An Introduction to Object-Oriented Analysis and Design

PART IV: Elaboration Iteration 2 More Patterns

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

Chapter 23: Iteration 2 – More Patterns

Objective

- Define the requirement for iteration 2

23.1 From Iteration 1 to 2

- When iteration 1 ends, the following should be accomplished
  - All the software has been tested
  - Customers have been regularly engaged in evaluating the partial system
  - The system has been completely integrated and stabilized as baselined internal release.

23.2 Iteration-2 Requirements and Emphasis

- Focuses on object design and patterns
- NextGen Pos
  - Support for variation in third-party external services.
  - Complex pricing rules
  - A design to refresh a GUI window when the sale total changes.

23.2 Iteration-2 Requirements and Emphasis

- Monopoly
  - Special square rules apply.
    - Each player receives $1500 at the beginning.
    - When a player lands on the Go square, the player receives $200.
    - When a player lands on the Go-To-Jail square, they move to the jail square.
    - When a player lands on the Income-Tax square, the player pays the minimum of $200 or 10% of their worth

Chapter 24: Quick Analysis Update

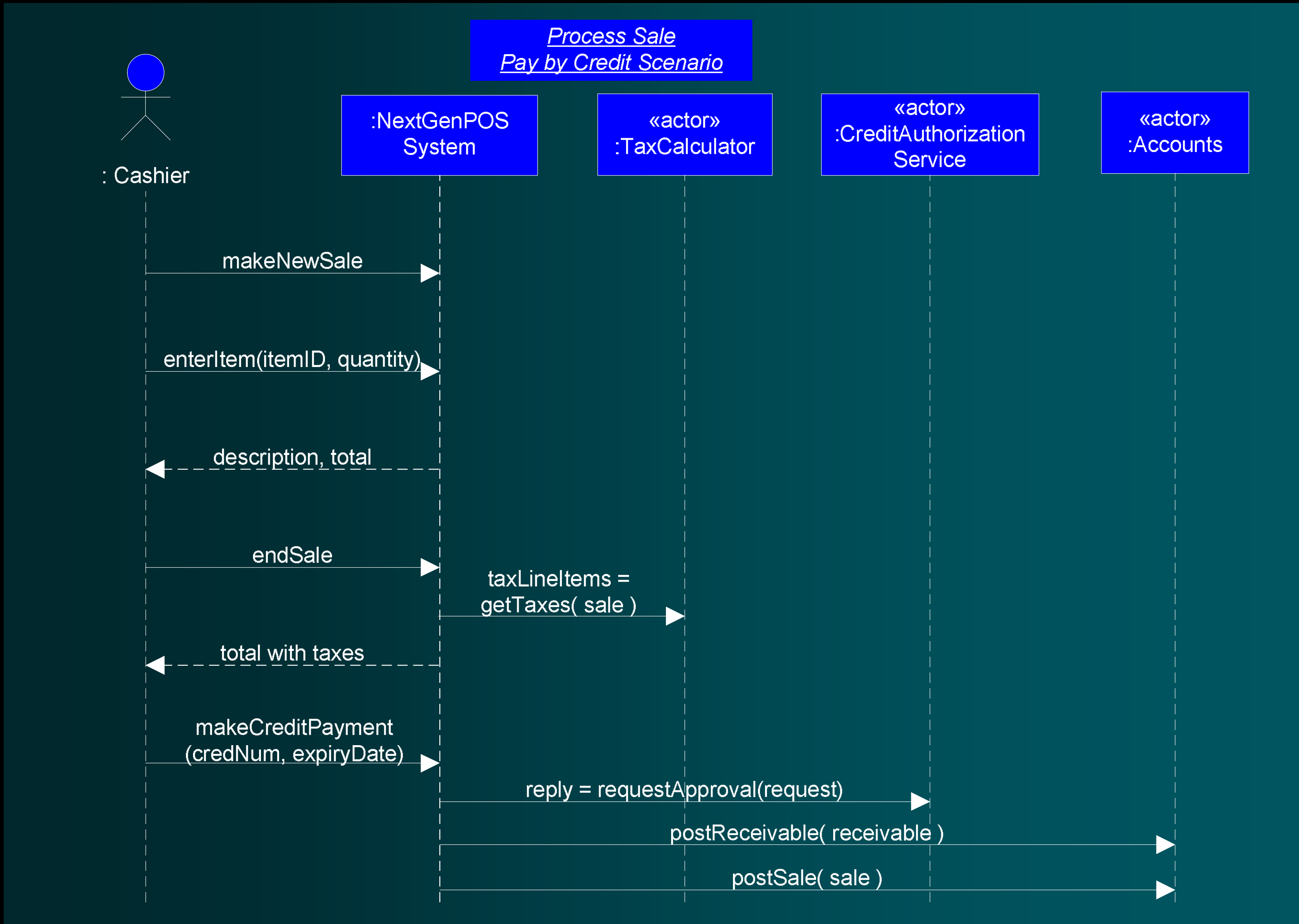
Objective

- Quickly highlight some analysis artifact changes, especially in the Monopoly domain model.

24.1. Case Study: NextGen POS

- Use Cases
  - No refinement is needed for the use cases this iteration
- SSDs
  - Add support for third-party external systems with varying interfaces

24.1. Case Study: NextGen POS

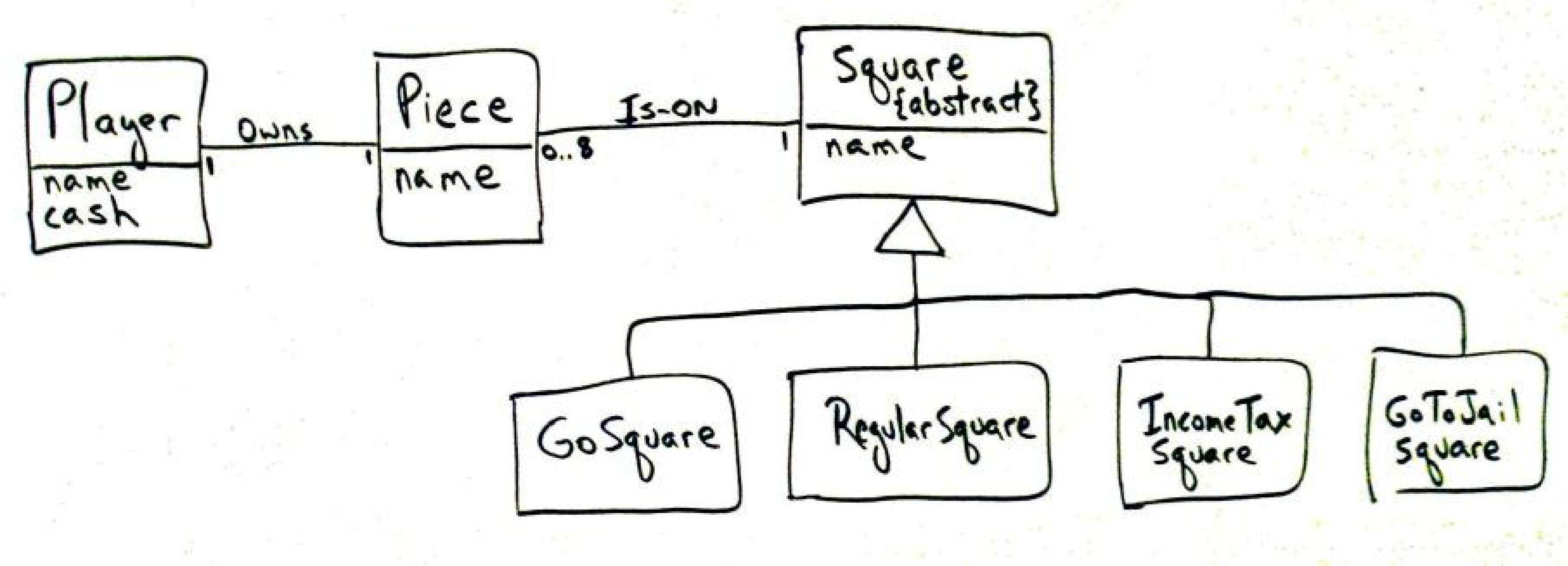


24.1. Case Study: NextGen POS

- Domain Model
  - Do not involve many new domain concepts
  - A sign of process maturity with the UP
    - Understanding when creating an artifact will add significant value,
    - A kind of mechanical "make work" step and better skipped.

24.2. Case Study: Monopoly

- Use Case
  - skipped
- Domain Model



24.2. Case Study: Monopoly

- 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, in noteworthy ways.

Chapter 25: GRASP: More Objects with Responsibilities

Objective

- Learn to apply the remaining GRASP patterns.

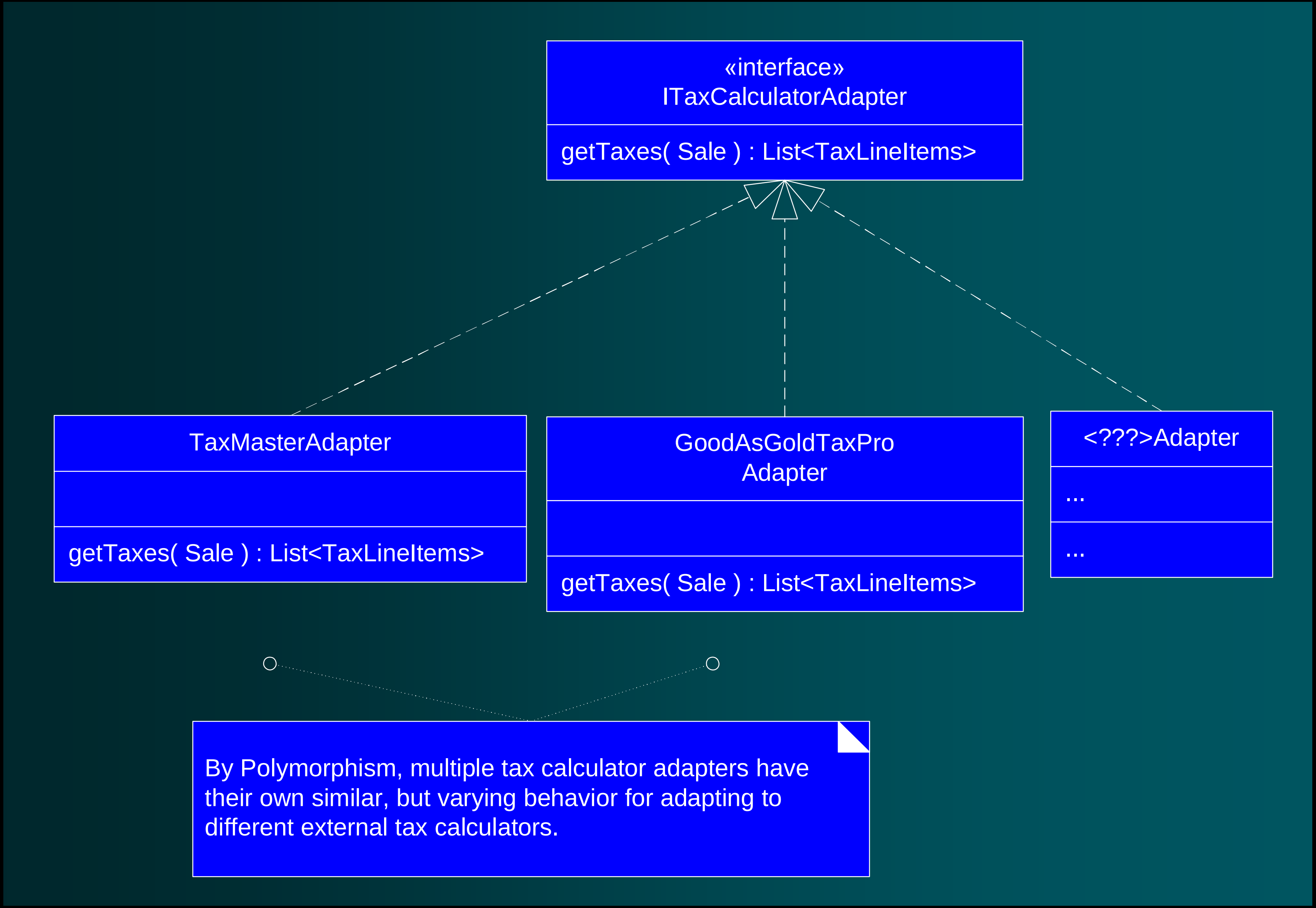
25.1. Polymorphism

- Problem
  - How to handle alternatives based on type?
  - How to create pluggable software components?
  - if-then-else or case statement conditional logic makes it difficult to extend a program with new variations

25.1. Polymorphism

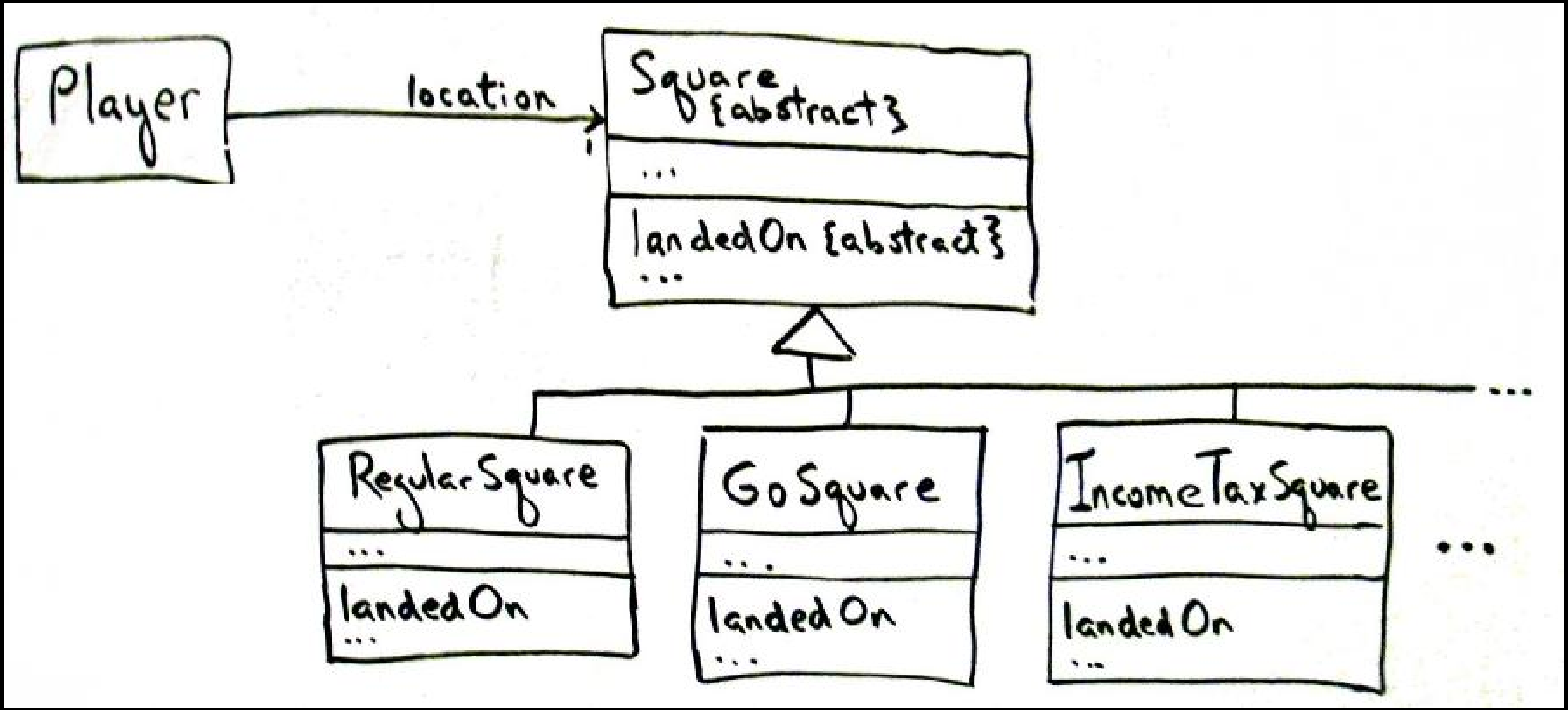
- Solution
  - Assign responsibility for the behavior--using polymorphic operations--to the types for which the behavior varies.
  - Give the same name to services in different objects
- Examples
  - NextGen Problem:
    - How to Support Third-Party Tax Calculators?

