Electronic Transaction (SET). XMLPay.
  - Payment authorization can be viewed as a regular synchronous operation;
  - All payment authorization protocols involve sending identifiers uniquely identifying the store, and the POS terminal. A reply includes an approval or denial code, and a unique transaction ID.
  - A store may use different external authorization services for different credit card types.
  - The credit company type can be deduced from the card number.
  - The adapter implementations will protect the upper layers of the system against all these variations in payment authorization.

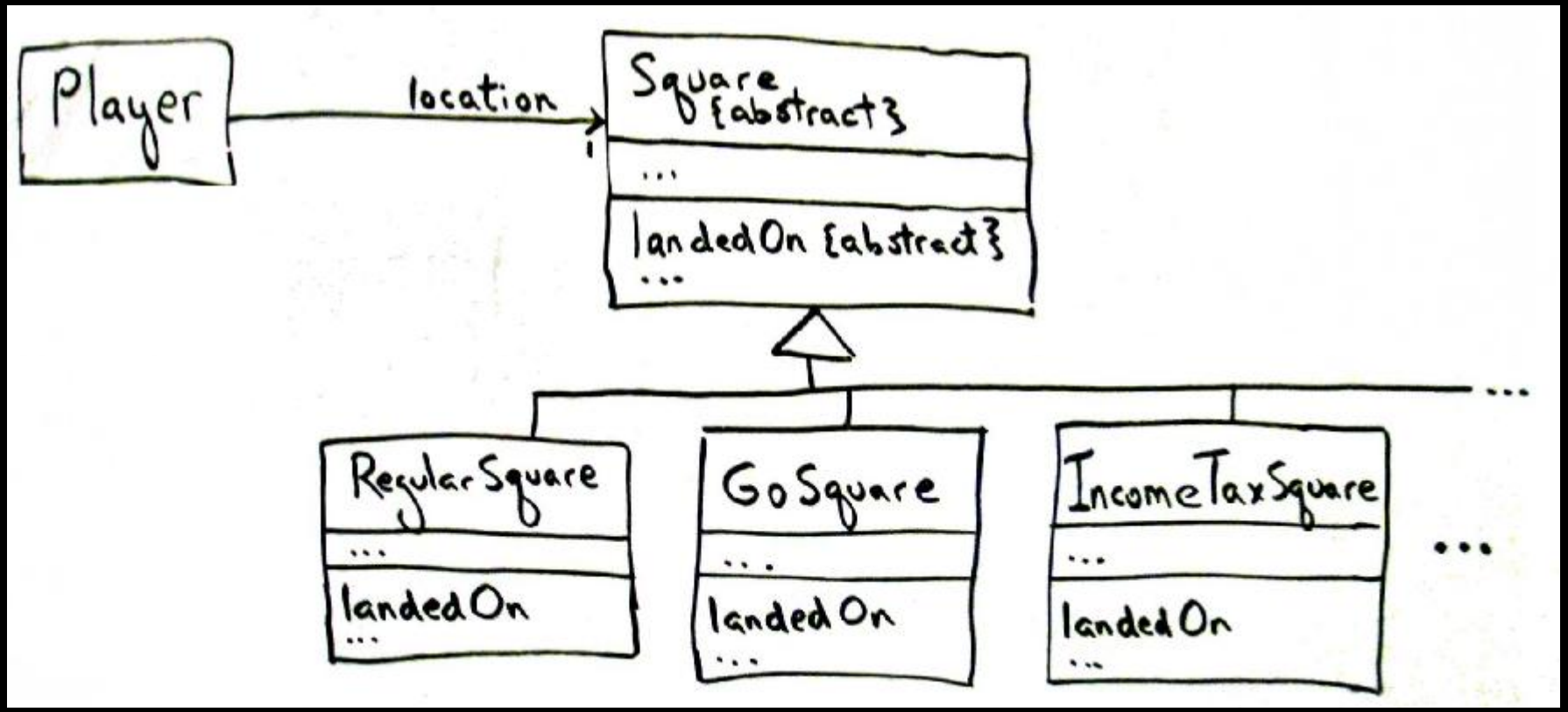

the Register (whose name suggests being a termimal) knows the terminal ID by low representational gap
by Expert
the Store knows the merchant IDs by low representational gap
merchant ID is indexed by a credit type code. e.g., 'Visa', 'MasterCard'
pmt :Credit Payment
1
: Register
: Credit Card
1
: Store
:MerchantID
authorize
tID = getTerminalID()
ct = getType : CreditType
mID = getMerchantID(ct)
mID = find(ct)
by Protected Variation a Factory and Adapter are used
1
: ServicesFactory
cas : ICreditAuthorization ServiceAdapter
cas = getCreditAuthorizationService(ct)
reply = requestApproval(pmt, tID, mID)
ref
Credit Approval


sd CreditApproval
by Adapter
the specific protocol of interaction with this remote system is abstracted
external actor service
cas : ICreditAuthorization ServiceAdapter
«actor» : CAS-System
reply = requestApproval(pmt, tID, mID)
request = getNewRequest (pmt, tID, mID)
This particular Adapter implementation will create the request and transform the reply for its format (such as the SET format).
The actual collaboration with the external service will be in the specific protocol, such as the SET protocol on top of SSL.
replyInProtocol = requestApproval(request)
reply = getNewTransformedReply (replyInProtocol)
by Adapter

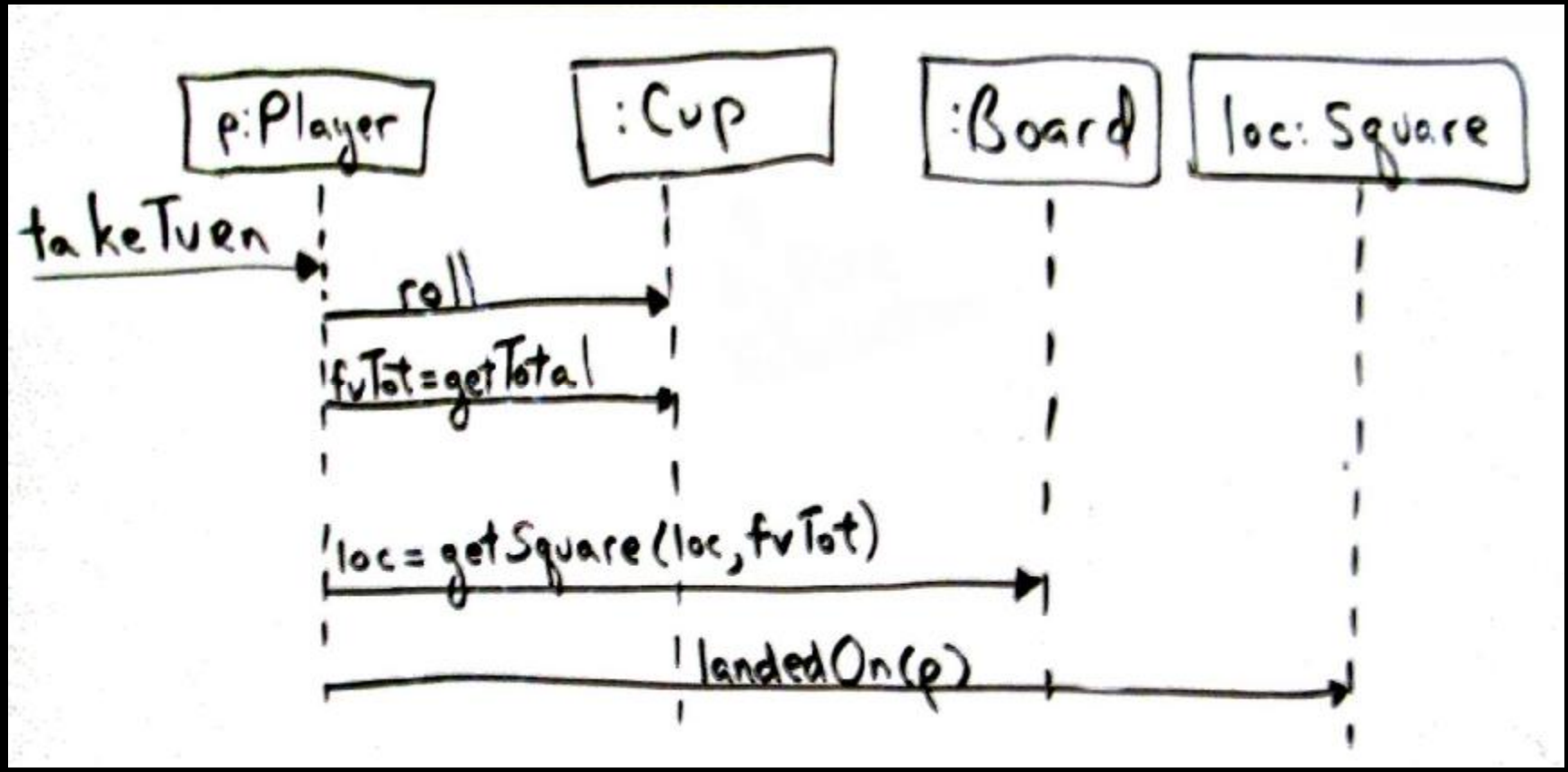
36.8. Handling Payments with Polymorphism and Do It Myself