25.1. Polymorphism

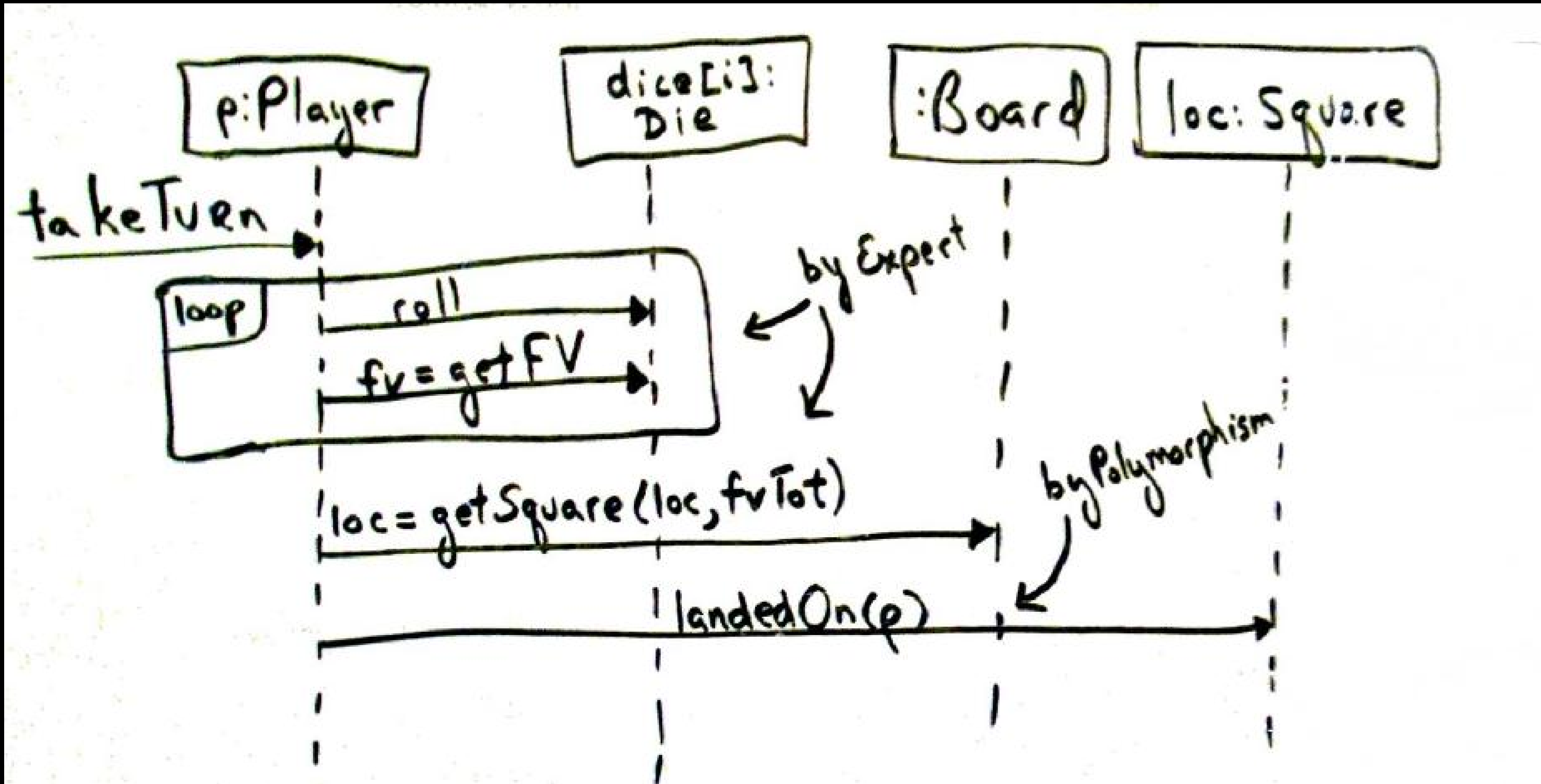


25.1. Polymorphism

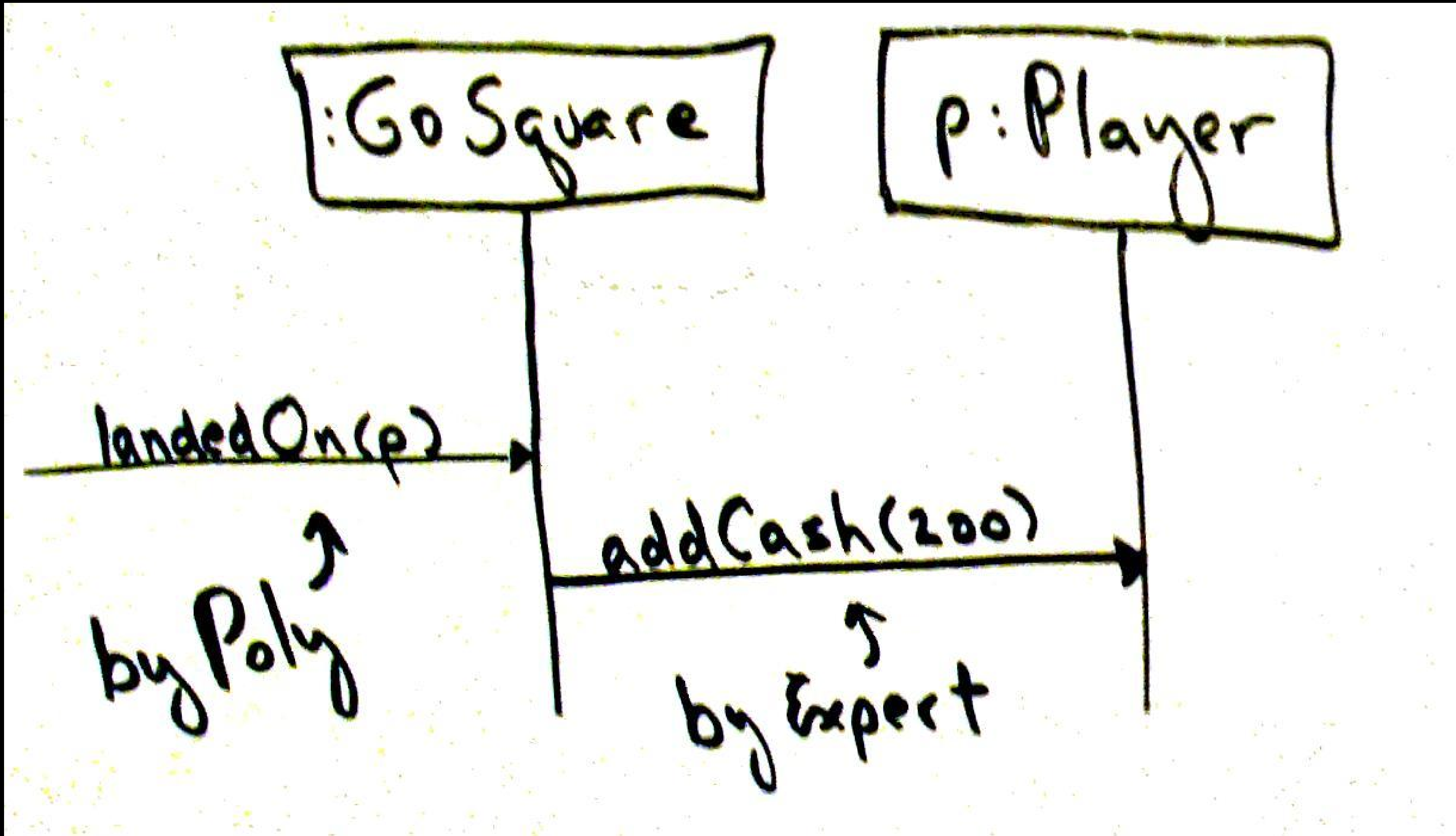
- Monopoly Problem:
  - How to Design for Different Square Actions?



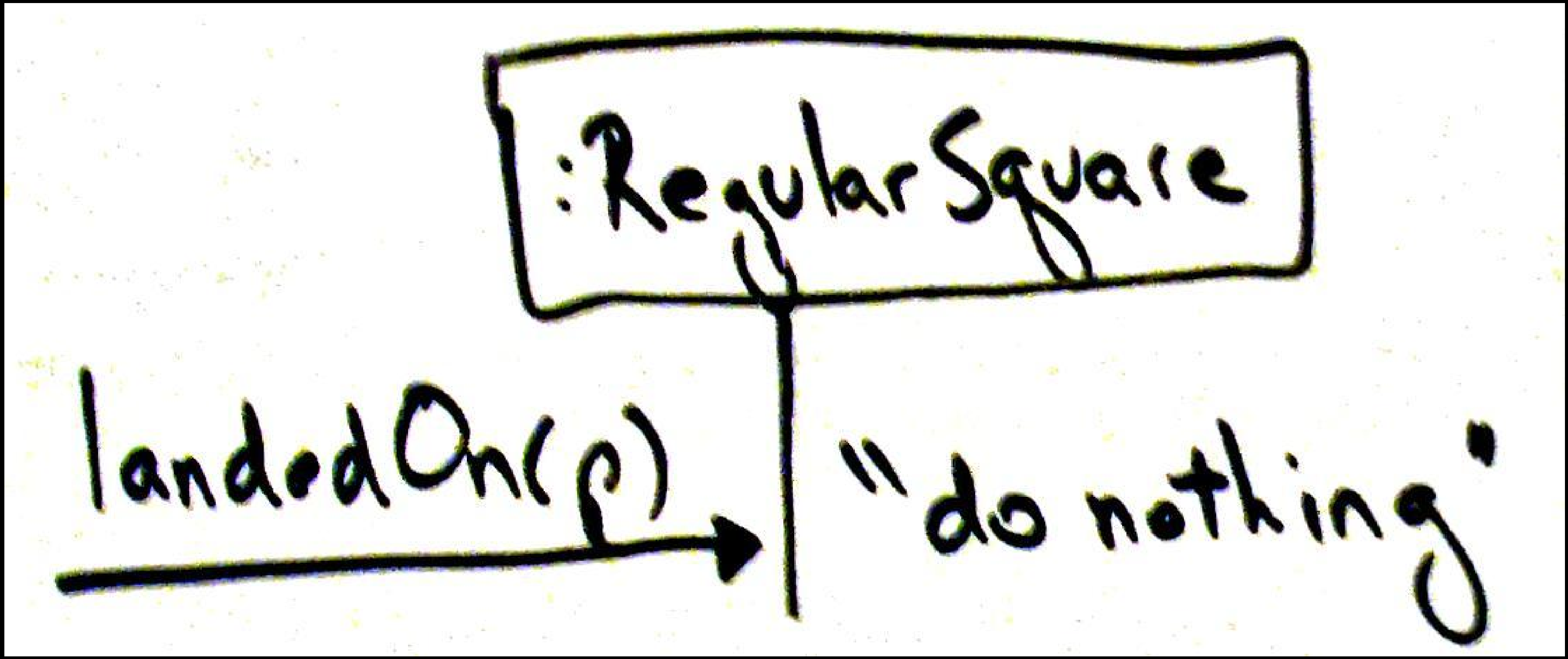
25.1. Polymorphism



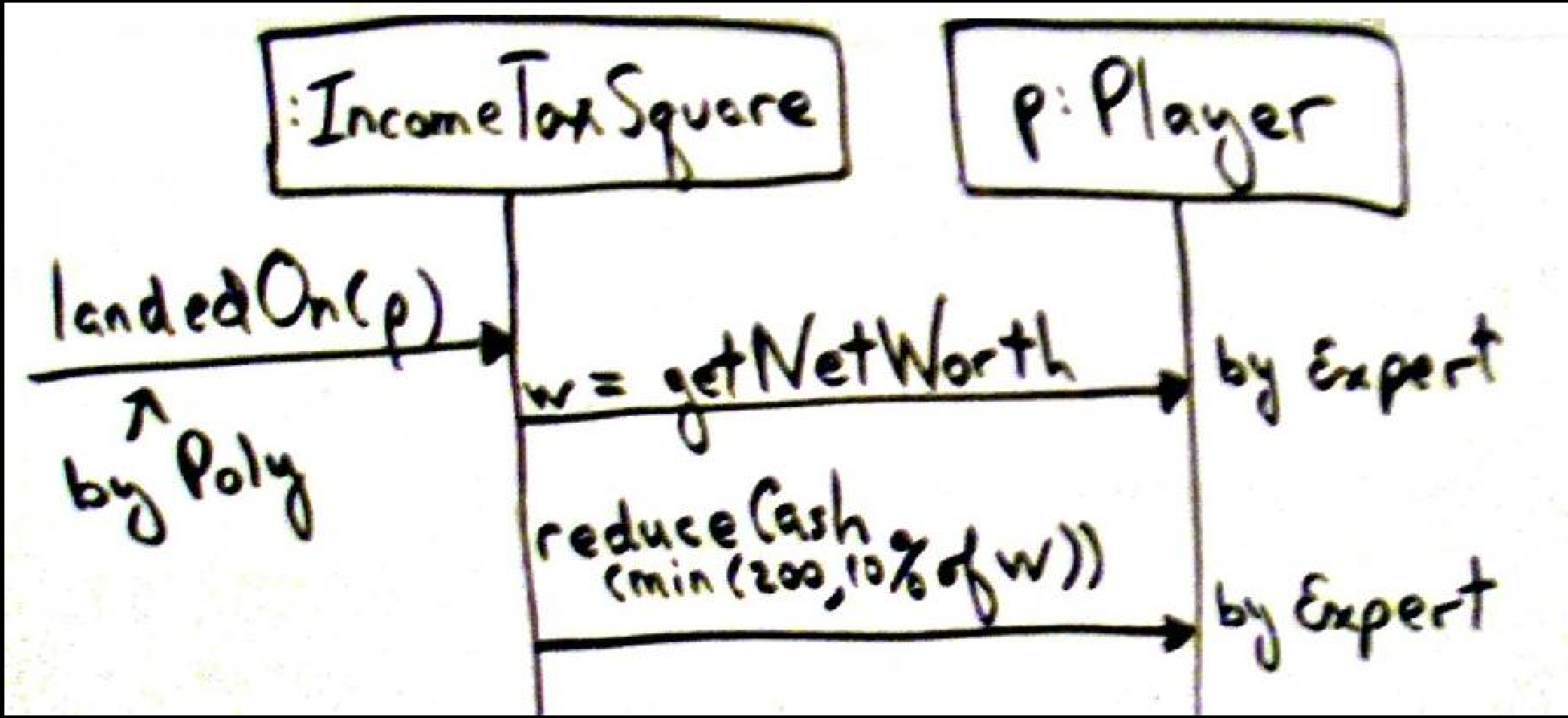
25.1. Polymorphism



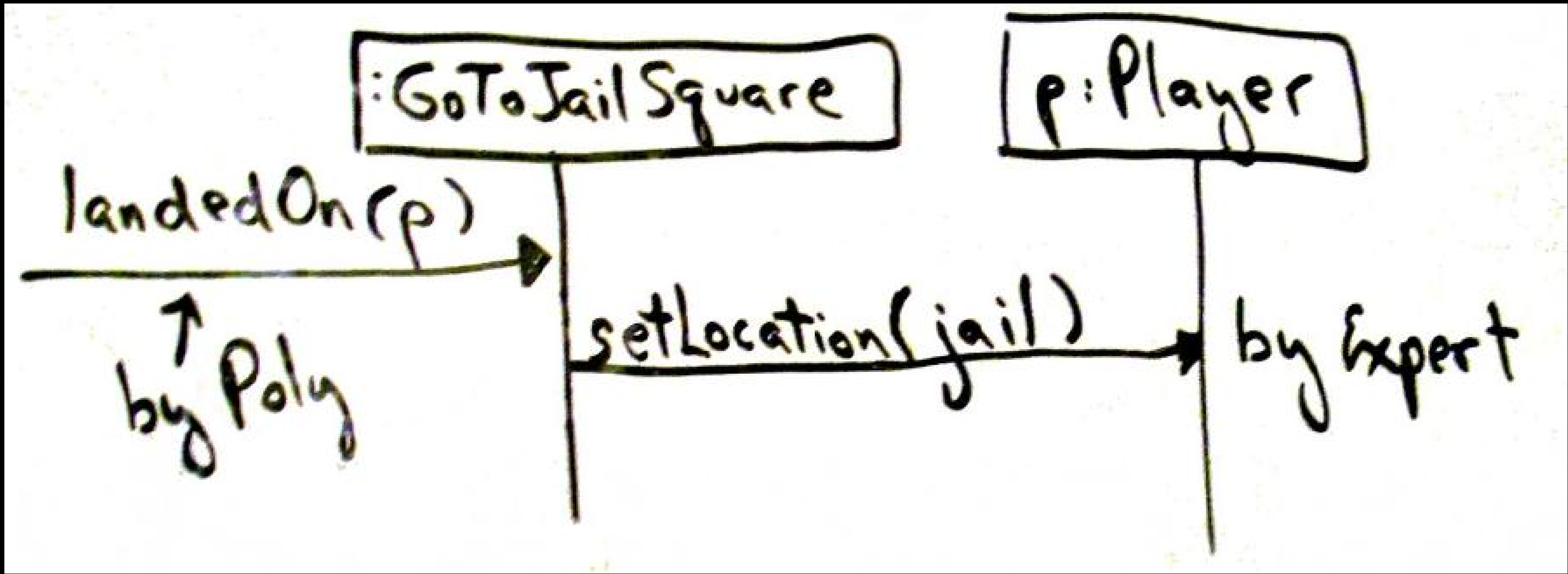
25.1. Polymorphism

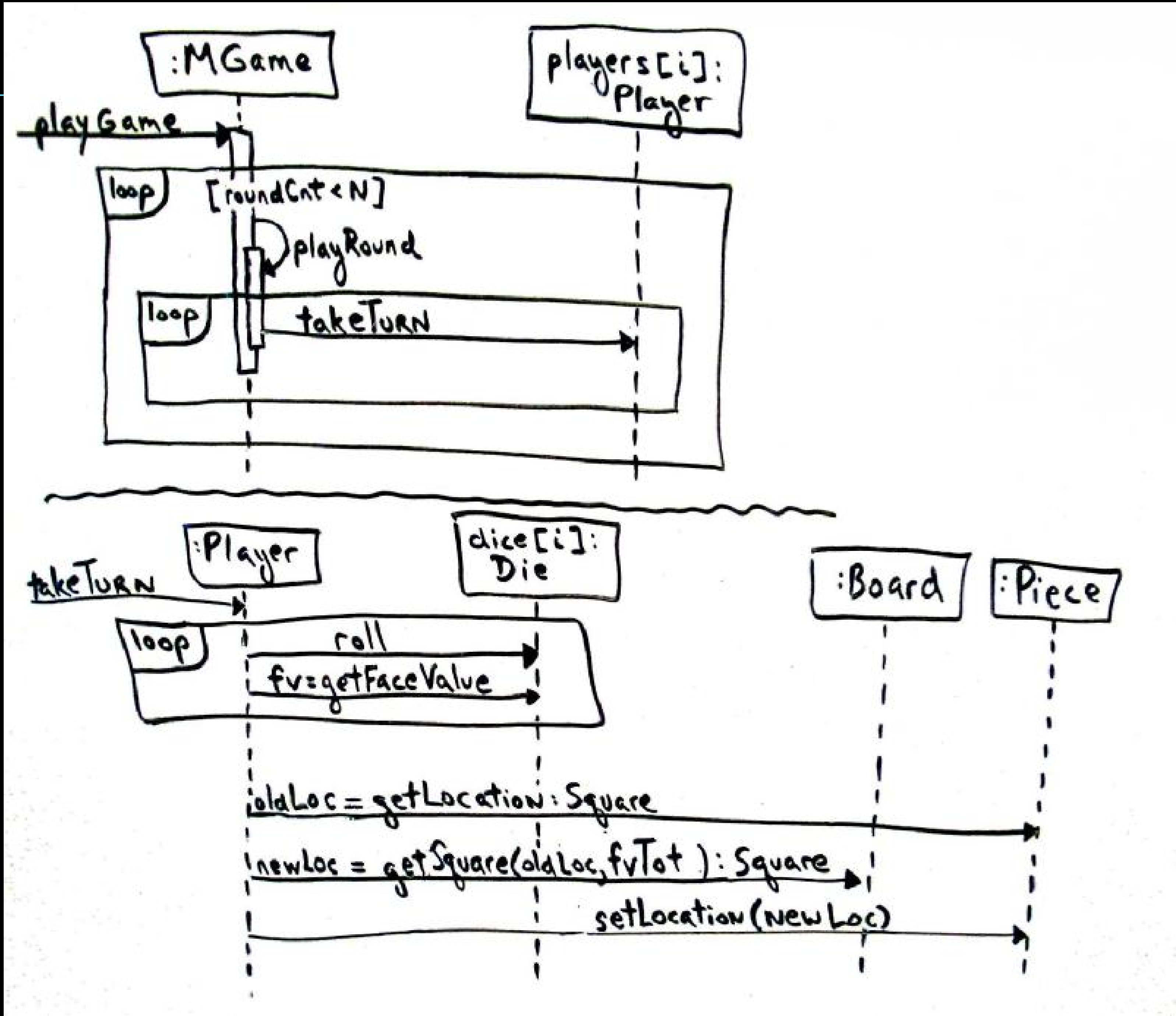


25.1. Polymorphism



25.1. Polymorphism




:MGame
players[i]:
Player
playGame
loop
[roundCnt < N]
playRound
loop
takeTurn
:Player
dice[i]:
Die
:Board
:Piece
takeTurn
loop
roll
fv=getFaceValue
oldLoc = getLocation : Square
newLoc = getSquare(oldLoc, fvTot) : Square
setLocation (newLoc)

25.1. Polymorphism

- Improving the Coupling
  - Player, Piece and Square
    - The Player knows the Piece
    - The Piece knows the Square where it stand on
    - The Player use the Square frequently
  - Refined the design
    - the Player rather than the Piece knows its square

25.1. Polymorphism

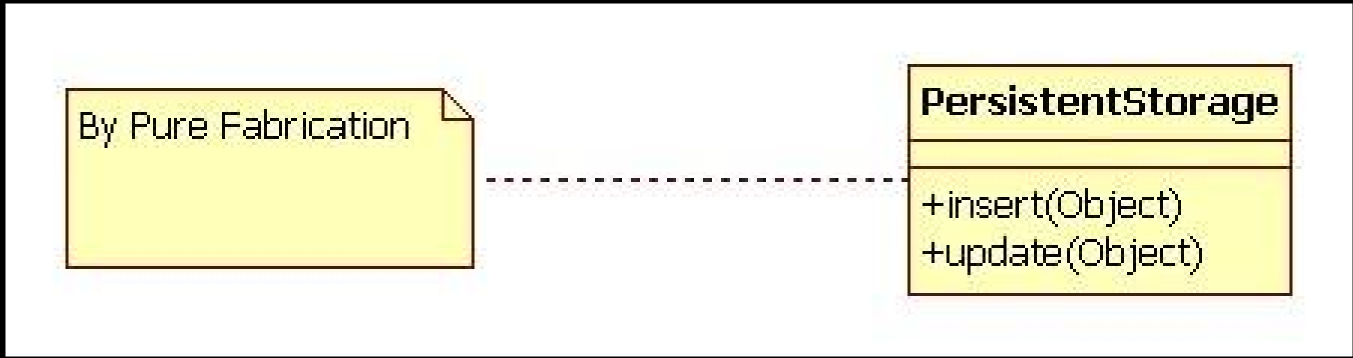
- Guideline: When to Design with Interfaces?
  - Introduce one when you want to support polymorphism without being committed to a particular class hierarchy.
- Contraindications
  - Be realistic about the true likelihood of variability before investing in increased flexibility.
- Benefits
  - Extensions required for new variations are easy to add.
  - New implementations can be introduced without affecting clients.

25.2. Pure Fabrication

- Problem
  - What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert are not appropriate?
- Solution
  - Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept, to support high cohesion, low coupling, and reuse.

- Examples
  - NextGen Problem: Saving a Sale Object in a Database
    - The task requires a relatively large number of supporting database-oriented operations.
    - The Sale class has to be coupled to the relational database interface.
    - Saving objects in a relational database is a very general task for which many classes need support.

25.2. Pure Fabrication

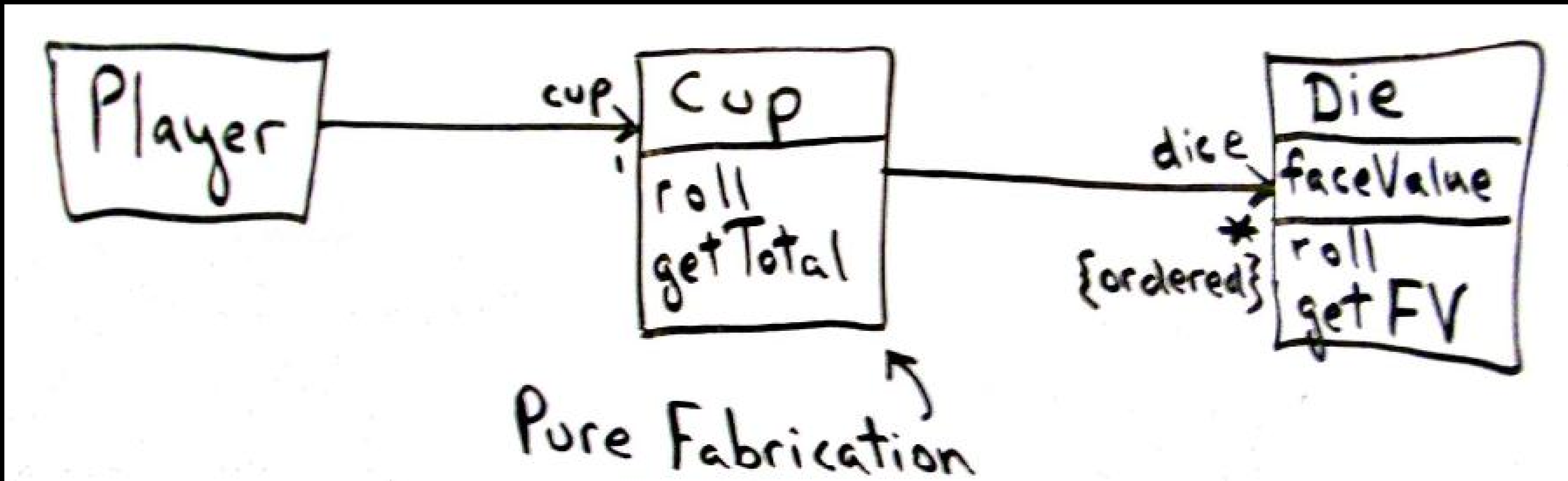


- This Pure Fabrication solves the following design problems:
  - The Sale remains well-designed, with high cohesion and low coupling.
  - The PersistentStorage class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium.
  - The PersistentStorage class is a very generic and reusable object.

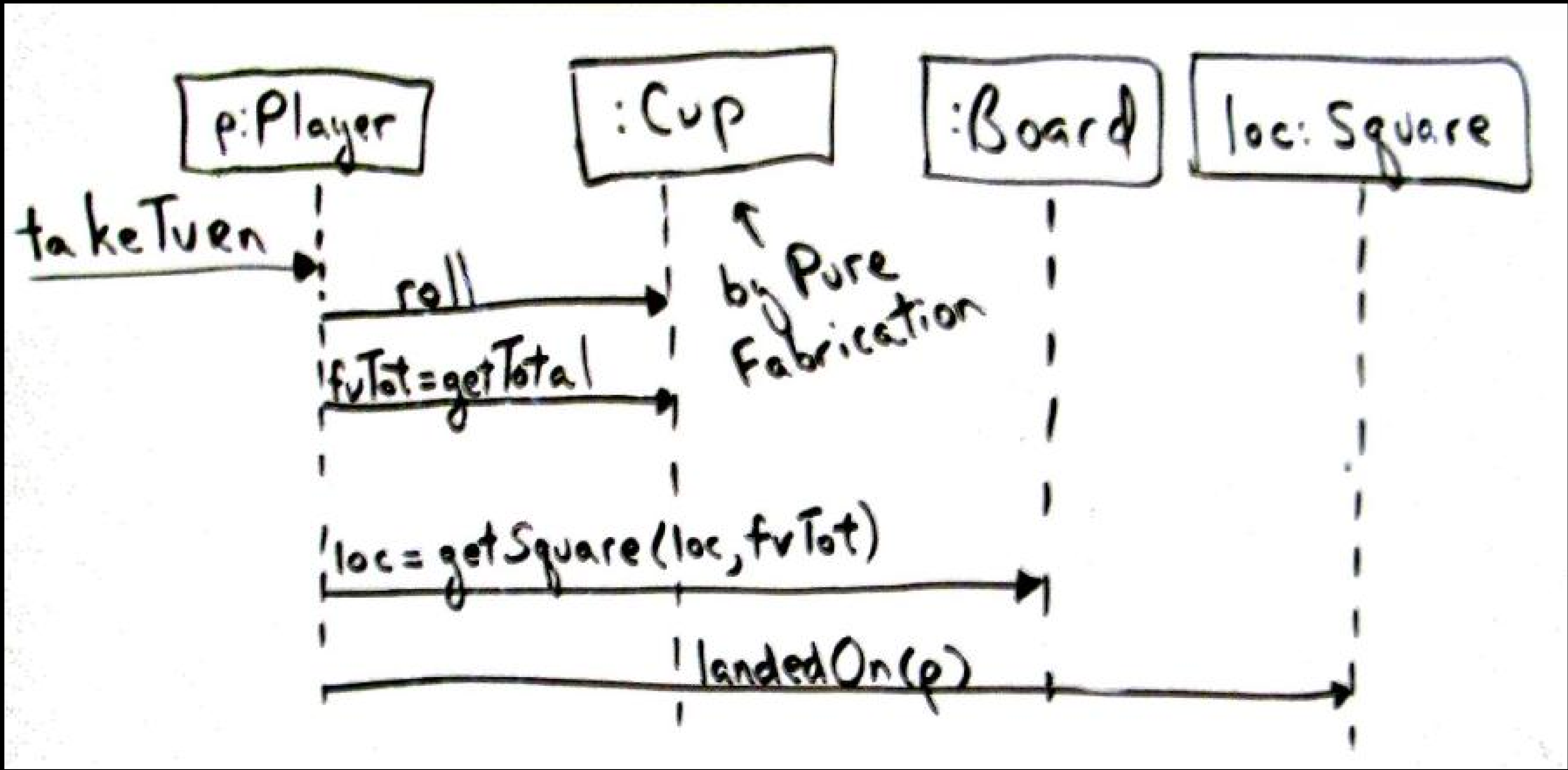
25.2. Pure Fabrication

- Monopoly Problem: Handling the Dice
  - In the current design,
    - The Player rolls all the dice and sums the total.
    - By putting this rolling and summing responsibility in a Monopoly game Player,
      - the summing service is not generalized for use in other games.
      - It is not possible to simply ask for the current dice total without rolling the dice again.
  - Propose a Pure Fabrication called Cup
    - to hold all the dice, roll them, and know their total.

25.2. Pure Fabrication



25.2. Pure Fabrication



25.2. Pure Fabrication

- Discussion
  - The design of objects can be broadly divided into two groups:
    - Those chosen by representational decomposition.
    - Those chosen by behavioral decomposition.
- Benefits
  - High Cohesion is supported
  - Reuse potential may increase.