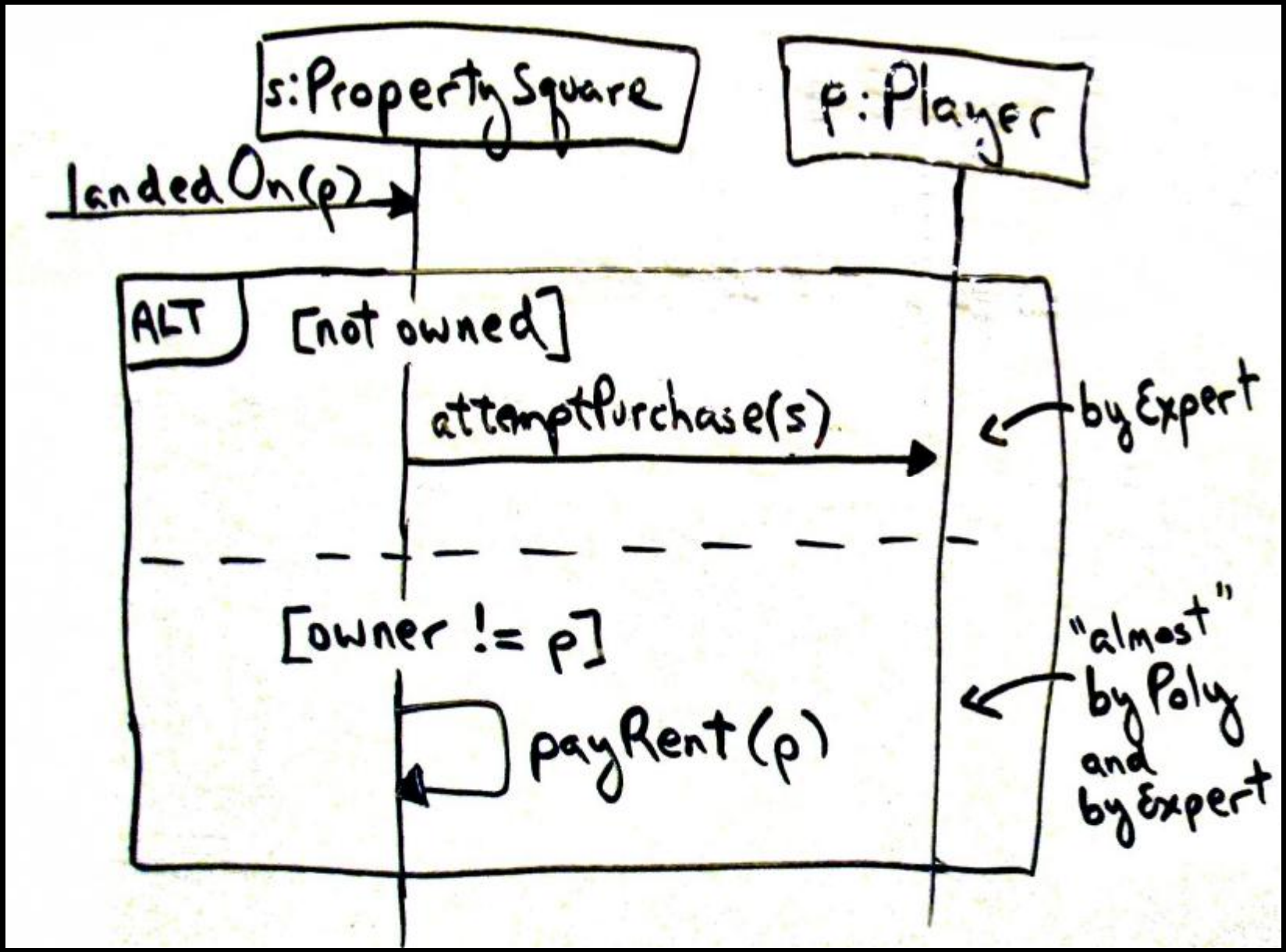
36.9. Example: Monopoly



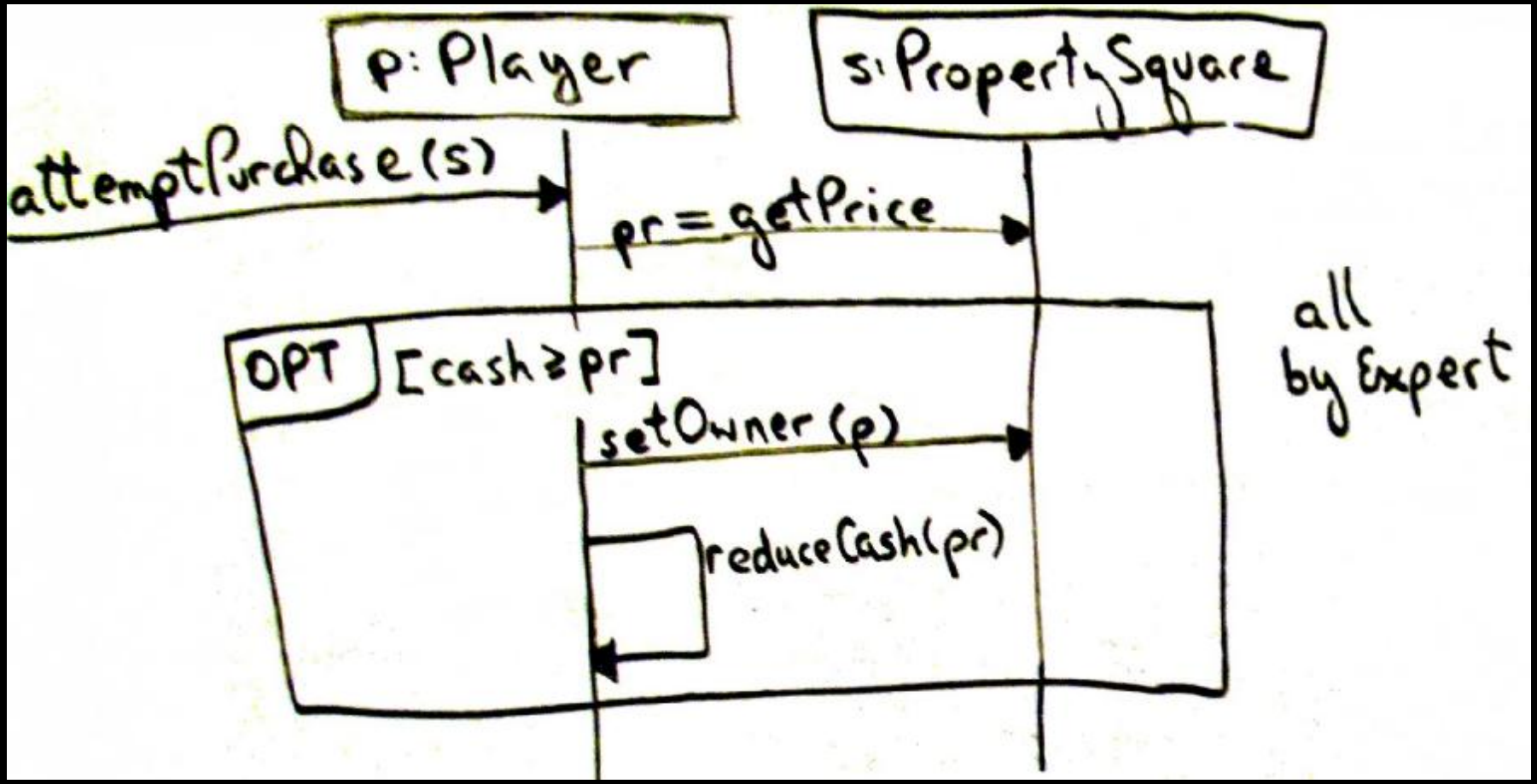
36.9. Example: Monopoly



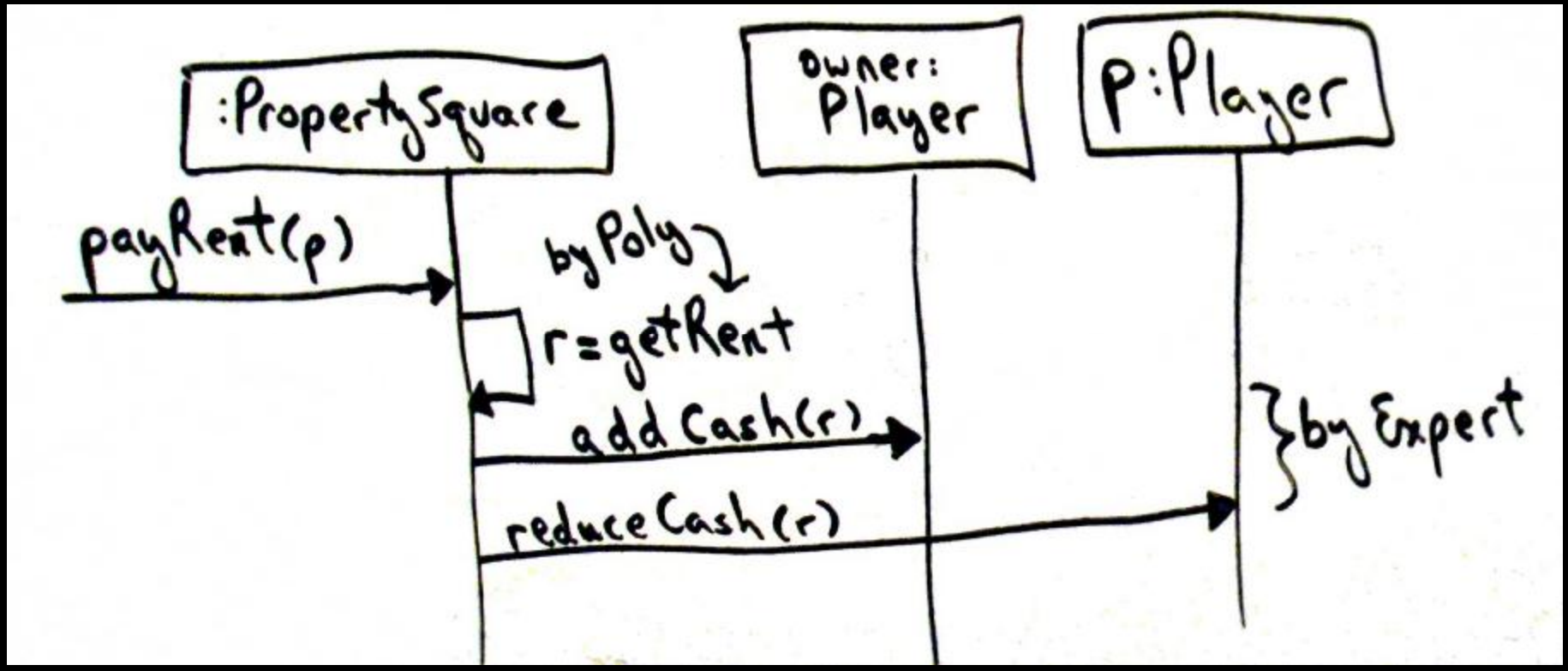
36.9. Example: Monopoly

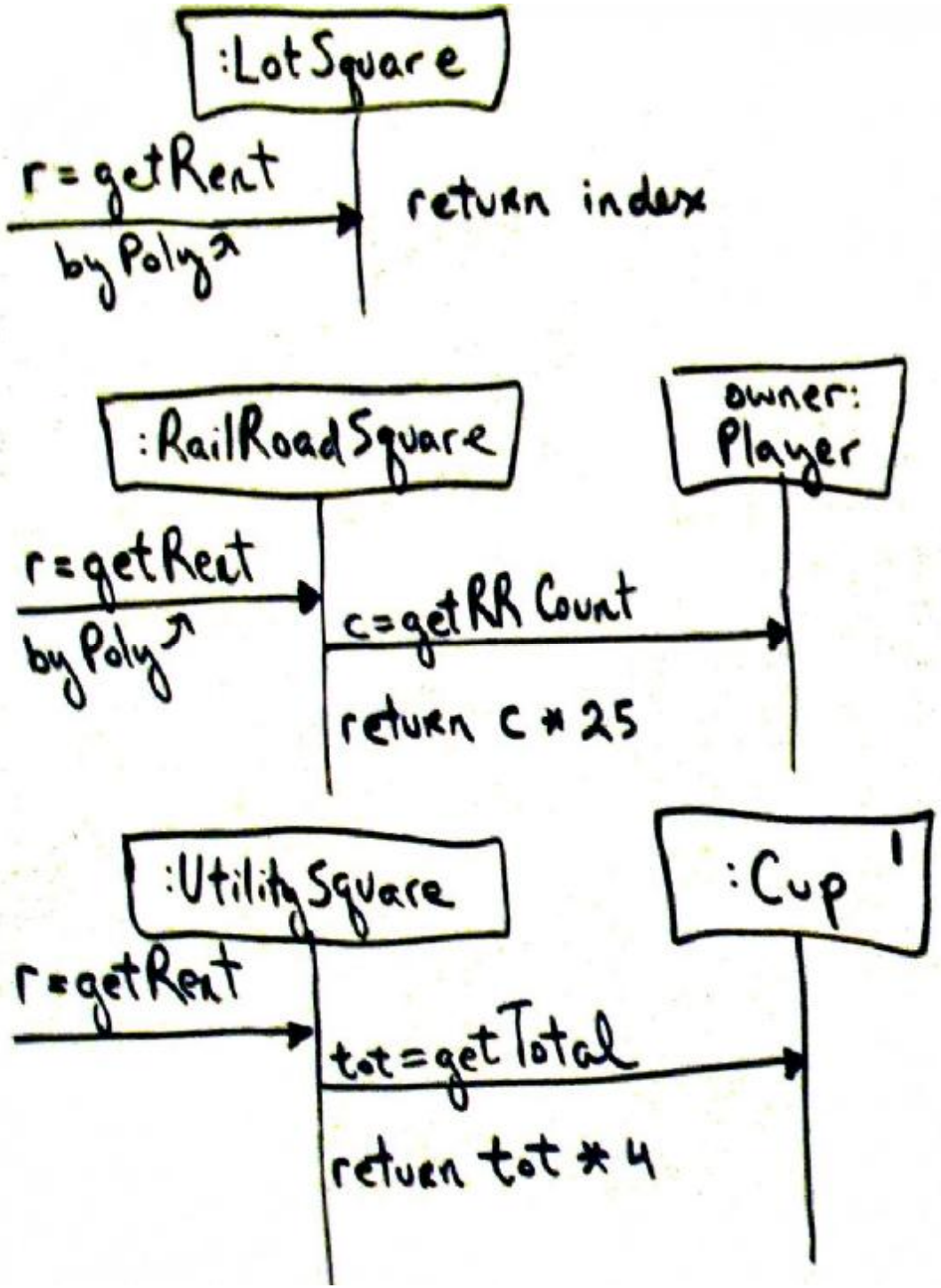


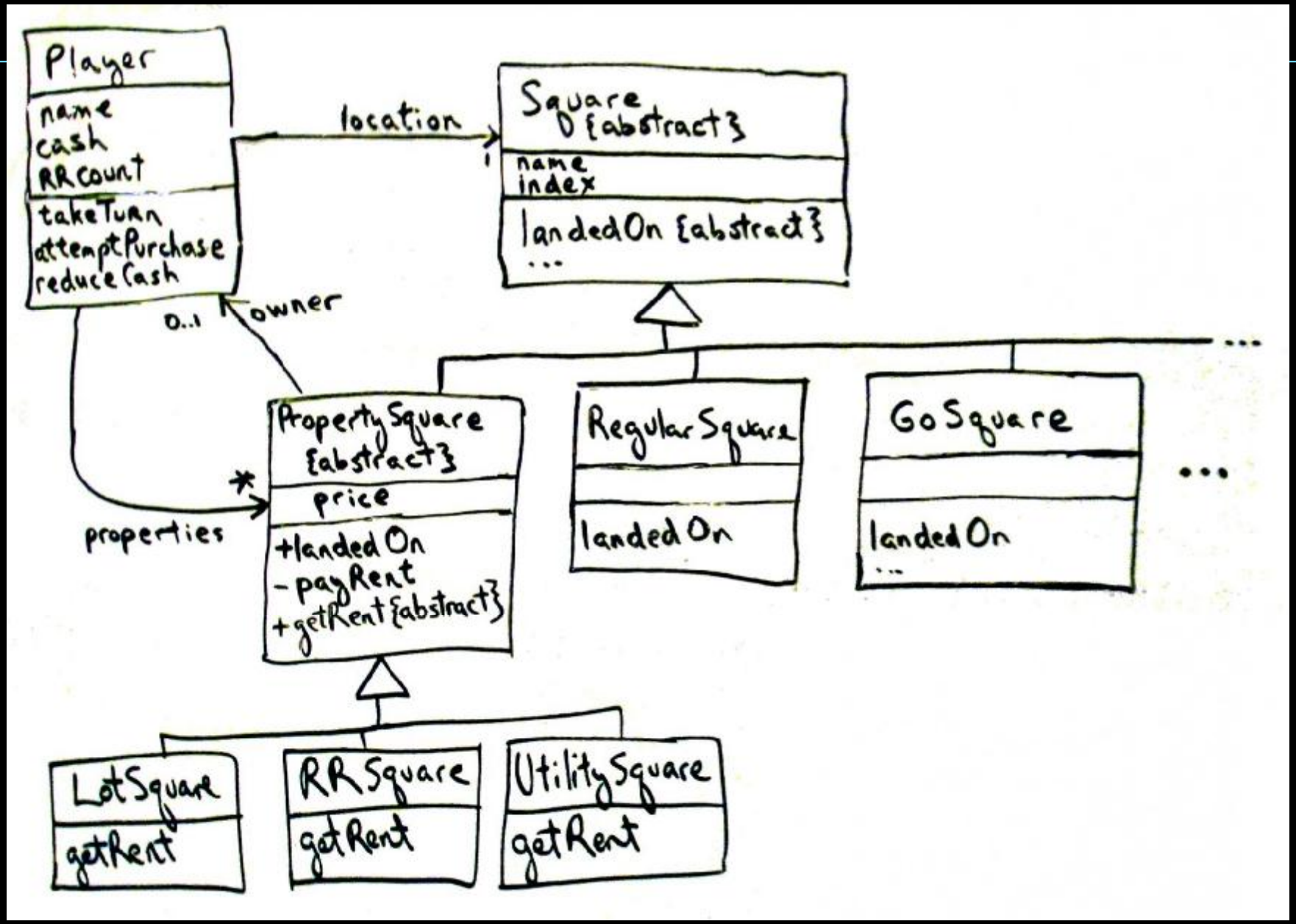
36.9. Example: Monopoly



36.9. Example: Monopoly




:LotSquare
r=getRent
by Poly
return index
:RailRoadSquare
owner:
Player
r=getRent
by Poly
c=getRR Count
return c * 25
:UtilitySquare
:Cup
r=getRent
tot=getTotal
return tot * 4


Player
name
cash
RRcount
takeTurn
attemptPurchase
reduceCash
location
1
Square
{abstract}
name
index
landedOn {abstract}
...
0..1
owner
properties
*
PropertySquare
{abstract}
price
+landedOn
-payRent
+getRent {abstract}
RegularSquare
landedOn
GoSquare
landedOn
...
LotSquare
getRent
RRSquare
getRent
UtilitySquare
getRent

Chapter 37: Designing a Persistence Framework with Patterns

Objective

- Design part of a framework with the Template Method, State, and Command patterns.
- Introduce issues in object-relational (O-R) mapping.
- Implement lazy materialization with Virtual Proxies.

37.1. The Problem: Persistent Objects

- Storage Mechanisms and Persistent Objects
  - Object databases
  - Relational databases
    - mismatch between record-oriented and object-oriented representations of data;
  - Other In addition to RDBs
    - files, XML structures, Palm OS PDB files, hierarchical databases

37.2. The Solution: A Persistence Service from a Persistence Framework

- A persistence framework
  - a general-purpose, reusable, and extendable set of types that provides functionality to support persistent objects.
- A persistence service
  - actually provides the service, and will be created with a persistence framework.

37.3. Frameworks

- Framework
  - An extendable set of objects for related functions,