25.2. Pure Fabrication

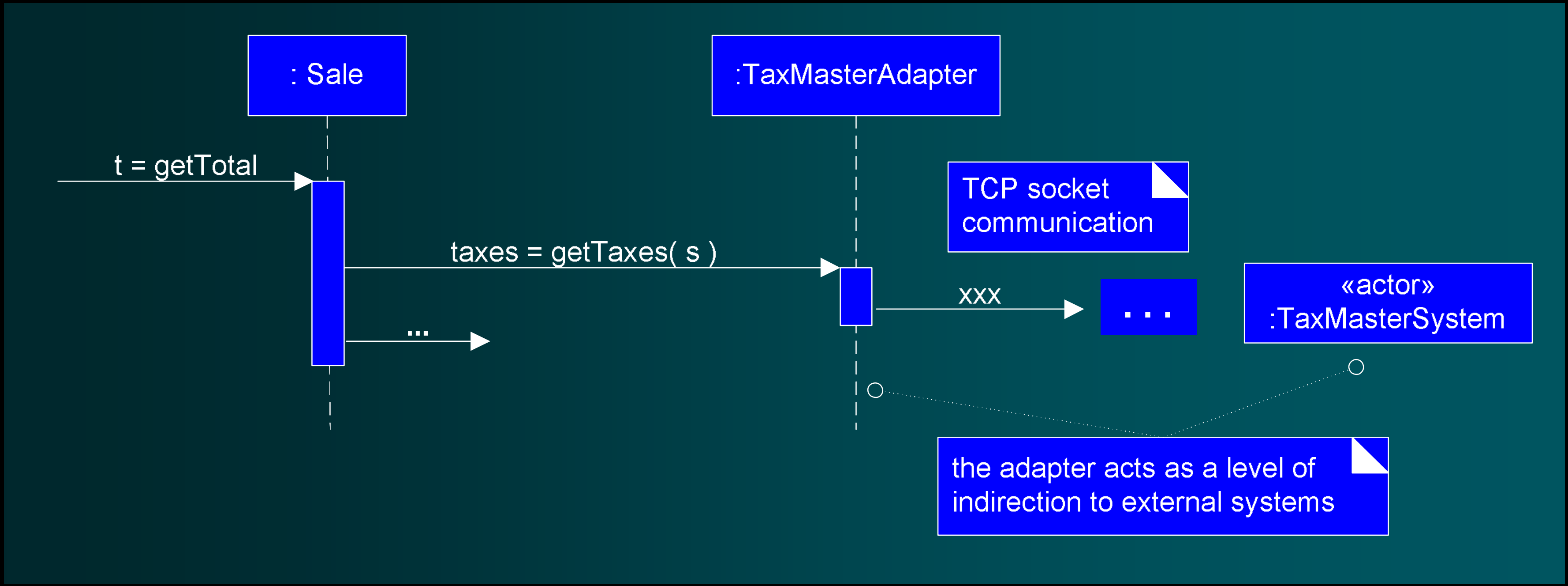
- Contraindications
  - sometimes overused by neophyte
    - lead to too many behavior objects that have responsibilities not co-located with the information required for their fulfillment.
    - The usual symptom is that most of the data inside the objects is being passed to other objects to reason with it.

25.3. Indirection

- Problem
  - Where to assign a responsibility, to avoid direct coupling between two (or more) things?
  - How to de-couple objects so that low coupling is supported and reuse potential remains higher?
- Solution
  - Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

25.3. Indirection

- Examples
  - TaxCalculatorAdapter



  - PersistentStorage

25.3. Indirection

- Discussion
  - Most problems in computer science can be solved by another level of indirection.
- Benefits
  - Lower coupling between components.

25.4. Protected Variations

- Problem
  - How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- Solution
  - Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

25.4. Protected Variations

- Example
  - The prior external tax calculator problem
    - Its solution with Polymorphism illustrate Protected Variations
- Discussion
  - A very important, fundamental principle of software design

25.4. Protected Variations

- Mechanisms Motivated by Protected Variations
  - the maturation of a developer or architect can be seen in their growing knowledge of mechanisms to
    - achieve PV,
    - pick the appropriate PV battles worth fighting,
    - their ability to choose a suitable PV solution.

25.4. Protected Variations

- **Core Protected Variations Mechanisms**
  - Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV.
  - virtual machines and operating systems are complex examples of indirection to achieve PV.
- **Data-Driven Designs**
  - Reading information from an external source in order to change the behavior of   system in some way at run-time.
    - reading codes, values, class file paths, class names
  - The system is protected from the impact of data, metadata, or declarative variations by externalizing the variant, reading it in, and reasoning with it.

25.4. Protected Variations

- Service Lookup
  - Using naming services (for example, Java's JNDI) or traders to obtain a service (for example, Java's Jini, or UDDI for Web services)
  - Clients are protected from variations in the location of services.

25.4. Protected Variations

- Interpreter-Driven Designs
  - rule interpreters that execute rules read from an external source,
  - script or language interpreters that read and run programs,
  - virtual machines, neural network engines that execute nets, constraint logic engines that read and reason with constraint sets
  - The system is protected from the impact of logic variations by externalizing the logic, reading it in.

25.4. Protected Variations

- Reflective or Meta-Level Designs
  - Using Java.beans.Introspector to obtain a BeanInfo object,
    - asking for the getter Method object for bean property X,
    - calling Method.invoke.
  - The system is protected from the impact of logic or external code variations by reflective algorithms that use introspection and meta-language services.

25.4. Protected Variations

- Uniform Access
  - aCircle.radius may invoke a radius():float method or directly refer to a public field, depending on the definition of the class.
  - We can change from public fields to access methods, without changing the client code.
- Standard Languages
  - Official language standards such as SQL provide protection against a proliferation of varying languages.

25.4. Protected Variations

- The Liskov Substitution Principle (LSP)
  - formalizes the principle of protection

What is wanted here is something like the following substitution property:
If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,
the behavior of P is unchanged when o1 is substituted for o2
then S is a subtype of T

25.4. Protected Variations

- Structure-Hiding Designs
  - Don't Talk to Strangers
    - within a method, messages should only be sent to the following objects:
      - The this object (or self).
      - A parameter of the method.
      - An attribute of this.
      - An element of a collection which is an attribute of this.
      - An object created within the method.
    - avoid coupling a client to knowledge of indirect objects and the object connections between objects.
    - farther along a path the program traverses, the more fragile it is.

25.4. Protected Variations

- Contraindications
  - Variation point
    - Variations in the existing, current system or requirements, such as the multiple tax calculator interfaces that must be supported.
  - Evolution point
    - Speculative points of variation that may arise in the future, but which are not present in the existing requirements.

25.4. Protected Variations

- Caution: Speculative PV and Picking Your Battles
  - Novice developers tend toward brittle designs,
  - Intermediate developers tend toward overly fancy and flexible, generalized ones (in ways that never get used).
  - Expert designers choose with insight; perhaps a simple and brittle design whose cost of change is balanced against its likelihood.

25.4. Protected Variations

- Benefits
  - Extensions required for new variations are easy to add.
  - New implementations can be introduced without affecting clients.
  - Coupling is lowered.
  - The impact or cost of changes can be lowered.

Chapter 26: Applying GoF Design Patterns

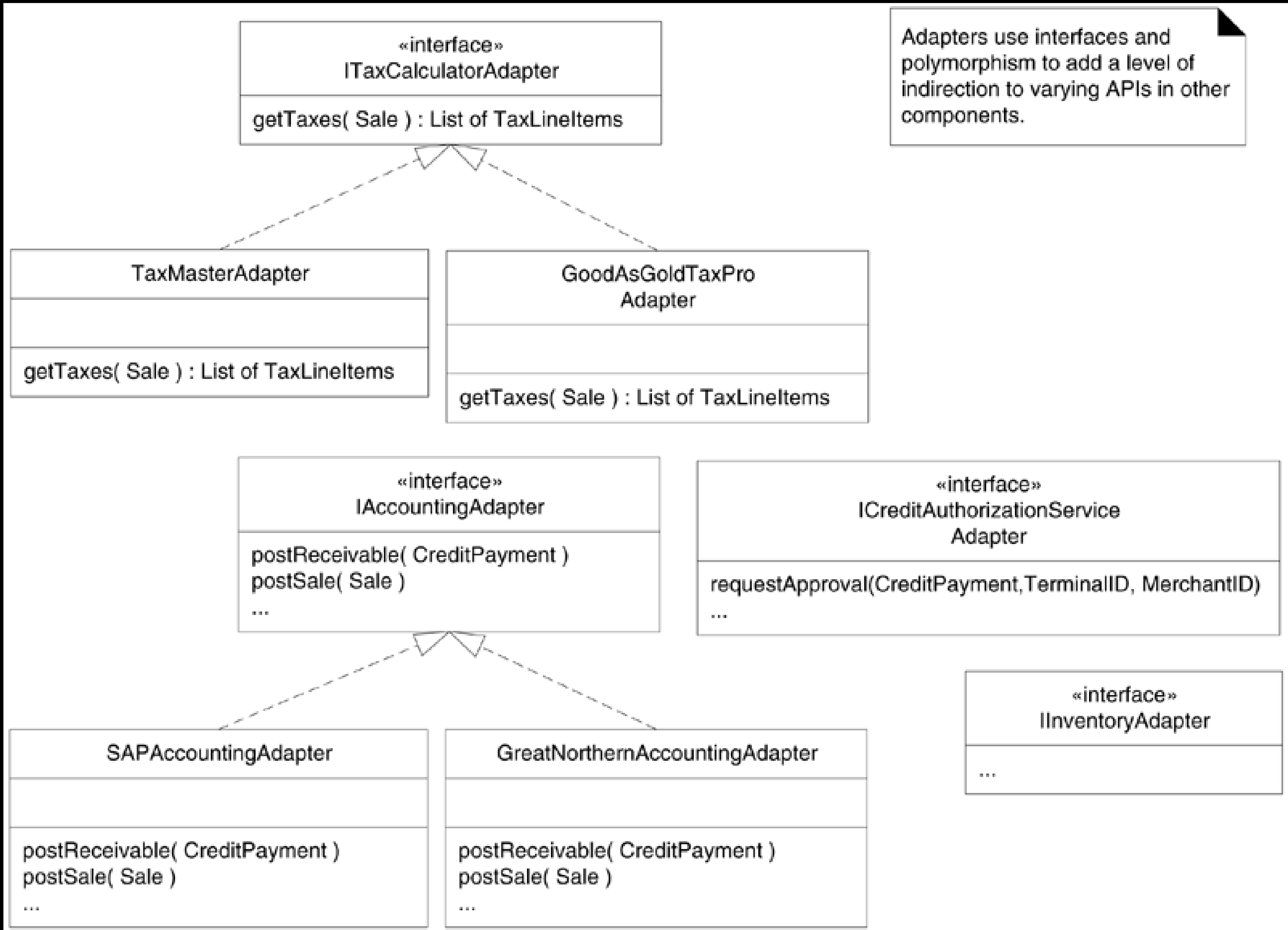
Objective

- Introduce and apply some GoF design patterns.
- Show GRASP principles as a generalization of other design patterns.

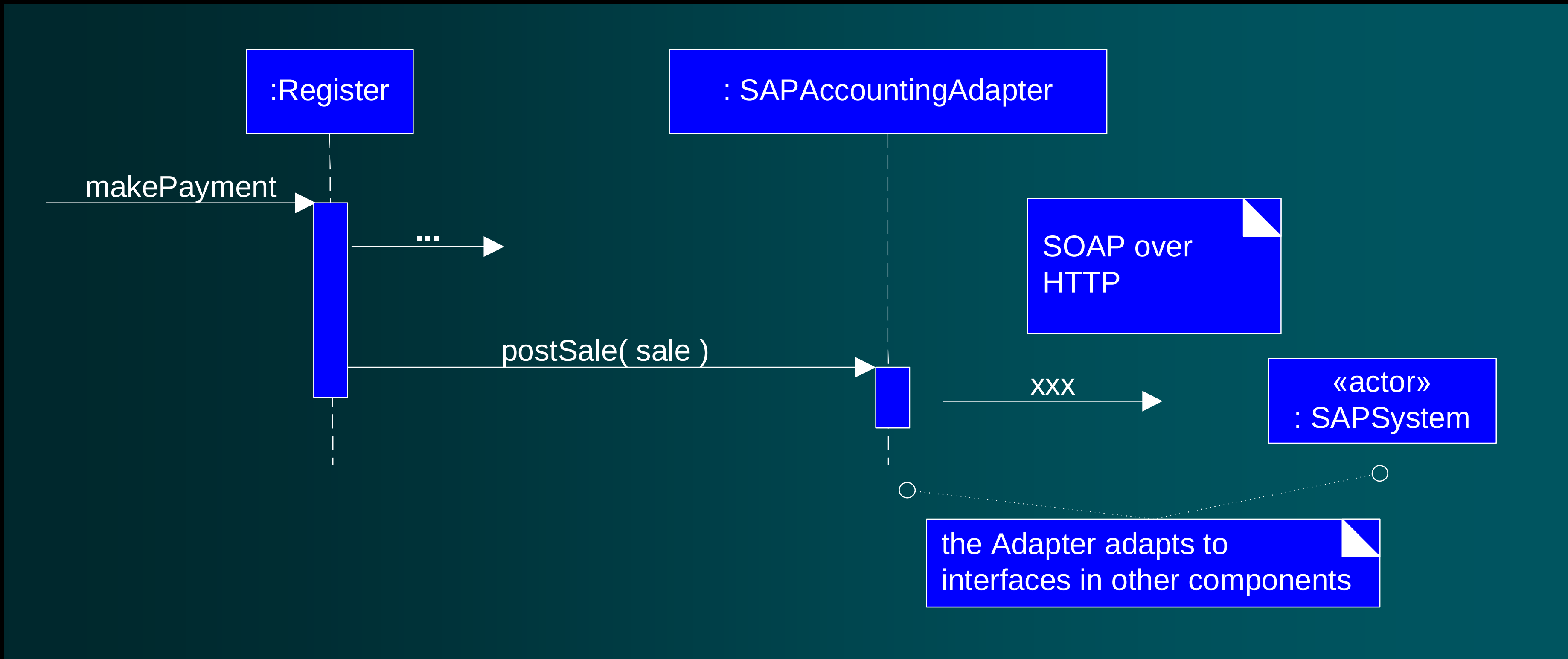
26.1. Adapter (GoF)

- Problem:
  - How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
- Solution:
  - Convert the original interface of a component into another interface, through an intermediate adapter object.

26.1. Adapter (GoF)



26.1. Adapter (GoF)



26.2. Some GRASP Principles as a Generalization of Other Patterns

- What's the Problem? Pattern Overload!
  - The Pattern Almanac 2000 lists around 500 design patterns.
- A Solution: See the Underlying Principles
  - Most design patterns can be seen as specializations of a few basic GRASP principles.

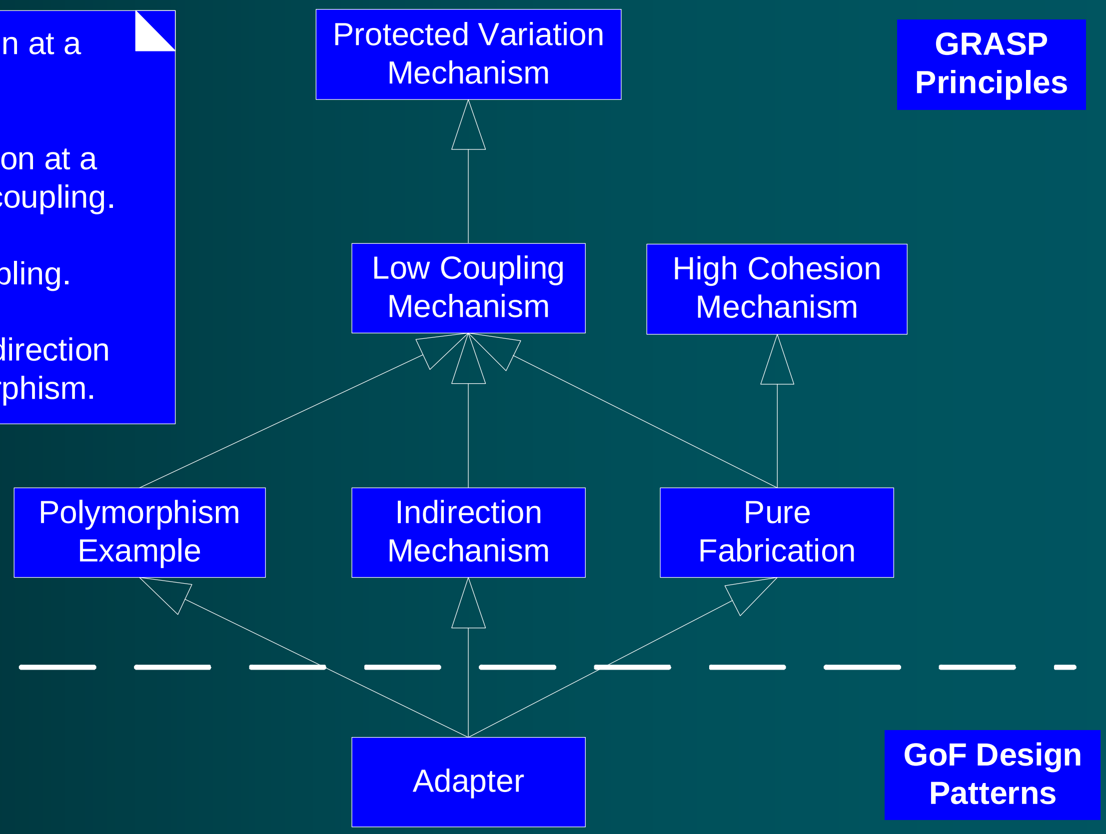
Example: Adapter and GRASP

Low coupling is a way to achieve protection at a variation point.

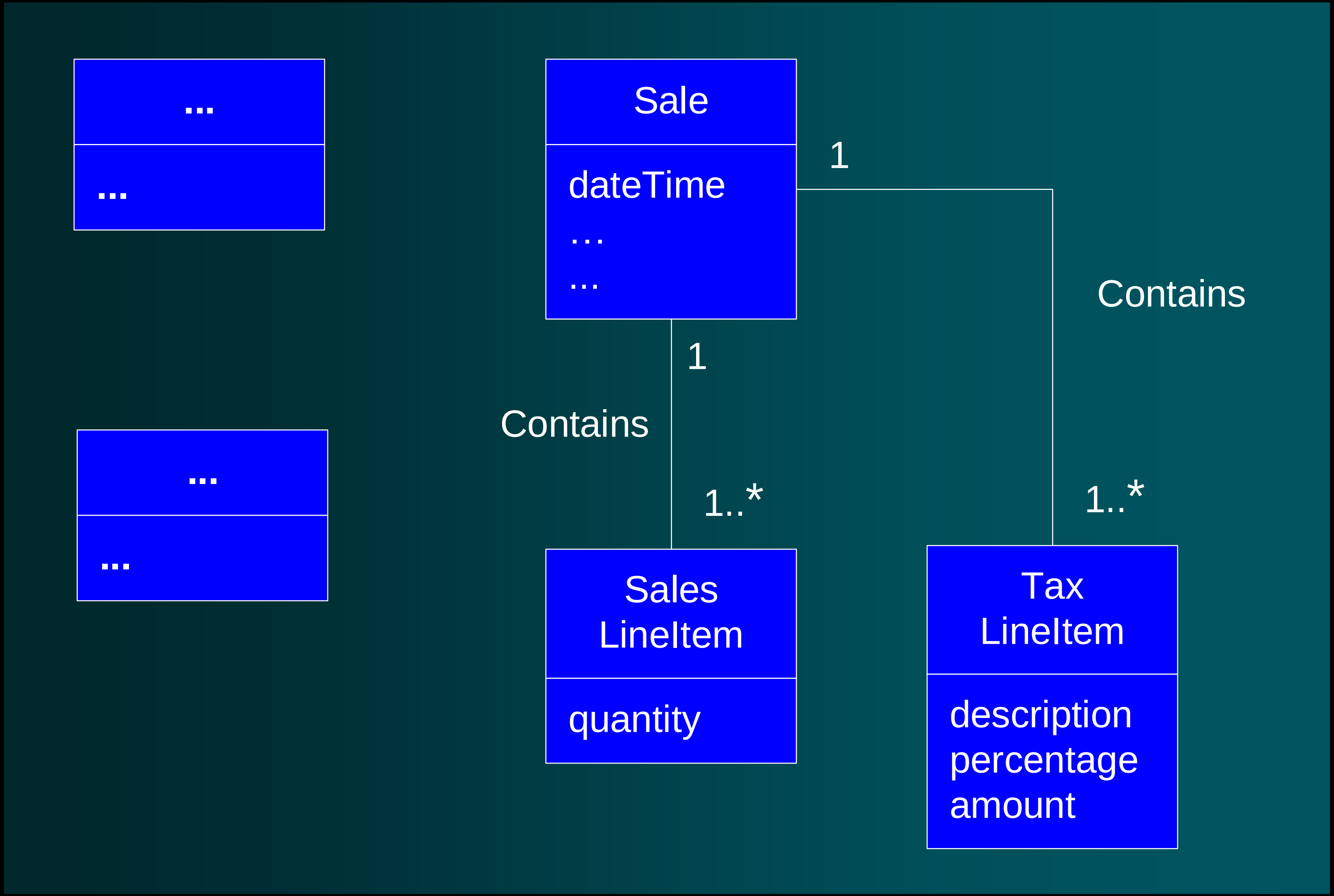
Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



26.3. "Analysis" Discoveries During Design: Domain Model



26.4. Factory

- Also called Simple Factory or Concrete Factory
  - A simplification of the GoF Abstract Factory pattern
  - Problems
    - Who creates the adapters in NextGenPos
    - Choosing a domain object (such as a Register) to create the adapters
      - Does not support the goal of a separation of concerns, and lowers its cohesion.

26.4. Factory

- Factory objects have several advantages:
  - Separate the responsibility of complex creation into cohesive helper objects.
  - Hide potentially complex creation logic.
  - Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

26.4. Factory

- Problem:
  - Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?
- Solution:
  - Create a Pure Fabrication object called a Factory that handles the creation.

26.4. Factory

ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

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

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

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

}
return taxCalculatorAdapter;
```

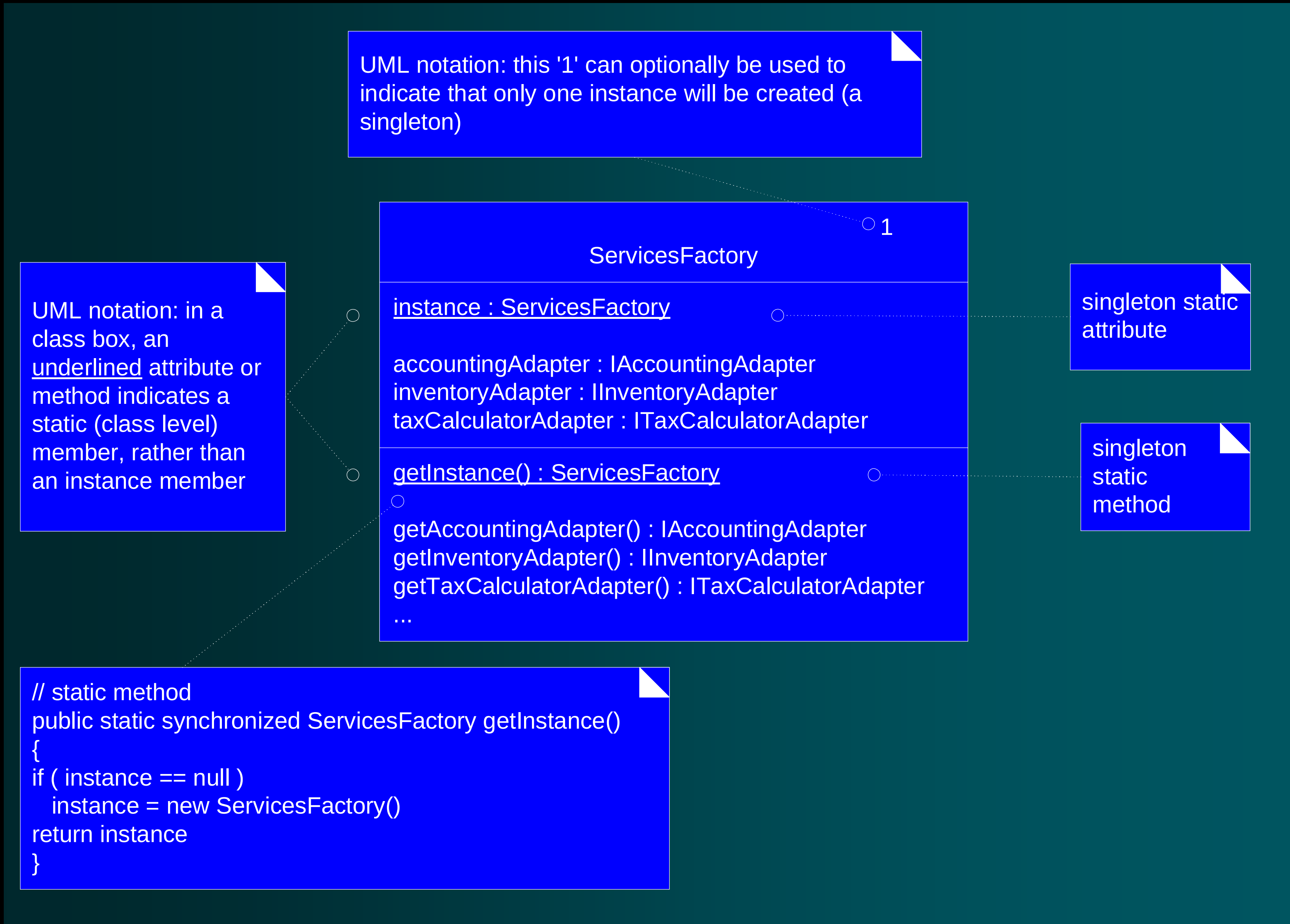
26.5. Singleton (GoF)

- Who creates the factory itself, and how is it accessed?
  - Only one instance of the factory is needed within the process.
  - The methods of this factory may need to be called from various places in the code,
    - Different places need access to the adapters for calling on the external services.
  - How to get visibility to this single ServicesFactory instance?

26.5. Singleton (GoF)

- Problem:
  - Exactly one instance of a class is allowed(singleton). Objects need a global and single point of access.
- Solution:
  - Define a static method of the class that returns the singleton.

26.5. Singleton (GoF)



Implementation and Design Issues

- Lazy initialization

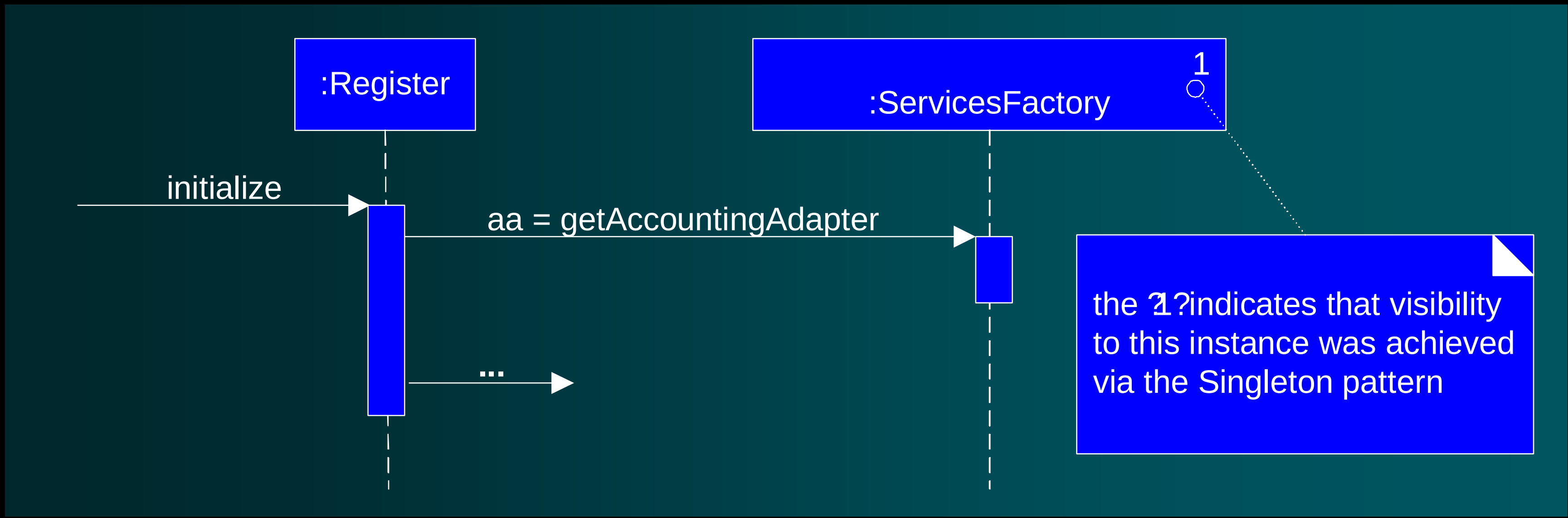
```
public static synchronized ServicesFactory getInstance() {
    if ( instance == null ) {
        // critical section  if multithreaded application
        instance = new ServicesFactory();
    }
    return instance;
}
```

26.5. Singleton (GoF)

§ Eager initialization