  - Provides an implementation for the core and unvarying functions,
  - Includes a mechanism to allow a developer to plug in the varying functions, or to extend the functions.
  - Relies on the Hollywood Principle
- "Don't call us, we'll call you."
  - Offer a high degree of reuse

37.4. Requirements for the Persistence Service and Framework

- Framework PFW (Persistence Framework).
  - Store and retrieve objects in a persistent storage mechanism
  - Commit and rollback transactions
  - Be extendable to support different storage mechanisms and formats, such as RDBs, records in flat files, or XML in files.

37.5. Key Ideas

- Mapping
  - There must be some mapping
    - between a class and its persistent store
    - between object attributes and the fields in a record.
- Object identity
  - easily relate records to objects,
  - ensure there are no inappropriate duplicates, records and objects have a unique object identifier.

37.5. Key Ideas

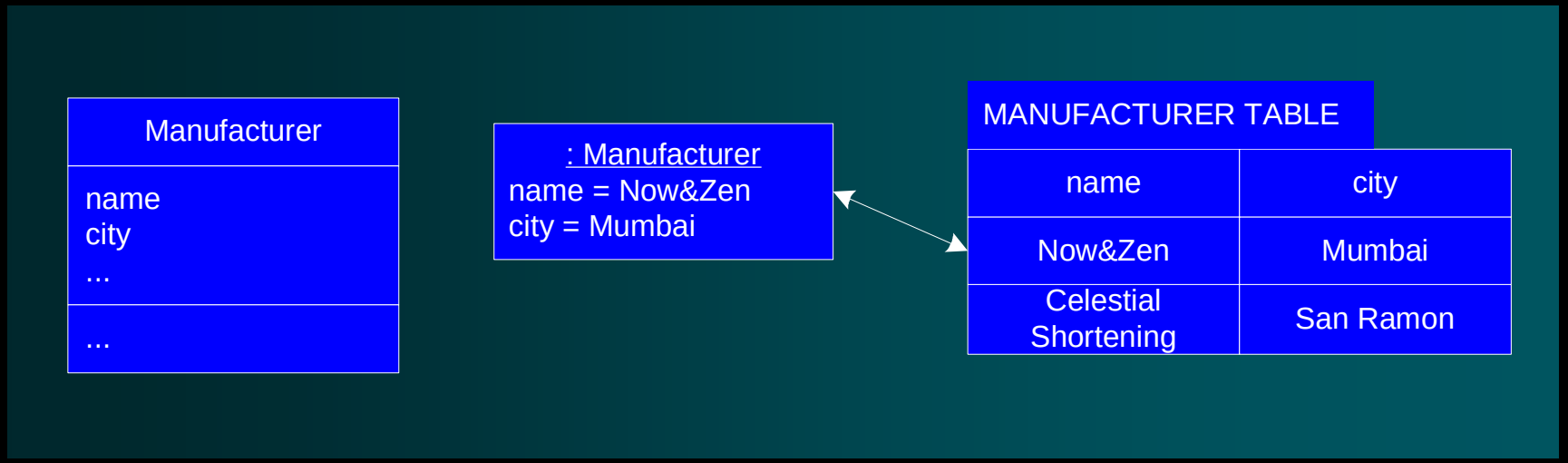
- Database mapper
  - A Pure Fabrication database mapper is responsible for materialization and dematerialization.
- Materialization and dematerialization
  - Materialization is the act of transforming a non-object representation of data from a persistent store into objects.
  - Dematerialization is the opposite activity.
- Caches
  - Persistence services cache materialized objects for performance.

37.5. Key Ideas

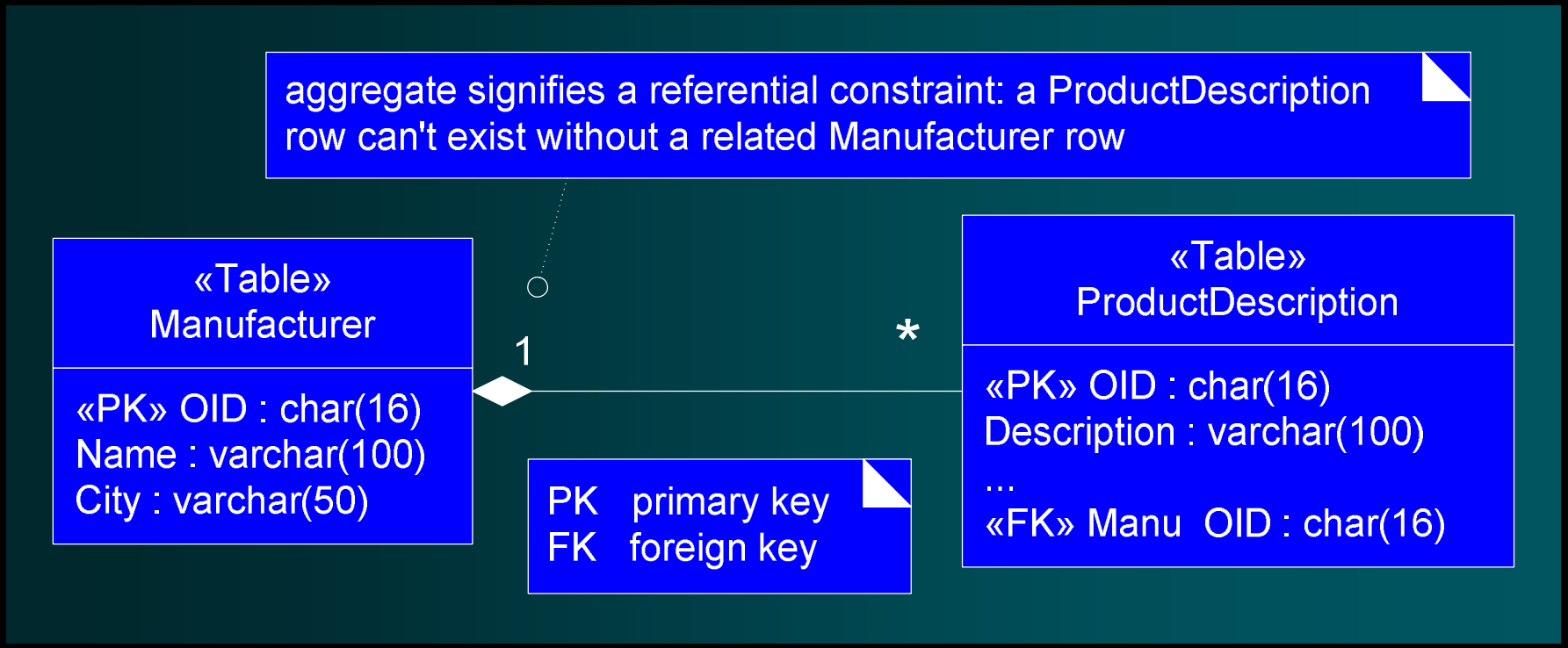
- Transaction state of object
  - It is useful to know the state of objects in terms of their relationship to the current transaction.
- Transaction operations
  - Commit and rollback operations.
- Lazy materialization
  - Not all objects are materialized at once; a particular instance is only materialized on-demand.
- Virtual proxies
  - Lazy materialization can be implemented using a smart reference known as a virtual proxy.