```
public class ServicesFactory {
    // eager initialization
    private static ServicesFactory instance =
                                        new ServicesFactory();
    public static ServicesFactory getInstance() {
        return instance;
    }
    // other methods...
}
```

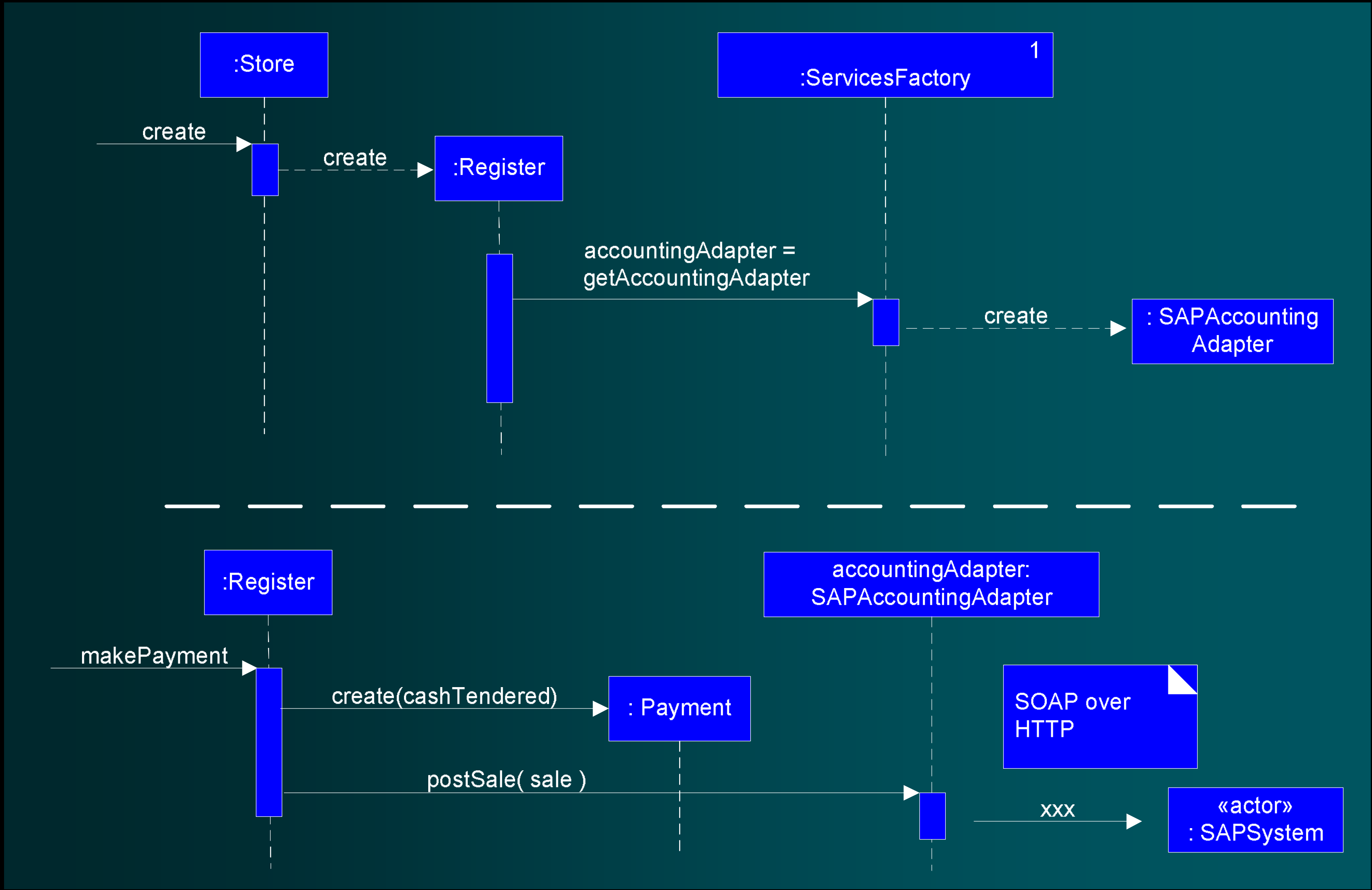
26.5. Singleton (GoF)



26.5. Singleton (GoF)

- Why not make all the service methods static methods of the class itself
  - Instance-side methods permit subclassing and refinement of the singleton class into subclasses;
  - Most object-oriented remote communication mechanisms (for example, Java's RMI) only support remote-enabling of instance methods.
  - The instance-side solution offers flexibility.
    - A class is not always a singleton in all application contexts.
    - It is common to start off a design thinking the object will be a singleton, and then discovering a need for multiple instances in the same process.

26.6. Conclusion of the External Services with Varying Interfaces Problem



To handle the problem of varying interfaces for external services, let's use Adapters generated from a Singleton Factory.

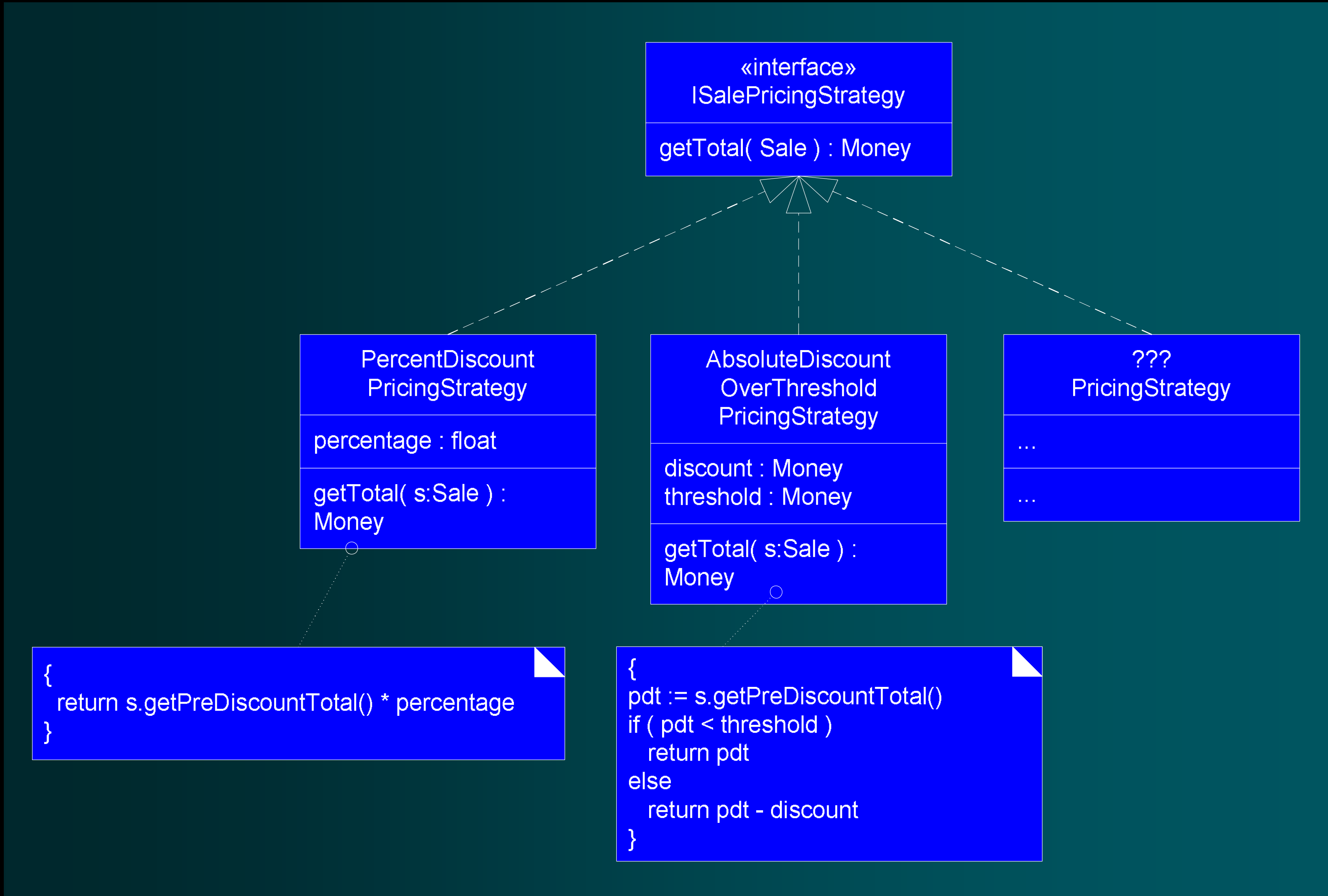
26.7. Strategy (GoF)

- How to provide more complex pricing logic?
  - Such as a store-wide discount for the day, senior citizen discounts, and so forth.
- The pricing strategy for a sale can vary.
  - During one period it may be 10% off all sales, later it may be $10 off if the sale total is greater than $200, and myriad other variations.
  - How do we design for these varying pricing algorithms?

26.7. Strategy (GoF)

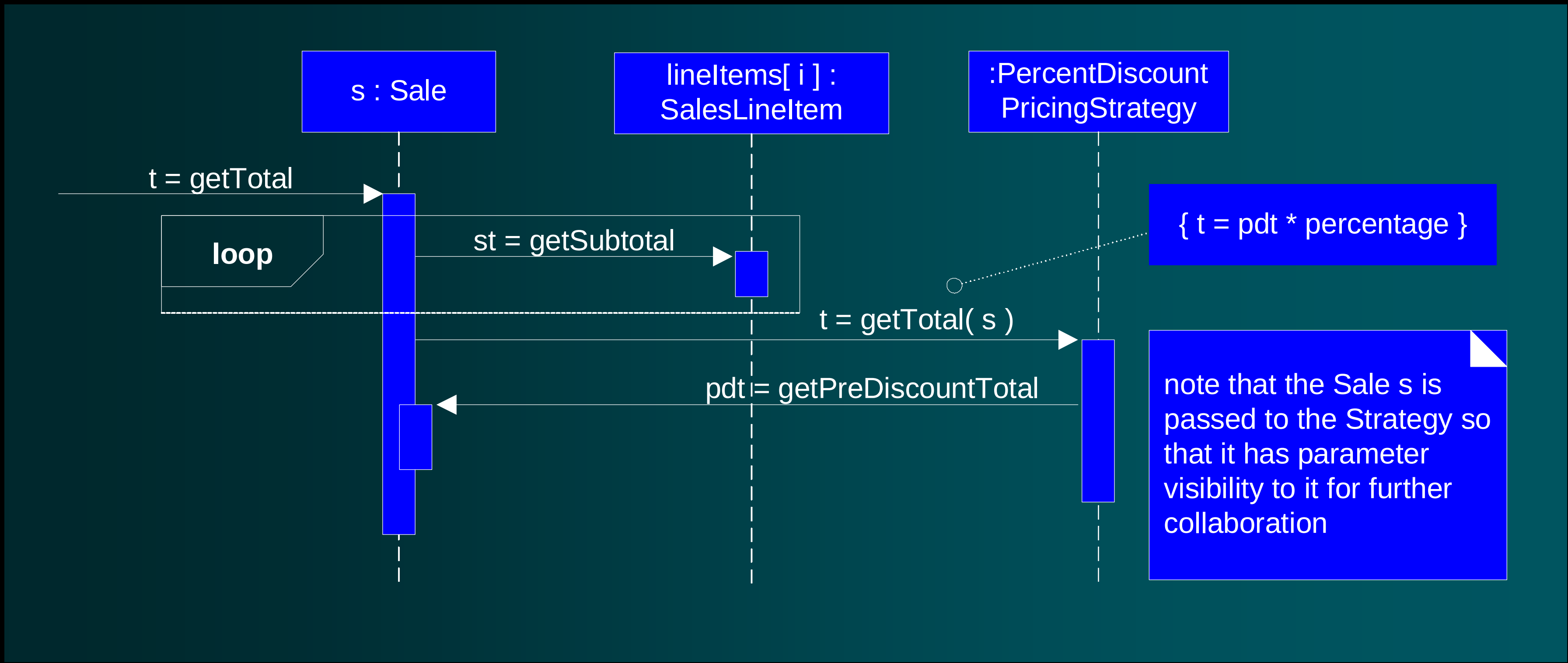
- Problem:
  - How to design for varying, but related, algorithms or policies?
  - How to design for the ability to change these algorithms or policies?
- Solution:
  - Define each algorithm/policy/strategy in a separate class, with a common interface

26.7. Strategy (GoF)

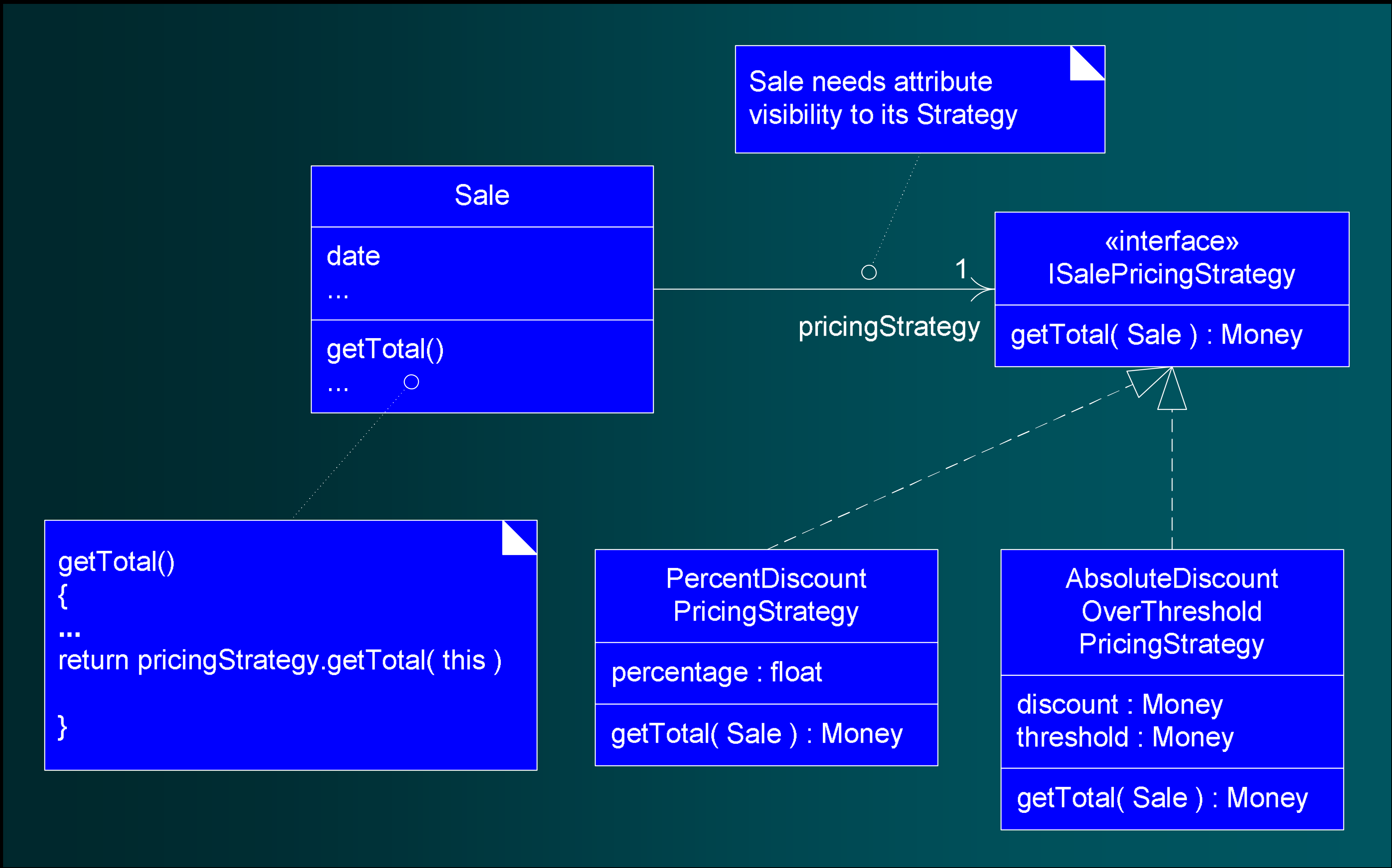


26.7. Strategy (GoF)

A strategy object is attached to a context object.

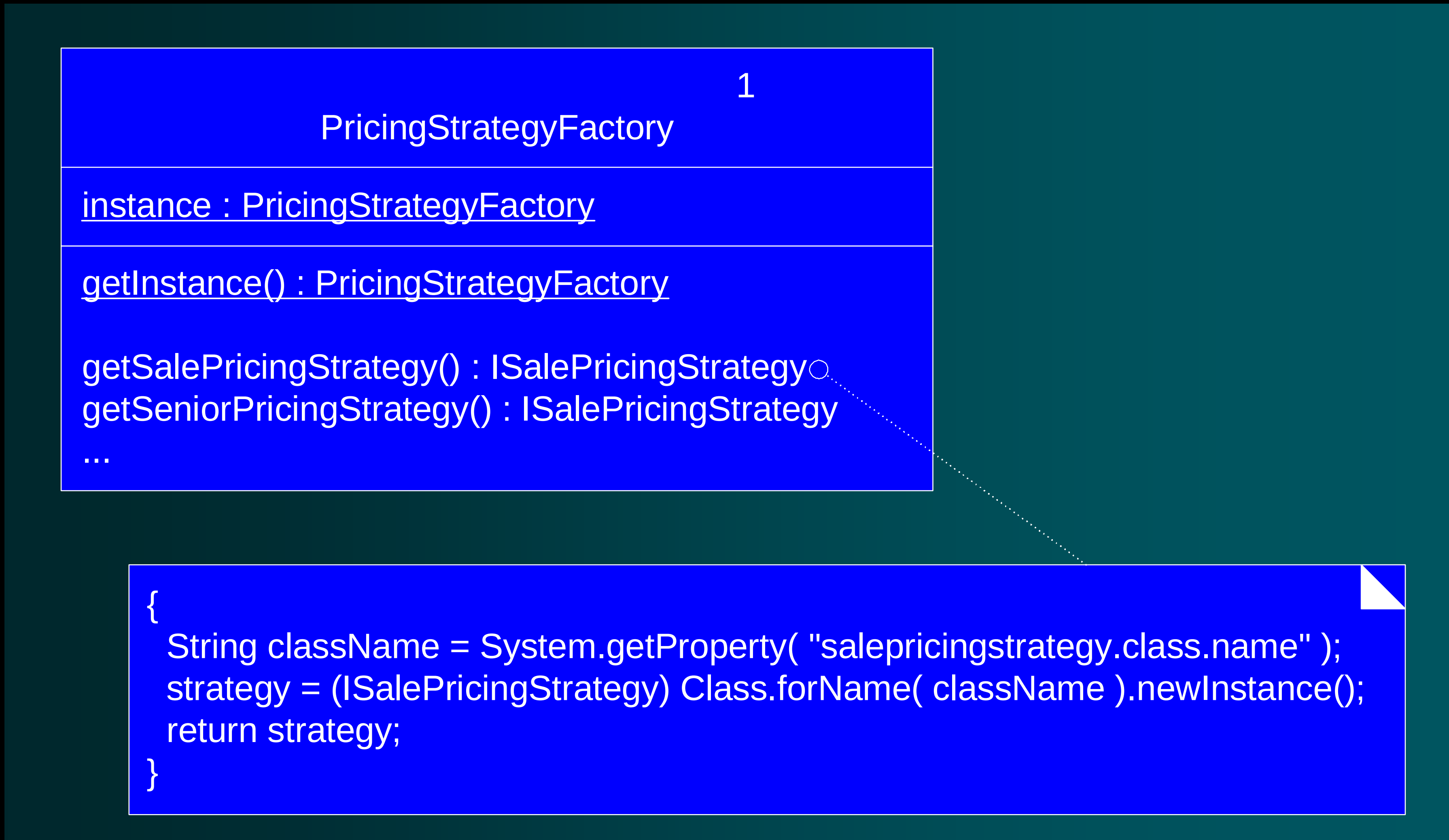


26.7. Strategy (GoF)

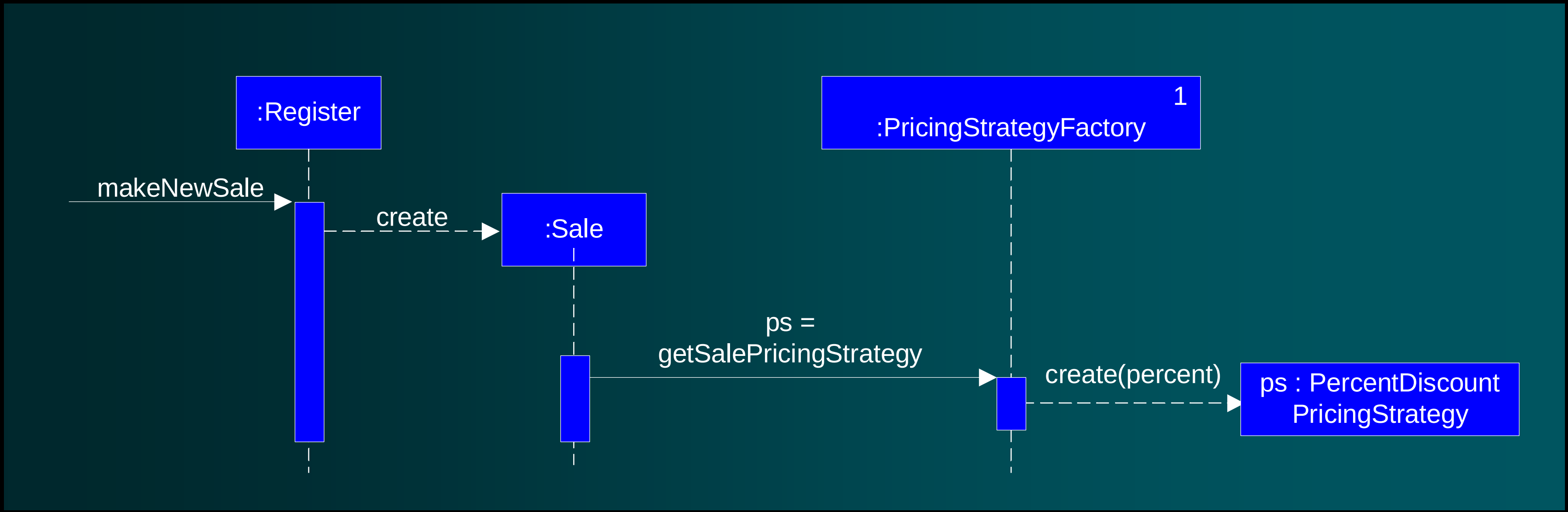


26.7. Strategy (GoF)

w Creating a Strategy with a Factory



26.7. Strategy (GoF)



26.7. Strategy (GoF)

- **Reading and Initializing the Percentage Value**
  - How to find the different numbers for the percentage or absolute discounts.
  - These numbers will be stored in some external data store.
  - StrategyFactory read them and assign them to the strategy.
- **Summary**
  - Protected Variations with respect to dynamically changing pricing policies has been achieved with the Strategy and Factory patterns.
    - Strategy builds on Polymorphism and interfaces to allow pluggable algorithms in an object design.

26.8. Composite (GoF) and Other Design Principles

- How do we handle the case of multiple, conflicting pricing policies?
  - For example, suppose a store has the following policies in effect today (Monday):
    - 20% senior discount policy
    - Preferred customer discount of 15% off sales over $400
    - On Monday, there is $50 off purchases over $500
    - Buy 1 case of Darjeeling tea, get 15% discount off of everything

26.8. Composite (GoF) and Other Design Principles

- Suppose a senior who is also a preferred customer buys
  - 1 case of Darjeeling tea,
  - $600 of veggieburgers.
- What pricing policy should be applied?

26.8. Composite (GoF) and Other Design Principles

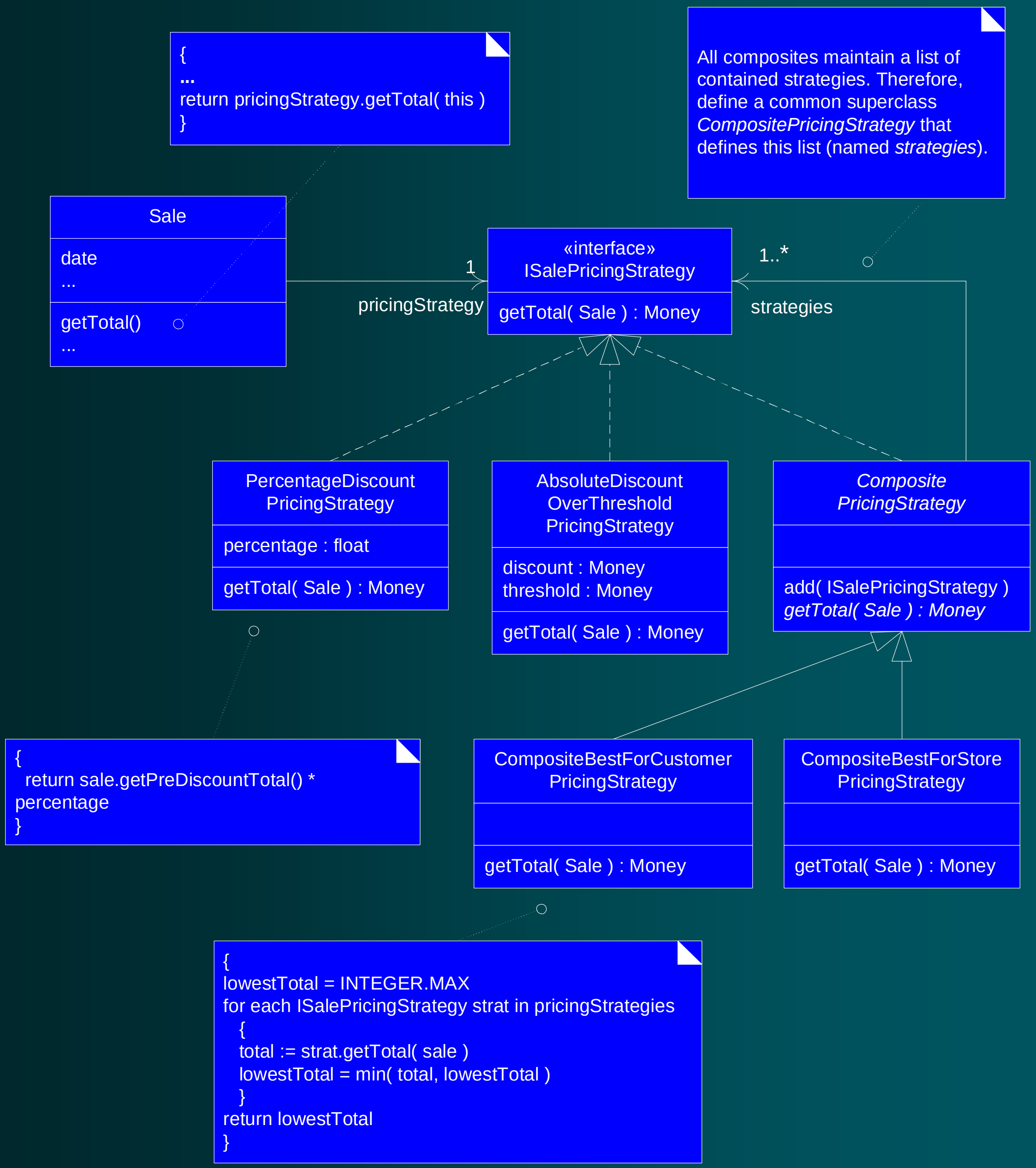
- Pricing strategies by virtue of three factors:
  - Time Period (Monday)
  - Customer Type (senior)
  - a particular line item product (Darjeeling tea)
- Three of the four example policies are really just "percentage discount" strategies

26.8. Composite (GoF) and Other Design Principles

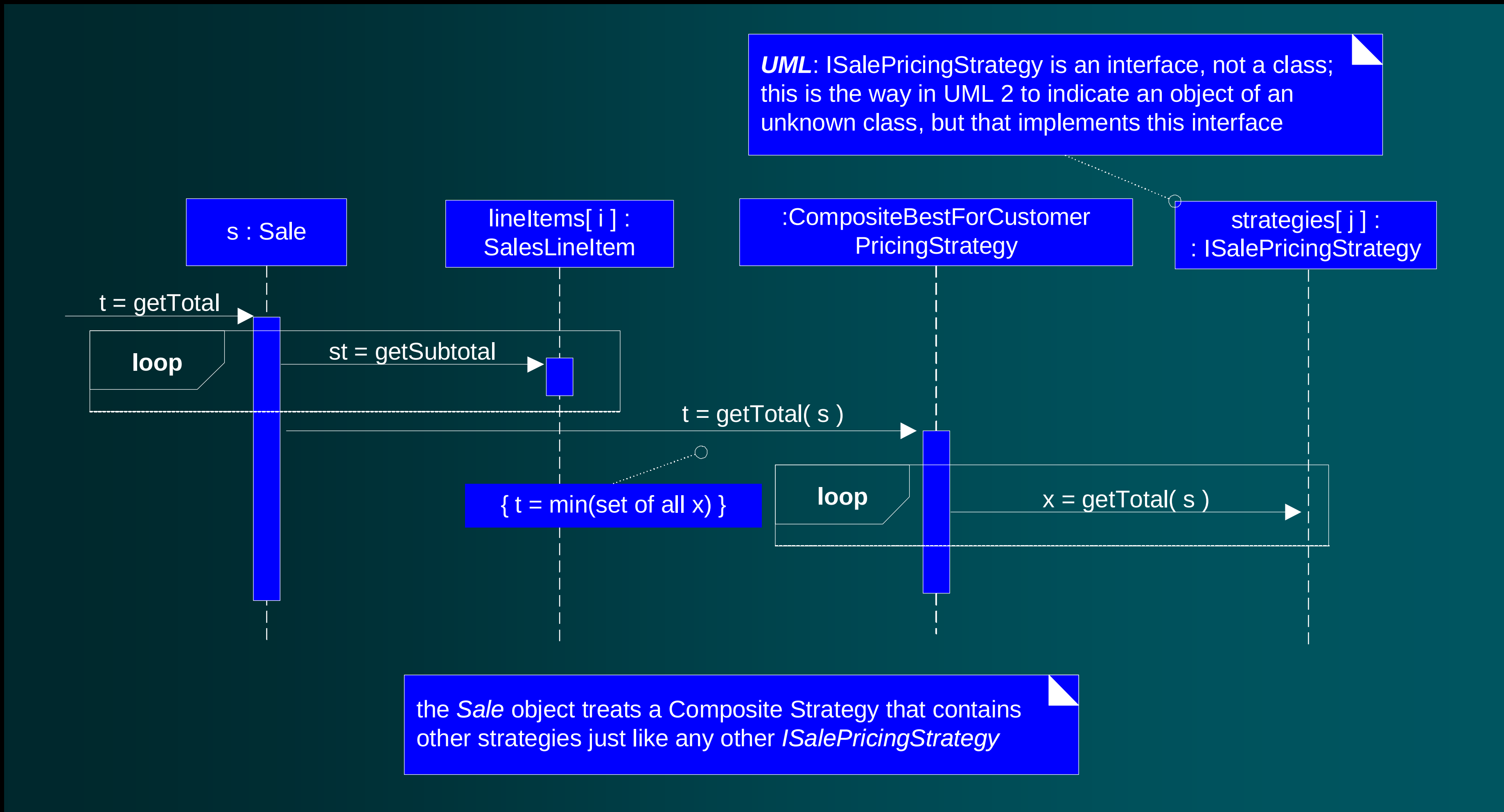
- How to define the store's conflict resolution strategy
  - Usually, a store applies the lowest price conflict resolution strategy
  - But the store may have to change to the highest price conflict resolution strategy

26.8. Composite (GoF) and Other Design Principles

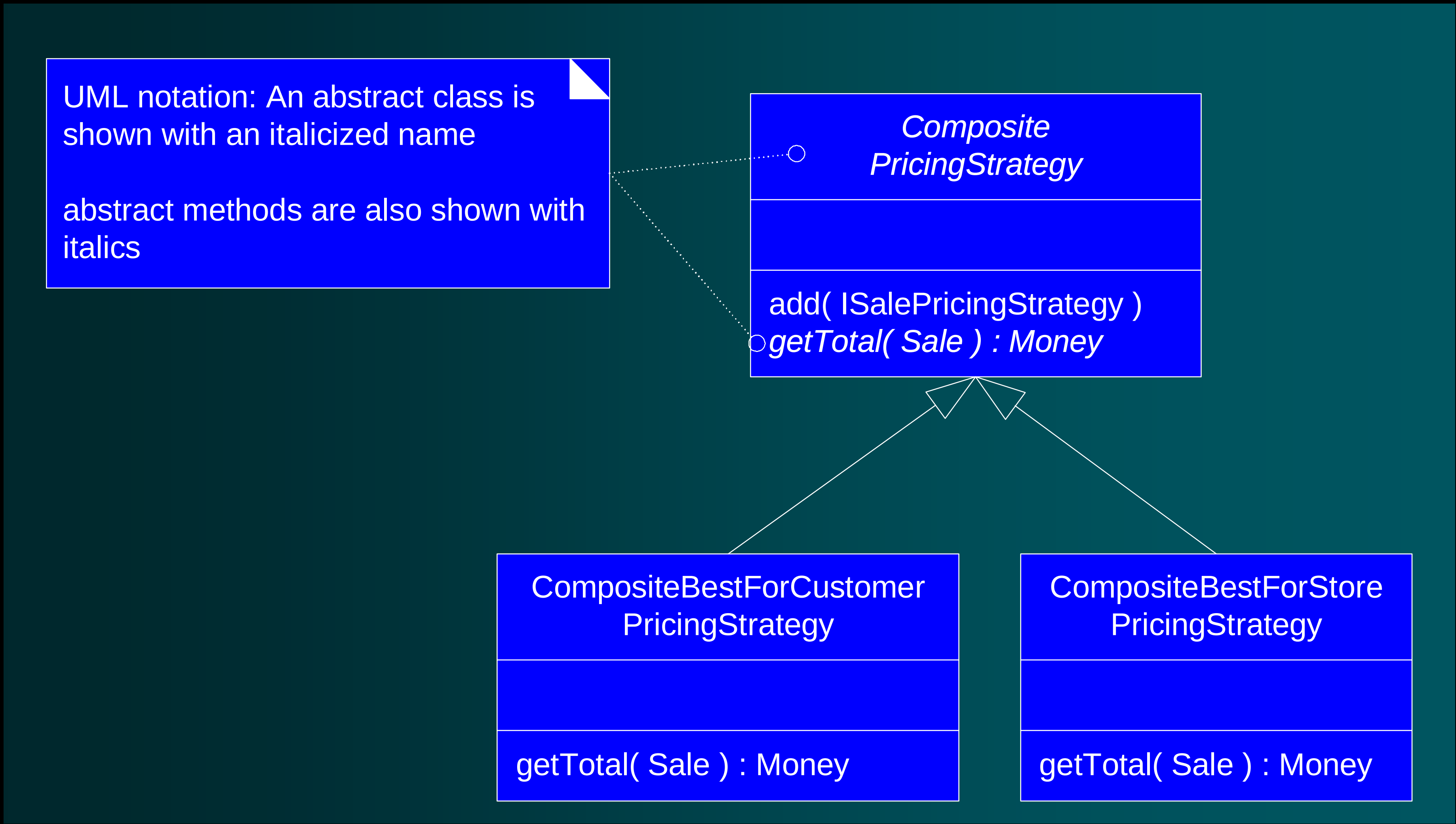
- Problem:
  - How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?
- Solution:
  - Define classes for composite and atomic objects so that they implement the same interface.


{
...
return pricingStrategy.getTotal(this)
}
All composites maintain a list of contained strategies. Therefore, define a common superclass *CompositePricingStrategy* that defines this list (named *strategies*).
Sale
date
...
getTotal()
...
«interface»
ISalePricingStrategy
getTotal(Sale) : Money
1
pricingStrategy
1..*
strategies
PercentageDiscount PricingStrategy
percentage : float
getTotal(Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(Sale) : Money
Composite PricingStrategy
add(ISalePricingStrategy)
getTotal(Sale) : Money
{
return sale.getPreDiscountTotal() * percentage
}
CompositeBestForCustomer PricingStrategy
getTotal(Sale) : Money
CompositeBestForStore PricingStrategy
getTotal(Sale) : Money
{
lowestTotal = INTEGER.MAX
for each ISalePricingStrategy strat in pricingStrategies
{
total := strat.getTotal(sale)
lowestTotal = min(total, lowestTotal)
}
return lowestTotal
}

26.8. Composite (GoF) and Other Design Principles



26.8. Composite (GoF) and Other Design Principles

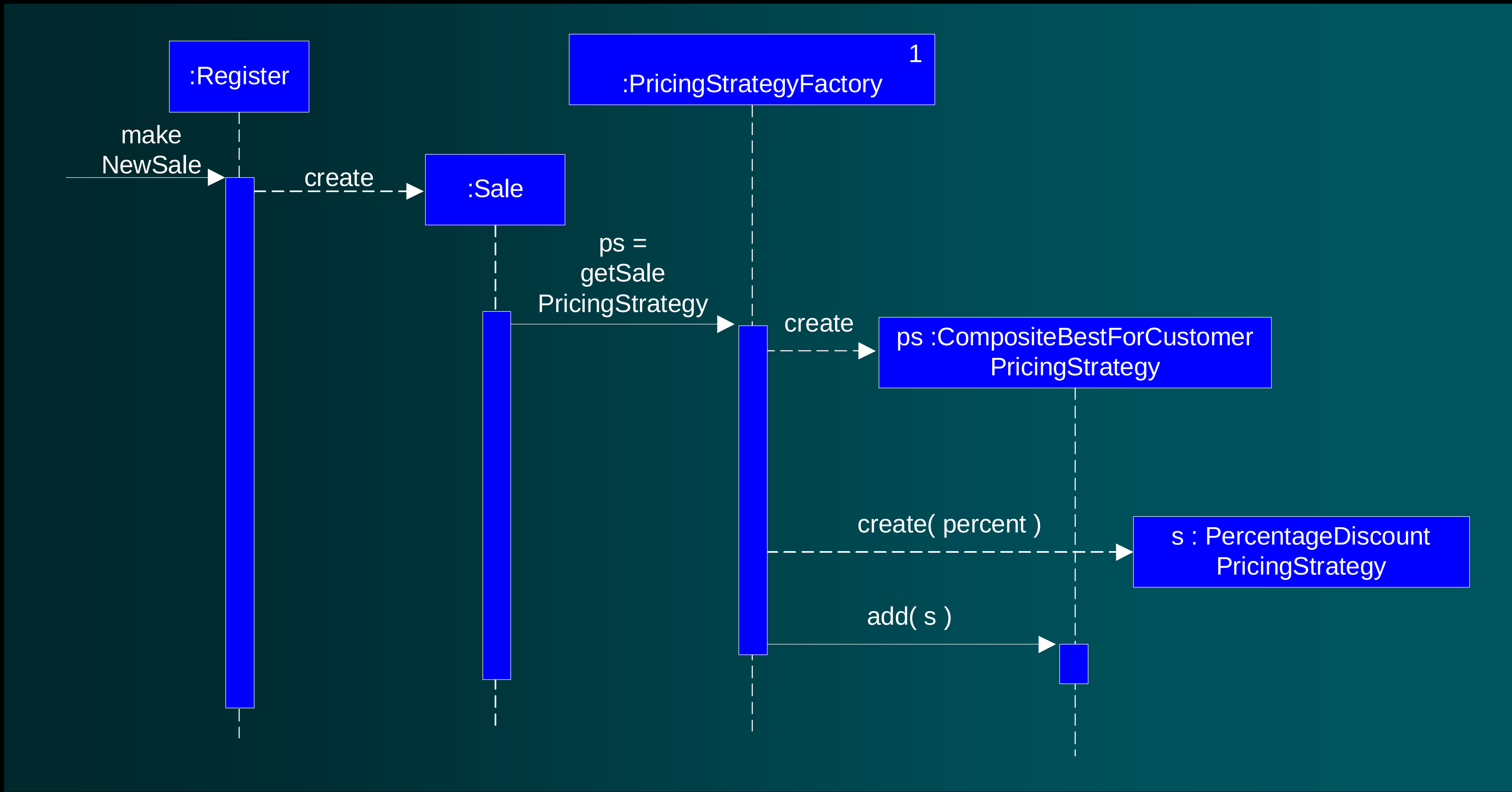


26.8. Composite (GoF) and Other Design Principles