37.6. Pattern: Representing Objects as Tables

- Representing Objects as Tables pattern

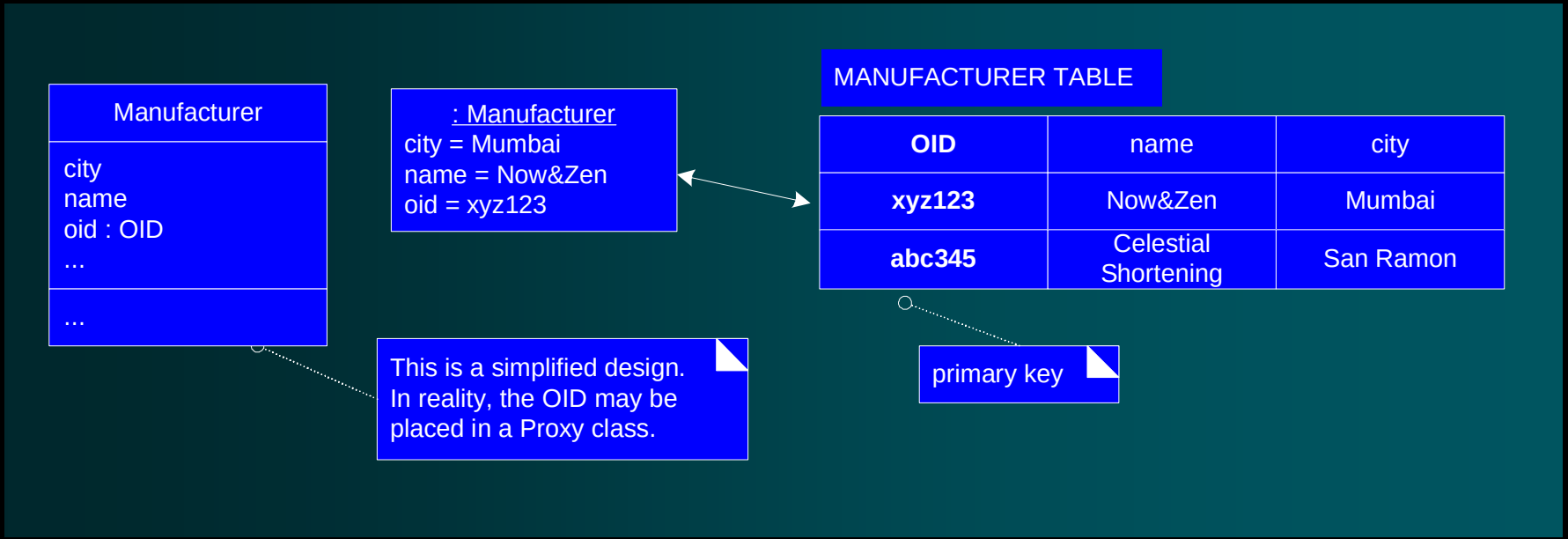


37.7. UML Data Modeling Profile



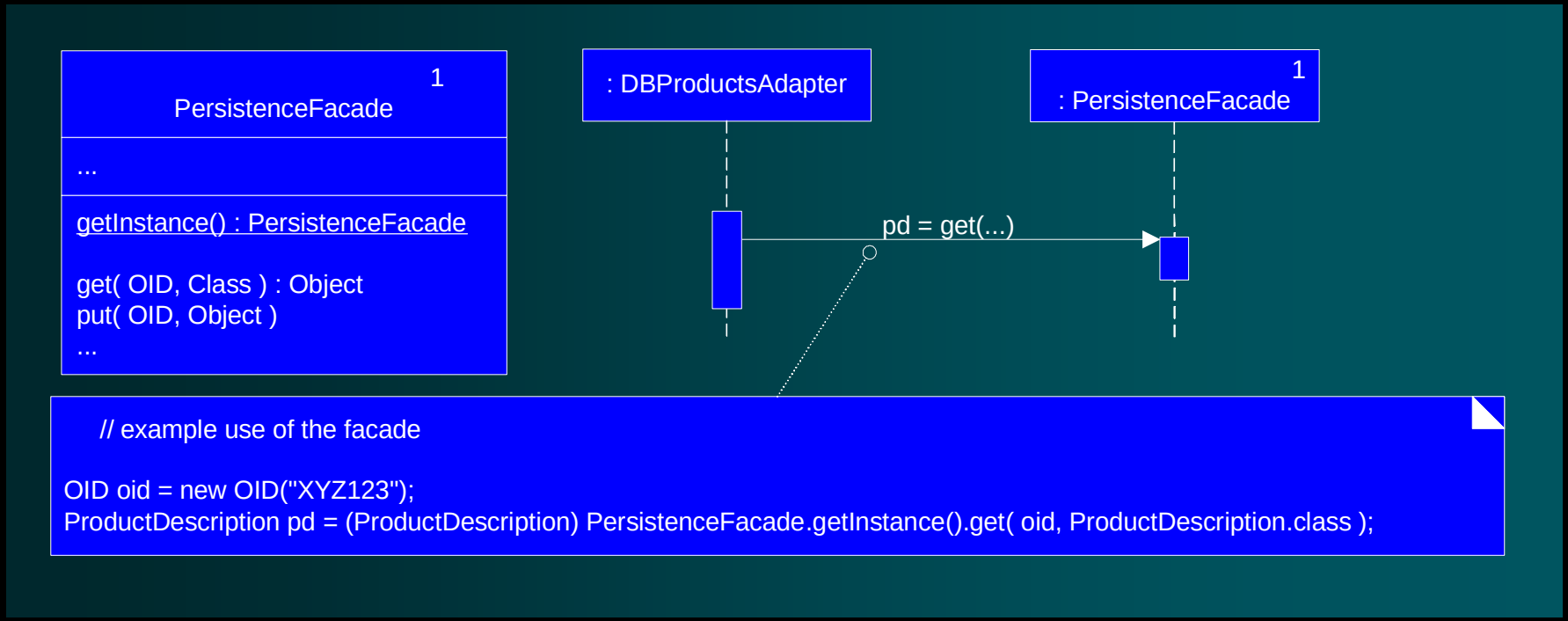
37.8. Pattern: Object Identifier

- Object Identifier pattern
  - assigning an object identifier (OID) to each record and object.



37.9. Accessing a Persistence Service with a Facade

- Define a facade for its services



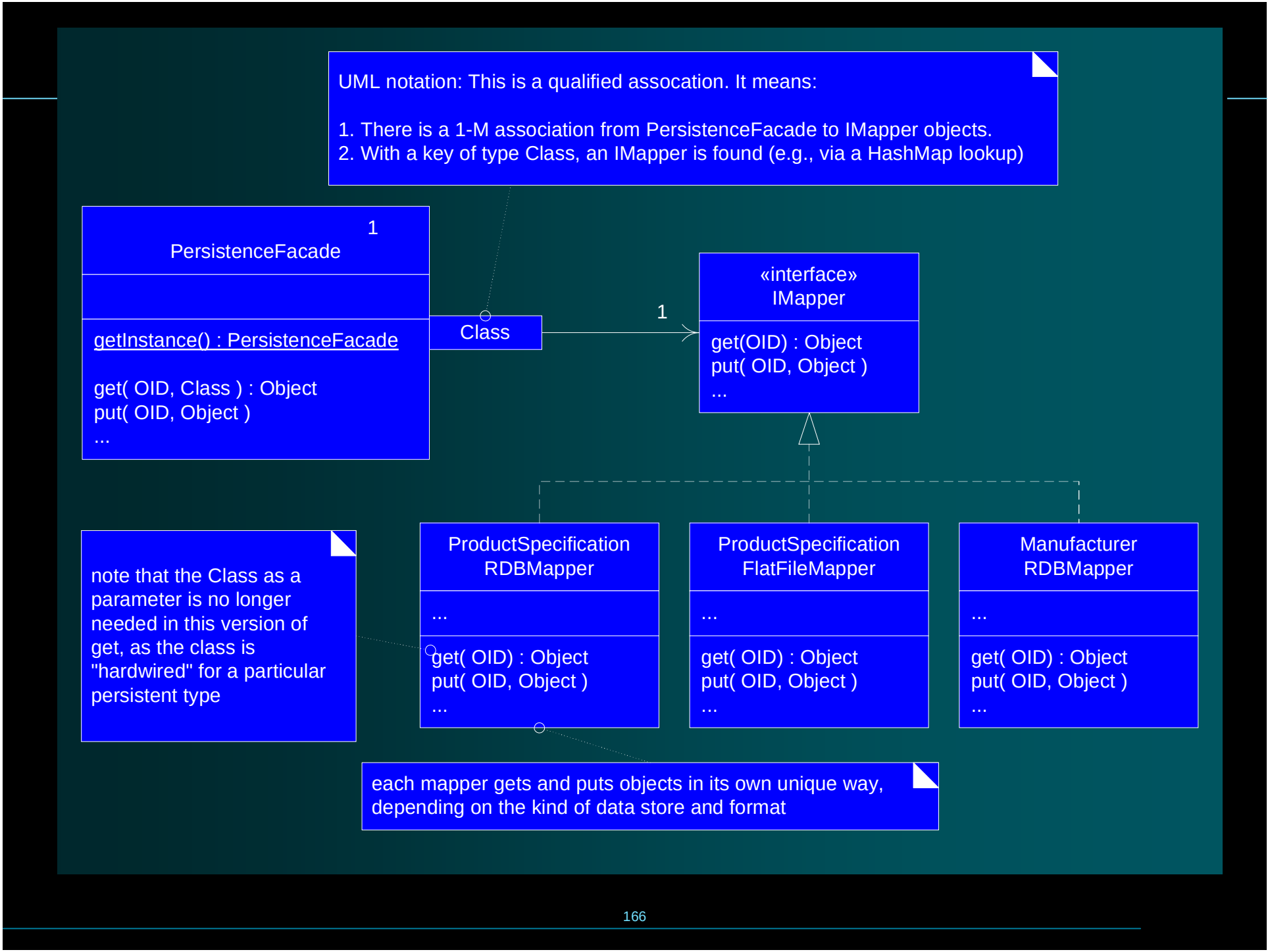
Who should be responsible for materialization and dematerialization of objects from a persistent store?

37.10. Mapping Objects: Database Mapper or Database Broker Pattern

- Direct Mapping
  - A persistent object class defines the code to save itself in a database
  - violation of Low Coupling.
    - Strong coupling of the persistent object class to persistent storage knowledge
  - violation of High Cohesion
    - Technical service concerns are mixing with application logic concerns.

37.10. Mapping Objects: Database Mapper or Database Broker Pattern

- Indirect mapping
  - Uses other objects to do the mapping for persistent objects.
  - Database Broker pattern/Database Mapper pattern
    - make a class that is responsible for materialization, dematerialization, and object caching.


UML notation: This is a qualified assocation. It means:
1. There is a 1-M association from PersistenceFacade to IMapper objects.
2. With a key of type Class, an IMapper is found (e.g., via a HashMap lookup)
PersistenceFacade
1
getInstance() : PersistenceFacade
get(OID, Class) : Object
put(OID, Object)
...
Class
1
«interface»
IMapper
get(OID) : Object
put(OID, Object)
...
note that the Class as a parameter is no longer needed in this version of get, as the class is "hardwired" for a particular persistent type
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

37.10. Mapping Objects: Database Mapper or Database Broker Pattern

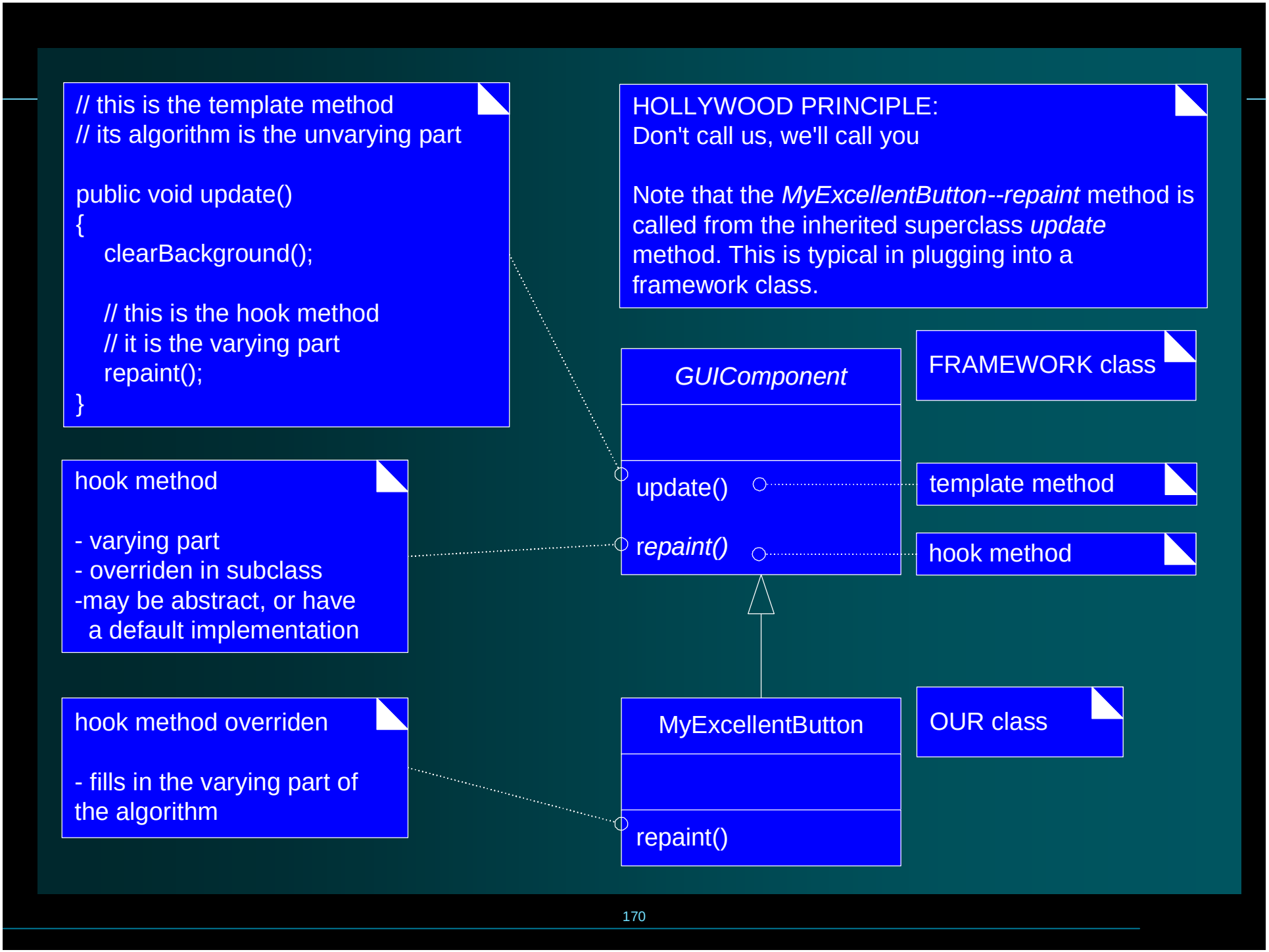
```
class PersistenceFacade {
        //...
         public Object get( OID oid,
                Class persistenceClass ) {

                // an IMapper is keyed by the Class
                // of the persistent object
                IMapper mapper = (IMapper)
                        mappers.get( persistenceClass );
                // delegate
                return mapper.get( oid );
        }
        //...
}
```

37.10. Mapping Objects: Database Mapper or Database Broker Pattern

- Metadata-Based Mappers
  - Dynamically generate the mapping from an object schema to another schema based on reading in metadata
    - Such as
      - TableX maps to Class Y;
      - column Z maps to object property P
  - This approach is feasible for languages with reflective programming capabilities, such as Java, C#, or Smalltalk

37.11. Framework Design with the Template Method Pattern

- Template Method GoF design pattern
  - Define a method (the Template Method) in a superclass that defines the skeleton of an algorithm,
    - with its varying and unvarying parts.


// this is the template method
// its algorithm is the unvarying part
public void update()
{
clearBackground();
// this is the hook method
// it is the varying part
repaint();
}
HOLLYWOOD PRINCIPLE:
Don't call us, we'll call you
Note that the MyExcellentButton--repaint method is called from the inherited superclass update method. This is typical in plugging into a framework class.
GUIComponent
update()
repaint()
FRAMEWORK class
template method
hook method
hook method
- varying part
- overriden in subclass
-may be abstract, or have
a default implementation
hook method overriden
- fills in the varying part of
the algorithm
MyExcellentButton
repaint()
OUR class

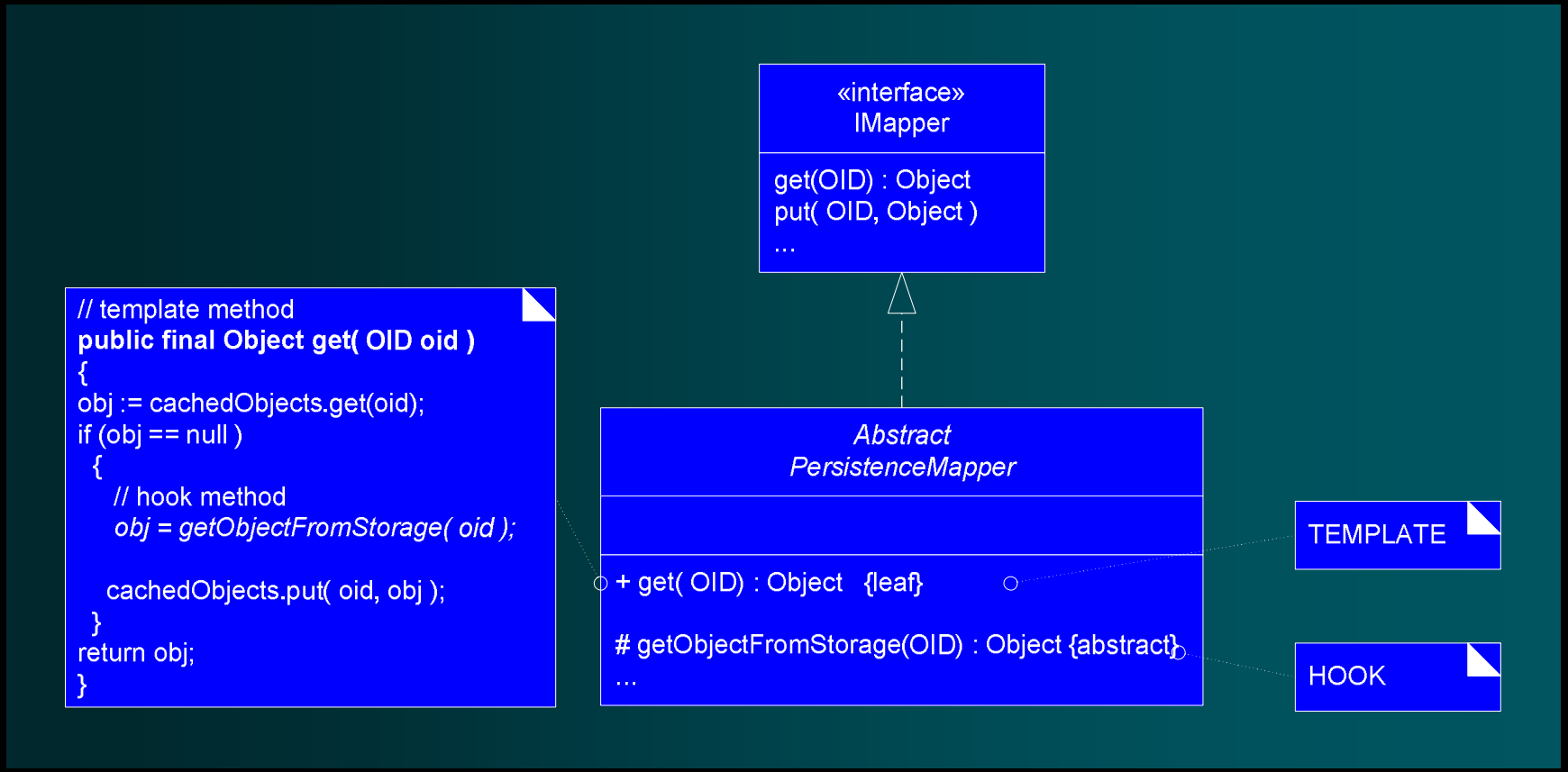
37.11. Framework Design with the Template Method Pattern

- Some commonality in mapper classes

```
if (object in cache)
    return it
else
    create the object from its representation in storage
    save object in cache
    return it
```

The point of variation is how the object is created from storage.

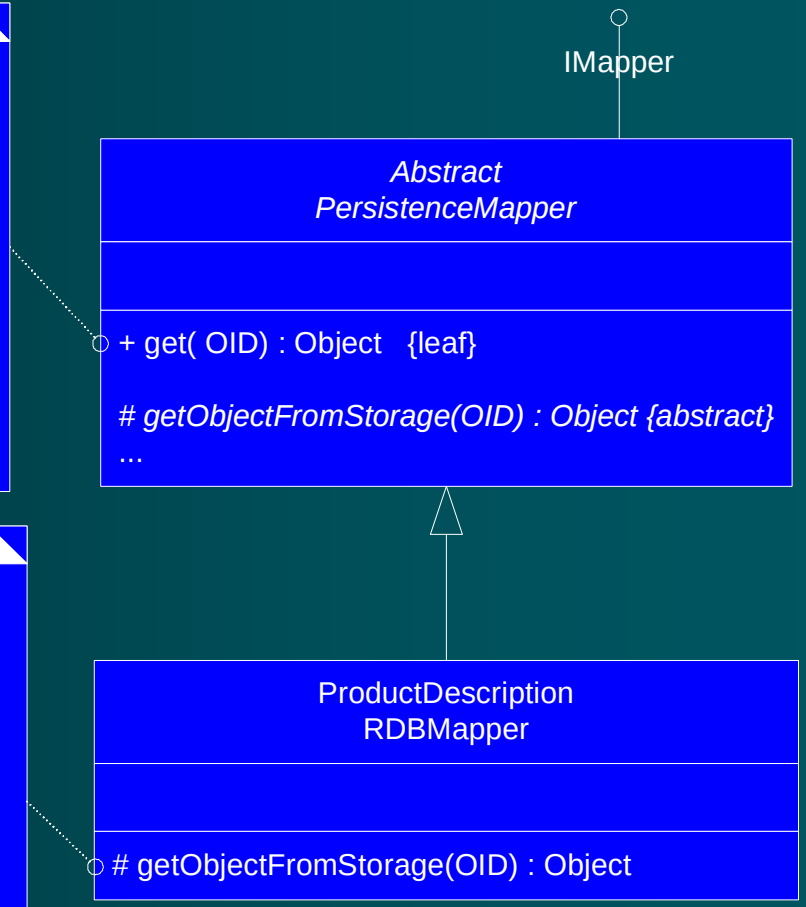
37.12. Materialization with the Template Method Pattern

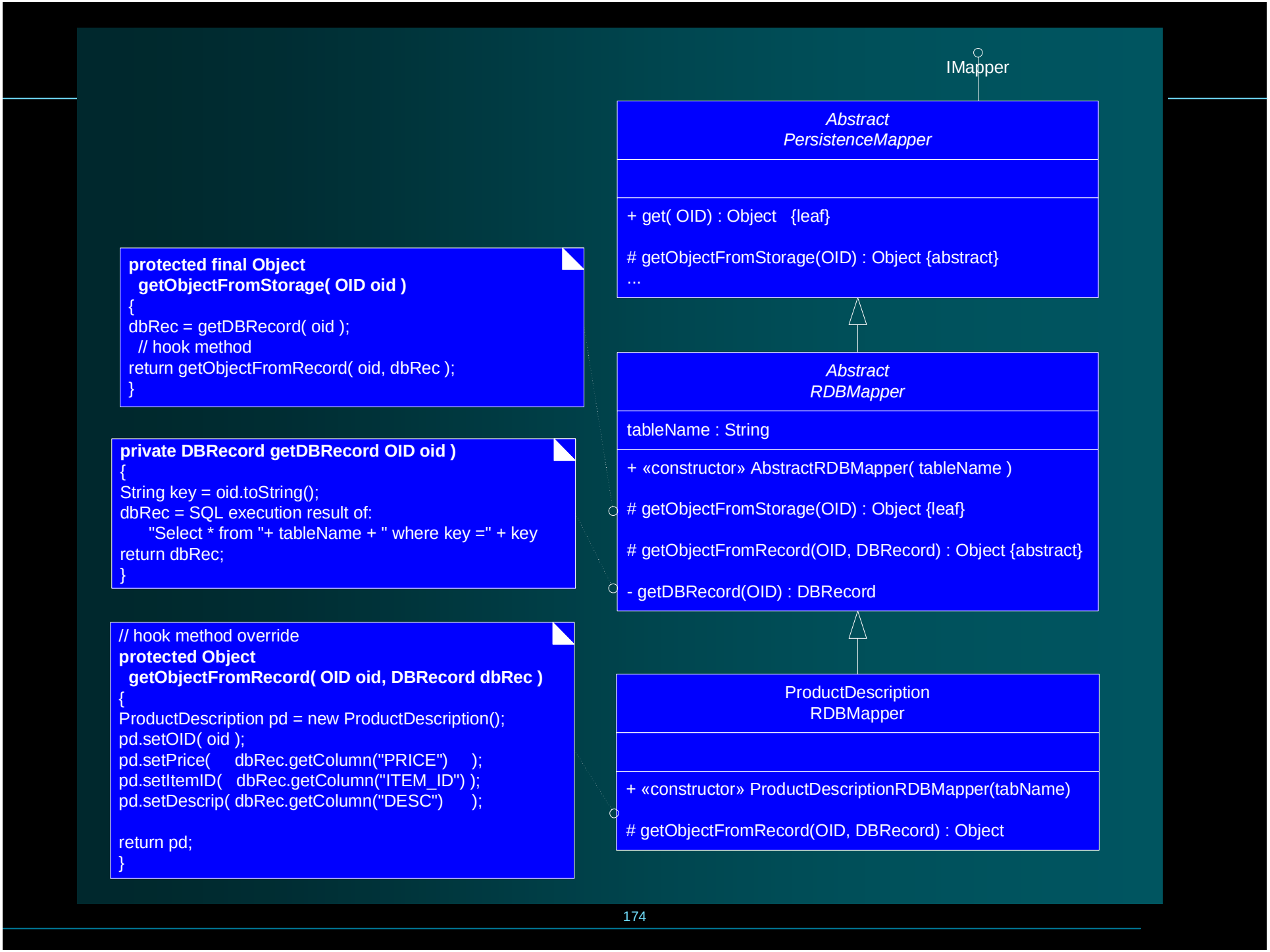


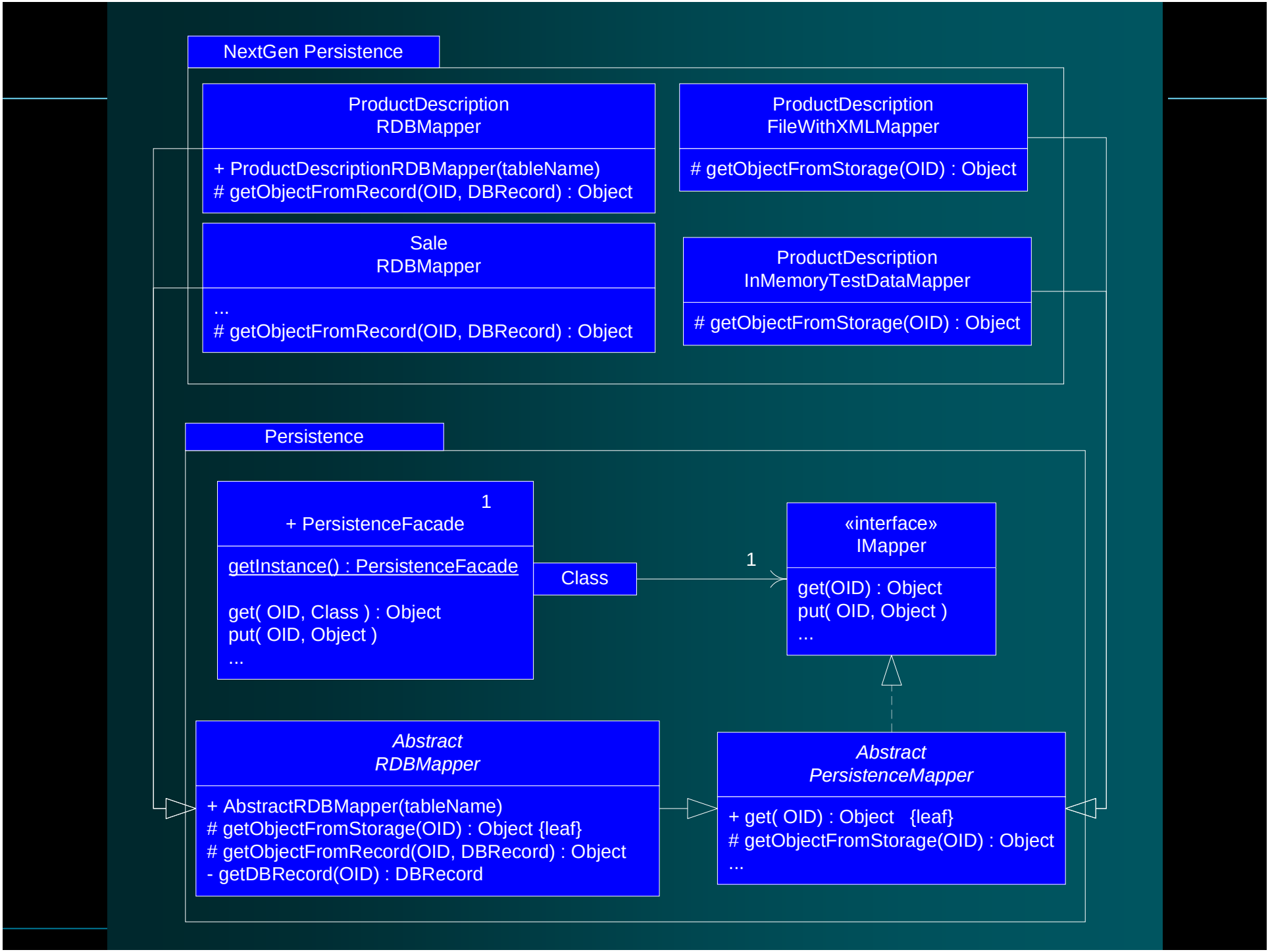
37.12. Materialization with the Template Method Pattern

```
// template method
public final Object get( OID oid )
{
obj := cachedObjects.get(oid);
if (obj == null )
  {
     // hook method
     obj = getObjectFromStorage( oid );

    cachedObjects.put( oid, obj )
  }
return obj
}
```

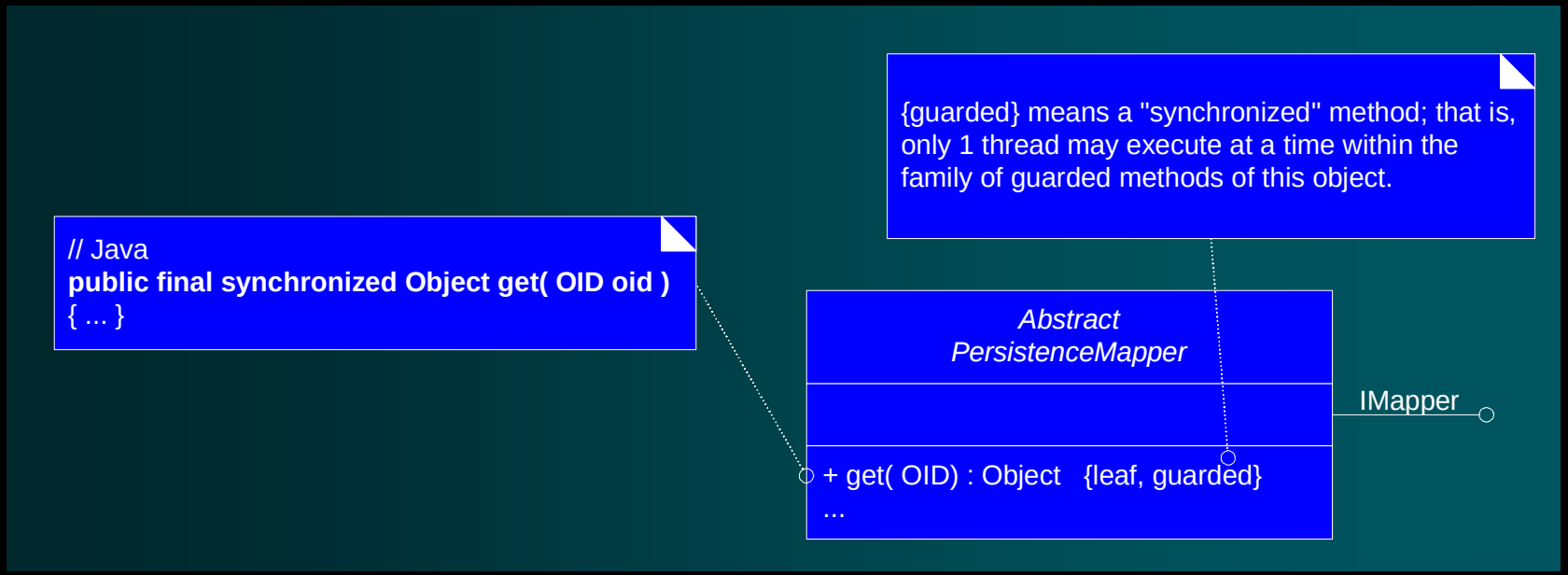
```
// hook method override
protected Object getObjectFromStorage( OID oid )
{
String key = oid.toString();
dbRec = SQL execution result of:
        "Select * from PROD_DESC where key =" + key

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;
}
```




IMapper
Abstract
PersistenceMapper
+ get(OID) : Object {leaf}
# getObjectFromStorage(OID) : Object {abstract}
...
protected final Object
getObjectFromStorage(OID oid)
{
dbRec = getDBRecord(oid);
// hook method
return getObjectFromRecord(oid, dbRec);
}
Abstract
RDBMapper
tableName : String
+ «constructor» AbstractRDBMapper(tableName)
# getObjectFromStorage(OID) : Object {leaf}
# getObjectFromRecord(OID, DBRecord) : Object {abstract}
- getDBRecord(OID) : DBRecord
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;
}
ProductDescription
RDBMapper
+ «constructor» ProductDescriptionRDBMapper(tabName)
# getObjectFromRecord(OID, DBRecord) : Object


NextGen Persistence
ProductDescription
RDBMapper
+ ProductDescriptionRDBMapper(tableName)
# getObjectFromRecord(OID, DBRecord) : Object
ProductDescription
FileWithXMLMapper
getObjectFromStorage(OID) : Object
Sale
RDBMapper
...
getObjectFromRecord(OID, DBRecord) : Object
ProductDescription
InMemoryTestDataMapper
getObjectFromStorage(OID) : Object
Persistence
1
+ PersistenceFacade
getInstance() : PersistenceFacade
get(OID, Class) : Object
put(OID, Object)
...
Class
1
«interface»
IMapper
get(OID) : Object
put(OID, Object)
...
Abstract
RDBMapper
+ AbstractRDBMapper(tableName)
# getObjectFromStorage(OID) : Object {leaf}
# getObjectFromRecord(OID, DBRecord) : Object
- getDBRecord(OID) : DBRecord
Abstract
PersistenceMapper
+ get(OID) : Object {leaf}
# getObjectFromStorage(OID) : Object
...

37.12. Materialization with the Template Method Pattern



37.13. Configuring Mappers with a MapperFactory

```
class MapperFactory {
    public IMapper getProductDescriptionMapper()
    {...}
    public IMapper getSaleMapper()
    {...}
    ...
}

class MapperFactory {
    public Map getAllMappers()
    {...}
     ...
}
```

37.15. Consolidating and Hiding SQL Statements in One Class

- To avoid hard-coding SQL statements into different RDB mapper classes
  - There is a single Pure Fabrication class RDBOperations where all SQL operations are consolidated.
  - The RDB mapper classes collaborate with it to obtain a DB record or record set.

```
class RDBOperations {
    public ResultSet getProductDescriptionData( OID oid )
    {...}
     public ResultSet getSaleData( OID oid )
     {...}
    ...
}
```

37.15. Consolidating and Hiding SQL Statements in One Class

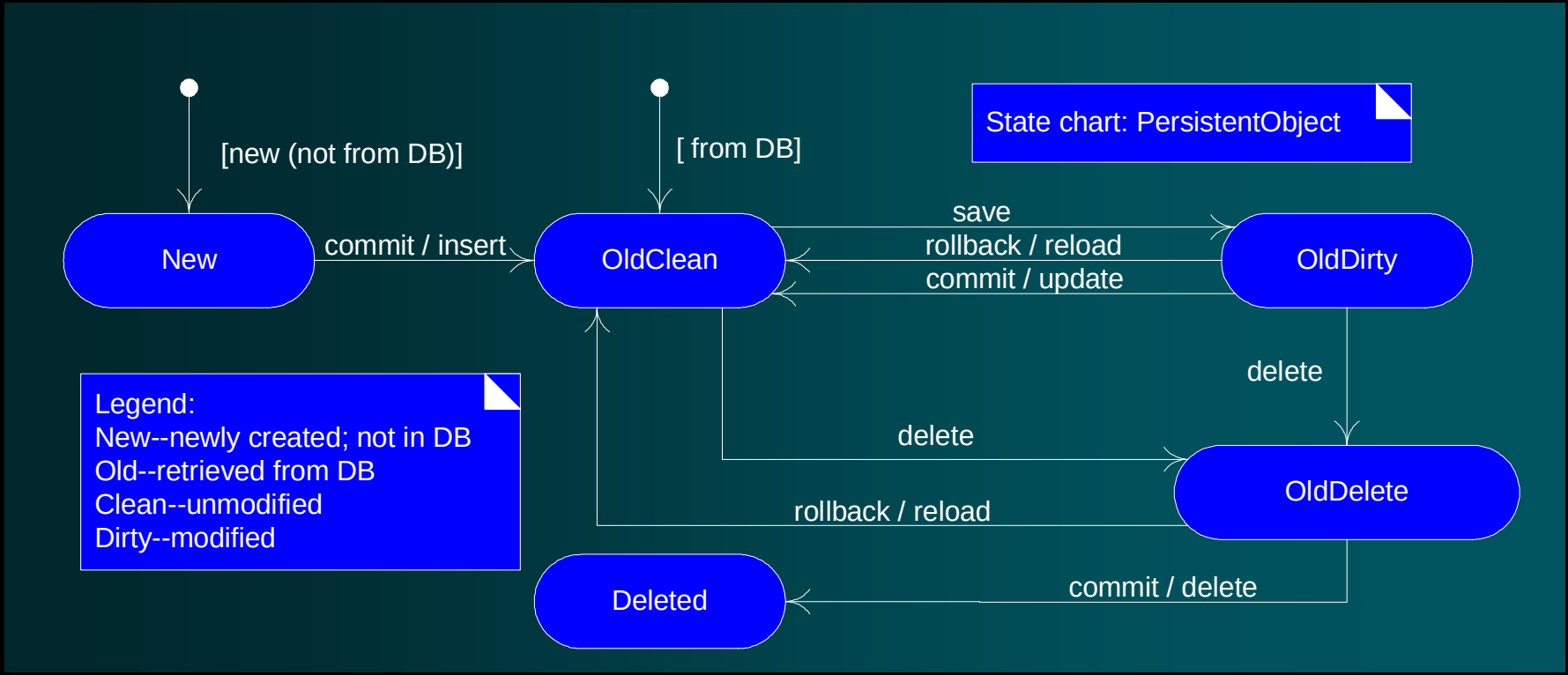
```
class ProductDescriptionRDBMapper extends
    AbstractPersistenceMapper {
    protected Object getObjectFromStorage( OID oid ) {
          ResultSet rs = RDBOperations.getInstance().
                getProductDescriptionData( oid );
          ProductDescription ps = new ProductDescription();
          ps.setPrice( rs.getDouble( "PRICE" ) );
          ps.setOID( oid );
          return ps;
     }
}
```

- Benefits accrue from this Pure Fabrication:
  - Ease of maintenance and performance tuning by an expert.
  - Encapsulation of the access method and details.

37.16. Transactional States and the State Pattern

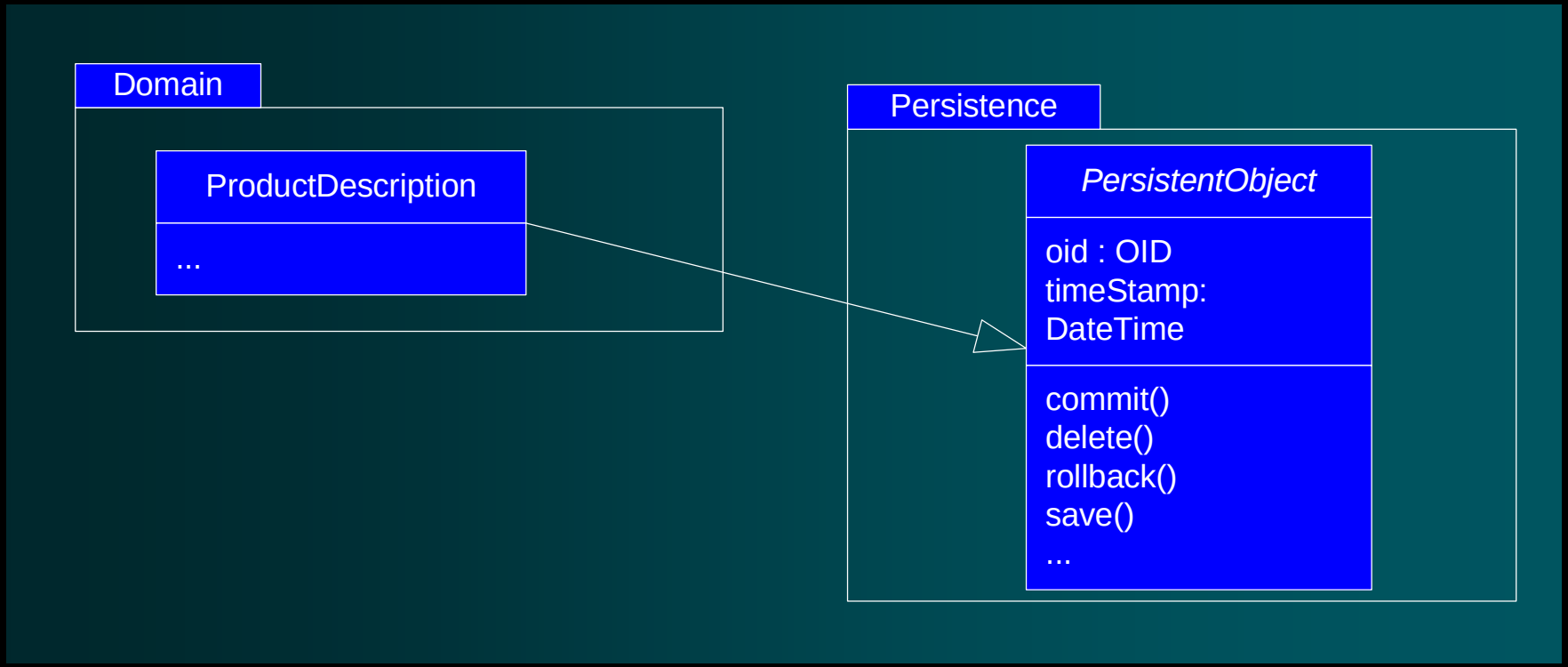
- Assuming following
  - Persistent objects can be inserted, deleted, or modified.
  - Operating on a persistent object does not cause an immediate database update; rather, an explicit commit operation must be performed.

37.16. Transactional States and the State Pattern



Statechart for PersistentObject

37.16. Transactional States and the State Pattern



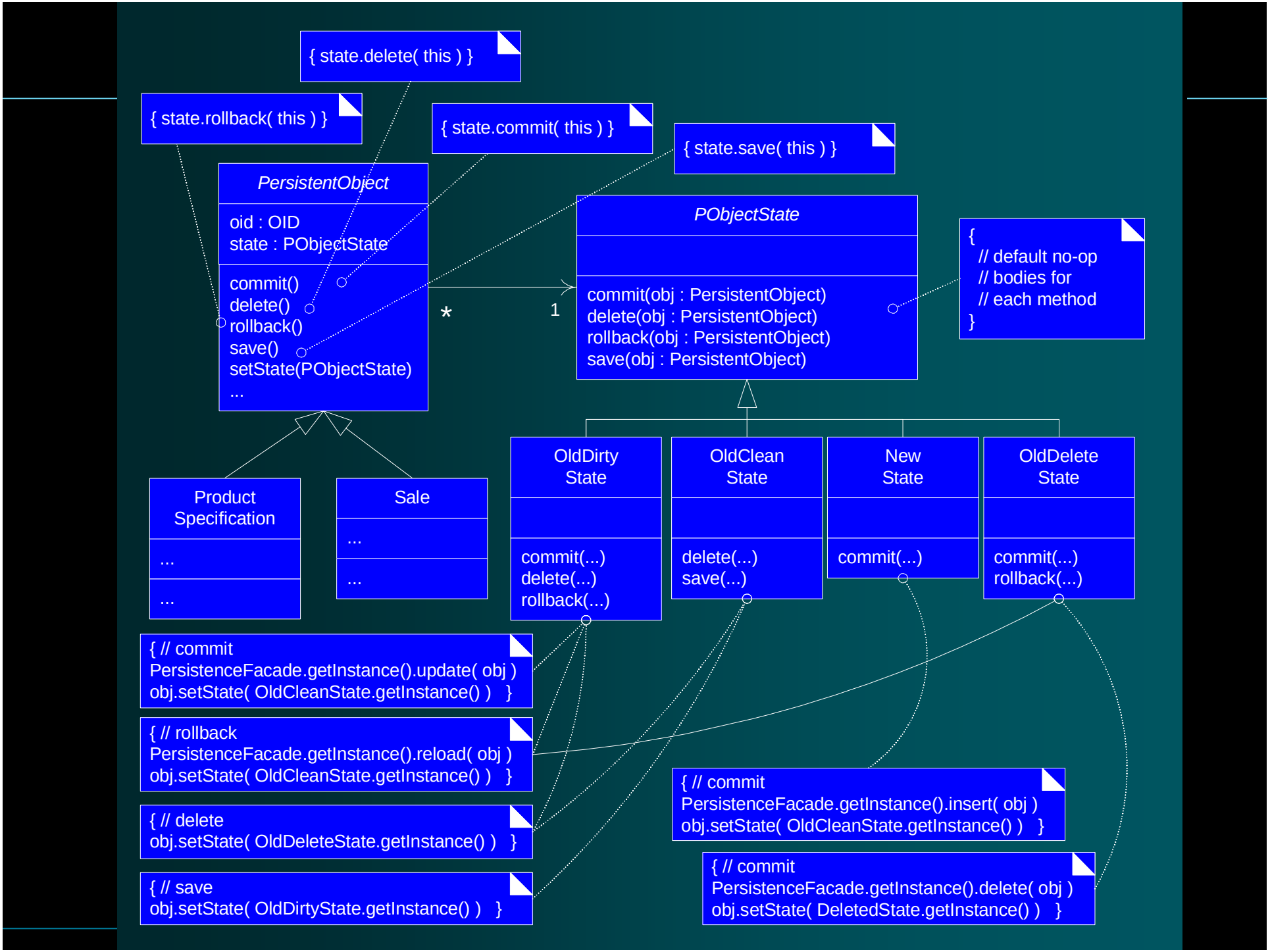
37.16. Transactional States and the State Pattern

```
public void commit() {
   switch ( state ) {
   case OLD_DIRTY:
            // ...
            break;
   case OLD_CLEAN:
            //...
            break;
   ...
  }
}
```

```
public void rollback() {
    switch ( state ) {
      case OLD_DIRTY:
            // ...
            break;
      case OLD_CLEAN:
            //...
            break;
        ...
      }
}
```

37.16. Transactional States and the State Pattern

- **State**
- 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?
- Solution
  - 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.

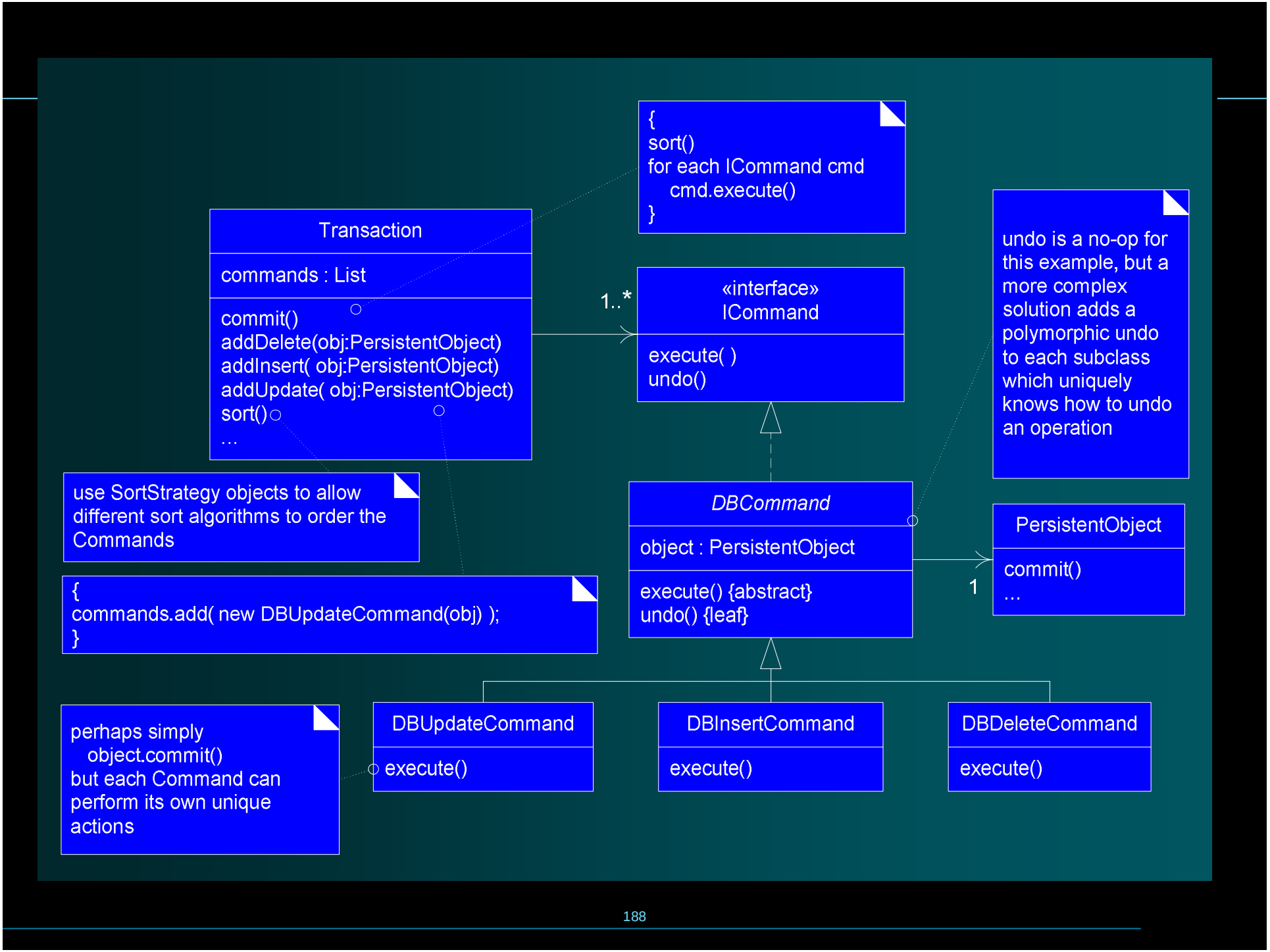

{ state.delete(this) }
{ state.rollback(this) }
{ state.commit(this) }
{ state.save(this) }
PersistentObject
oid : OID
state : PObjectState
commit()
delete()
rollback()
save()
setState(PObjectState)
...
*
1
PObjectState
commit(obj : PersistentObject)
delete(obj : PersistentObject)
rollback(obj : PersistentObject)
save(obj : PersistentObject)
{
// default no-op
// bodies for
// each method
}
Product Specification
...
...
Sale
...
...
OldDirty State
commit(...)
delete(...)
rollback(...)
OldClean State
delete(...)
save(...)
New State
commit(...)
OldDelete State
commit(...)
rollback(...)
{ // 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()) }

37.17. Designing a Transaction with the Command Pattern

- Transaction
  - Is a unit of work whose tasks must all complete successfully, or none must be completed.
    - Suppose the database has a referential integrity constraint such that when a record is updated in TableA that contains a foreign key to a record in TableB, the database requires that the record in TableB already exists.
    - Suppose a transaction contains an INSERT task to add the TableB record, and an UPDATE task to update the TableA record. If the UPDATE executes before the INSERT, a referential integrity error is raised.

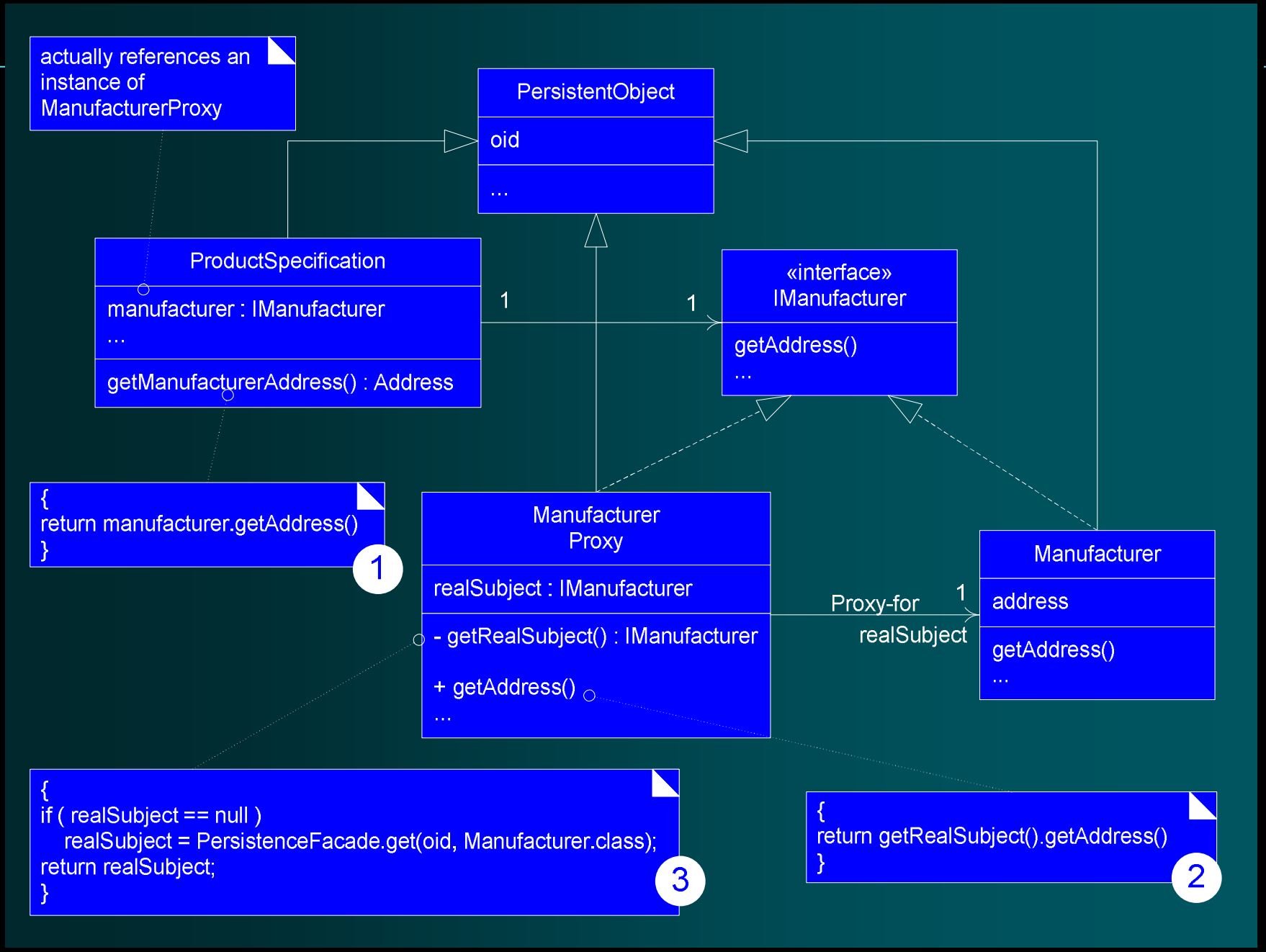
37.17. Designing a Transaction with the Command Pattern

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


{
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()
undo is a no-op for this example, but a more complex solution adds a polymorphic undo to each subclass which uniquely knows how to undo an operation
use SortStrategy objects to allow different sort algorithms to order the Commands
{
commands.add(new DBUpdateCommand(obj));
}
DBCommand
object : PersistentObject
execute() {abstract}
undo() {leaf}
1
PersistentObject
commit()
...
perhaps simply
object.commit()
but each Command can perform its own unique actions
DBUpdateCommand
execute()
DBInsertCommand
execute()
DBDeleteCommand
execute()

37.18. Lazy Materialization with a Virtual Proxy

- Lazy Materialization
  - Defer the materialization of an object until it is absolutely required.
- Virtual Proxy
  - a proxy for another object that materializes the real subject when it is first referenced;


actually references an instance of ManufacturerProxy
PersistentObject
oid
...
ProductSpecification
manufacturer : IManufacturer
...
getManufacturerAddress() : Address
1
1
«interface»
IManufacturer
getAddress()
...
{
return manufacturer.getAddress()
}
1
Manufacturer
Proxy
realSubject : IManufacturer
- getRealSubject() : IManufacturer
+ getAddress()
...
Proxy-for
realSubject
1
Manufacturer
address
getAddress()
...
{
if (realSubject == null)
realSubject = PersistenceFacade.get(oid, Manufacturer.class);
return realSubject;
}
3
{
return getRealSubject().getAddress()
}
2

37.18. Lazy Materialization with a Virtual Proxy

```
// EAGER MATERIALIZATION OF MANUFACTURER
class ProductDescriptionRDBMapper extends
    AbstractPersistenceMapper {
    protected Object getObjectFromStorage( OID oid ) {
        ResultSet rs = RDBOperations.getInstance().
                getProductDescriptionData( oid );
        ProductDescription ps = new ProductDescription();
        ps.setPrice( rs.getDouble( "PRICE" ) );

        // here's the essence of it
        String manufacturerForeignKey =
                rs.getString( "MANU_OID" );
        OID manuOID = new OID( manufacturerForeignKey );
        ps.setManufacturer( (IManufacturer)
                PersistenceFacade.getInstance().
                get(manuOID,Manufacturer.class);
        ...
    }
}
```

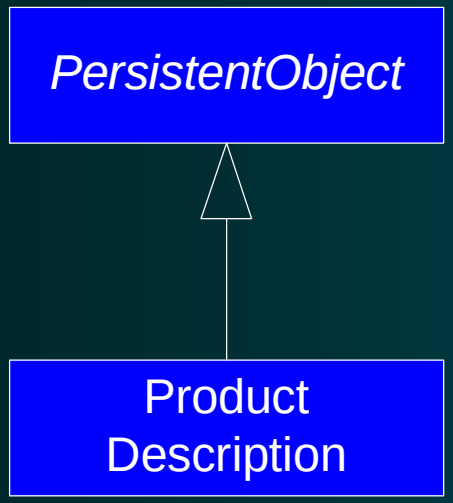
37.18. Lazy Materialization with a Virtual Proxy

```
// LAZY MATERIALIZATION OF MANUFACTURER
class ProductDescriptionRDBMapper extends
    AbstractPersistenceMapper {
    protected Object getObjectFromStorage( OID oid ) {
            ResultSet rs = RDBOperations.getInstance().
                        getProductDescriptionData( oid );
            ProductDescription ps = new ProductDescription();
            ps.setPrice( rs.getDouble( "PRICE" ) );

            // here's the essence of it
            String manufacturerForeignKey = rs.getString( "MANU_OID" );
            OID manuOID = new OID( manufacturerForeignKey );
            ps.setManufacturer( new ManufacturerProxy( manuOID ) );
             ...
    }
}
```

37.19. How to Represent Relationships in Tables

- Representing Object Relationships as Tables pattern
  - one-to-one associations
    - Place an OID foreign key in one or both tables representing the objects in relationship.
    - Or, create an associative table that records the OIDs of each object in relationship.
  - one-to-many associations, such as a collection
    - Create an associative table that records the OIDs of each object in relationship.
  - many-to-many associations
    - Create an associative table that records the OIDs of each object in relationship.

37.20. PersistentObject Superclass and Separation of Concerns



possible design, but problematic in terms of coupling and mixing the technical service concern of persistence with the application logic of a domain object.

37.21. Unresolved Issues

- dematerializing objects
  - Briefly, the mappers must define putObjectToStorage methods. Dematerializing composition hierarchies requires collaboration between multiple mappers and the maintenance of associative tables (if an RDB is used).
- materialization and dematerialization of collections
- queries for groups of objects
- thorough transaction handling
- error handling when a database operation fails
- multiuser access and locking strategies
- securitycontrolling access to the database

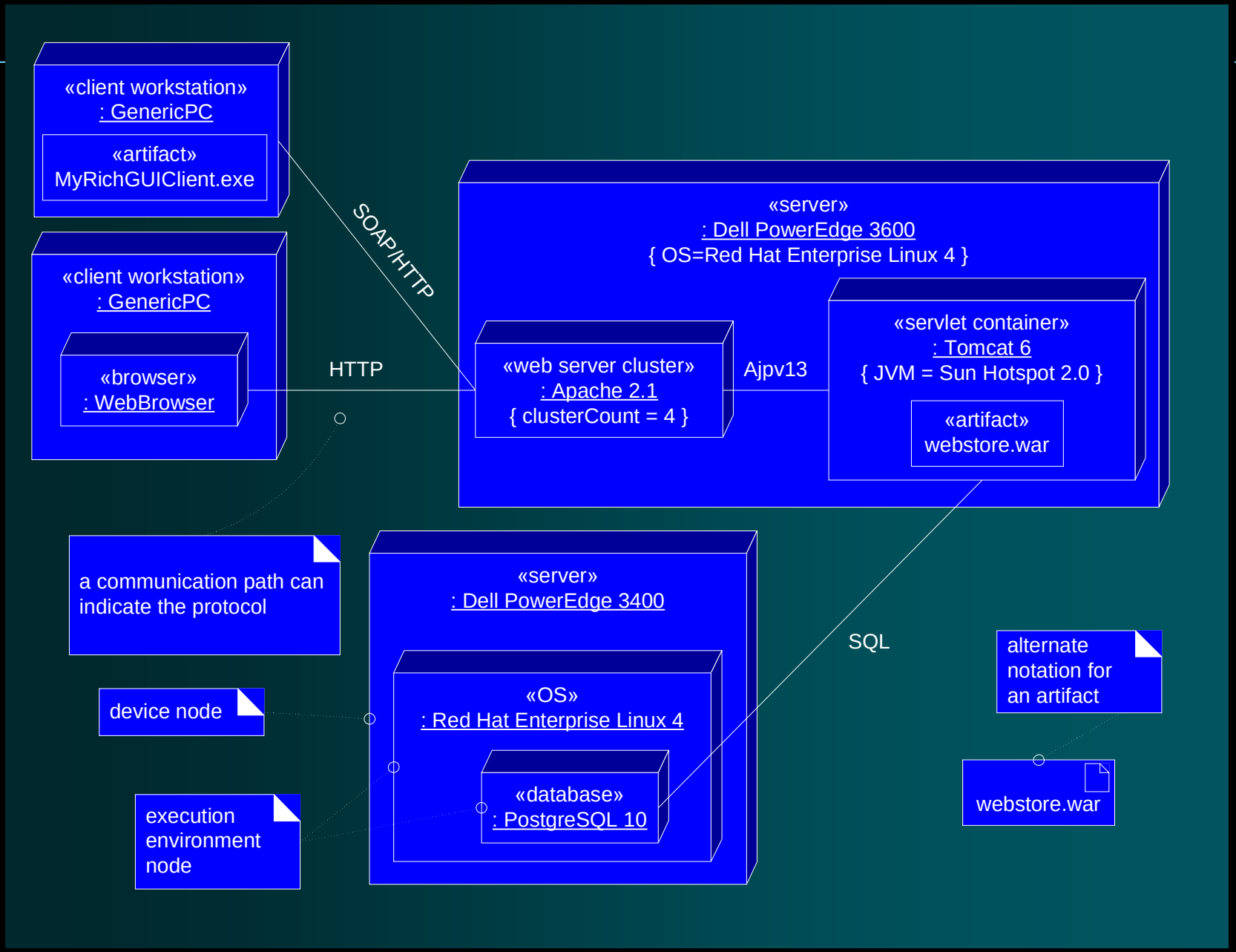
Chapter 38: UML Deployment and Component Diagrams

Objective

- Summarize UML deployment and component diagram notation.

38.1. Deployment Diagrams

- A Deploy Diagram shows
  - the assignment of concrete software artifacts (such as executable files) to computational nodes (something with processing services)
  - the deployment of software elements to the physical architecture and the communication between physical elements.


«client workstation»
: GenericPC
«artifact»
MyRichGUIClient.exe
«client workstation»
: GenericPC
«browser»
: WebBrowser
SOAP/HTTP
HTTP
«server»
: Dell PowerEdge 3600
{ OS=Red Hat Enterprise Linux 4 }
«web server cluster»
: Apache 2.1
{ clusterCount = 4 }
Ajpv13
«servlet container»
: Tomcat 6
{ JVM = Sun Hotspot 2.0 }
«artifact»
webstore.war
a communication path can indicate the protocol
«server»
: Dell PowerEdge 3400
SQL
alternate notation for an artifact
device node
«OS»
: Red Hat Enterprise Linux 4
«database»
: PostgreSQL 10
execution environment node
webstore.war

38.1. Deployment Diagrams

- Basic element of a deployment diagram
  - Device node
    - A physical computing resource with processing and memory services to execute software.

38.1. Deployment Diagrams

- Execution Environment Node (EEN)
  - A software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements. For example:
    - an operating system (OS) is software that hosts and executes programs
    - a virtual machine hosts and executes programs
    - a database engine receives SQL program requests and executes them, and hosts/executes internal stored procedures
    - a Web browser hosts and executes JavaScript, Java applets, Flash, and other executable technologies
    - a workflow engine
    - a servlet container or EJB container

38.2. Component Diagrams

- A component represents a modular part of a system
  - encapsulates its contents and whose manifestation is replaceable within its environment.
- Uses a UML component to emphasize that
  - the interfaces are important,
  - it is modular, self-contained and replaceable.

38.2. Component Diagrams