- Creating Multiple SalePricingStrategies
  - When do we create these strategies?
    - Three points in the scenario where pricing strategies may be added to the composite:
      - Current store-defined discount, added when the sale is created.
      - Customer type discount, added when the customer type is communicated to the POS.
      - Product type discount, added when the product is entered to the sale.

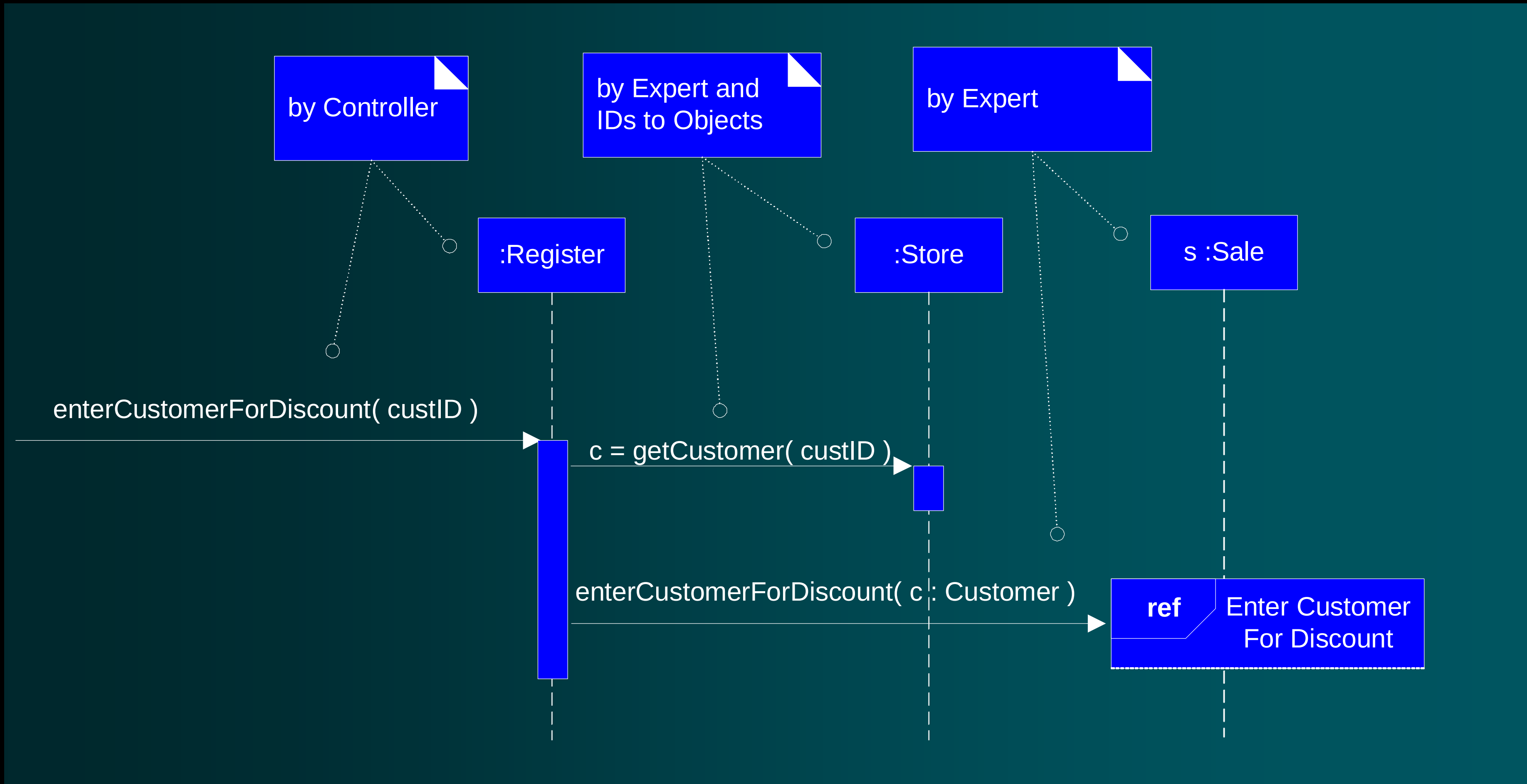
26.8. Composite (GoF) and Other Design Principles

- The design of the first case

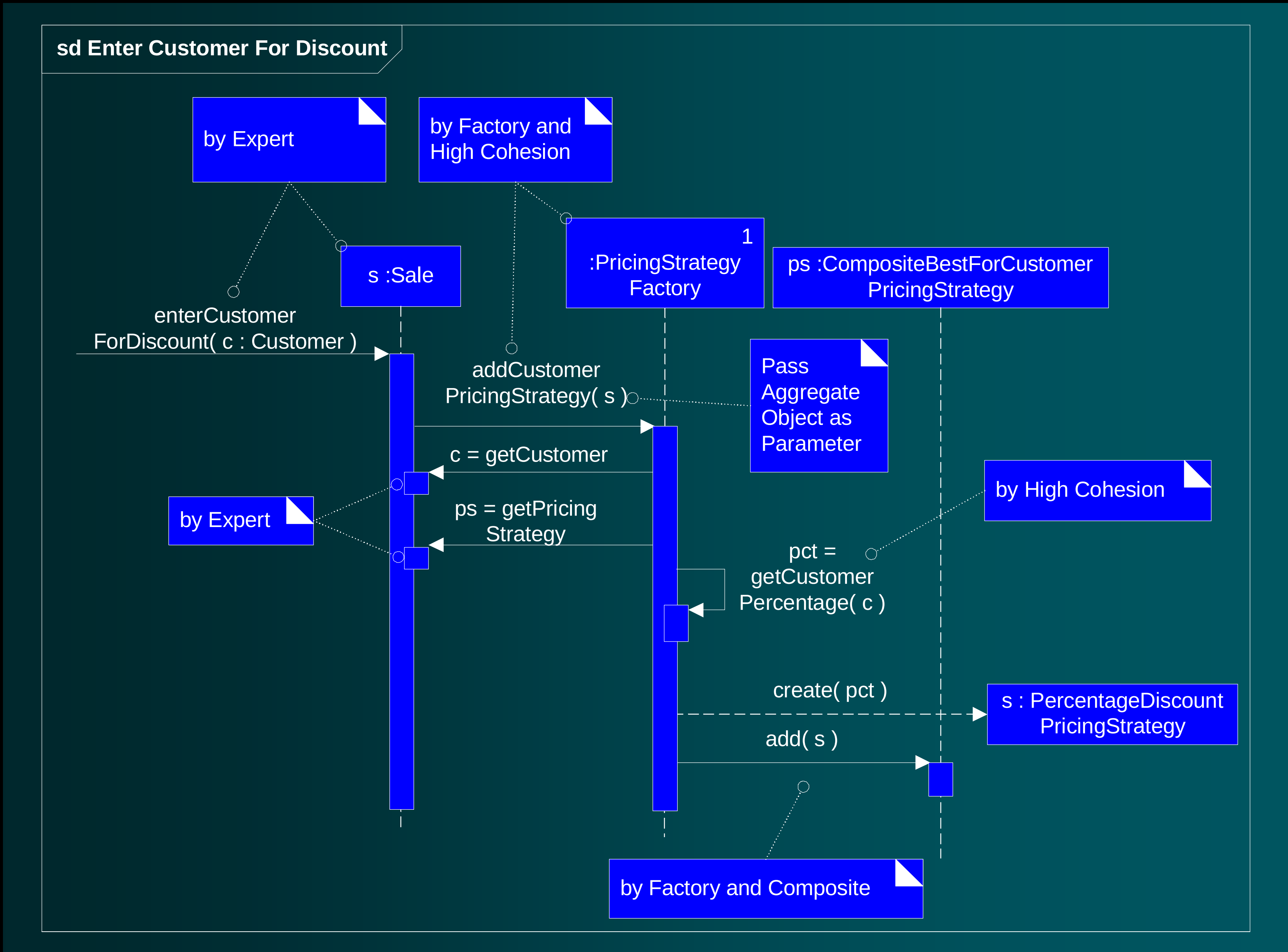


26.8. Composite (GoF) and Other Design Principles

- The 2nd case of a customer type discount



26.8. Composite (GoF) and Other Design Principles



26.8. Composite (GoF) and Other Design Principles

- Considering GRASP and Other Principles in the Design
  - Why not have the Register send a message to the PricingStrategyFactory,
    - to create this new pricing strategy and then pass it to the Sale?
  - One reason is to support Low Coupling
    - The Sale is already coupled to the factory
  - Furthermore, the Sale knows its current pricing strategy;

26.8. Composite (GoF) and Other Design Principles

- Why customerID is transformed into a Customer object via the Register asking the Store for a Customer, given an ID.
  - By Information Expert, the Store can know all the Customers.
  - Register already has attribute visibility to the Store.
  - If the Sale had to ask the Store, the Sale would need a reference to the Store

26.8. Composite (GoF) and Other Design Principles

- IDs to Objects
  - Why transform the customerID into a Customer object?
    - Having a true Customer object becomes beneficial and flexible.
- Pass Aggregate Object as Parameter
  - Why pass a Sale to the factory, not the Customer and PricingStrategy?
    - Avoid extracting child objects out of parent or aggregate objects, and then passing around the child objects.

26.9. Facade (GoF)

- **To support pluggable business rules.**
  - at predictable points in the scenarios, different customers would like to customize its behavior slightly
    - Suppose when a new sale is created, if it will be paid by a gift certificate. Then, only one item is allowed to be purchased.
    - If the sale is paid by a gift certificate, invalidate all payment types of change due back to the customer except for another gift certificate.
    - Suppose when a new sale is created, it is possible to identify that it is for a charitable donation. A store may also have a rule to only allow item entries less than $250 each, and also to only add items to the sale if the currently logged in "cashier" is a manager.

26.9. Facade (GoF)

- **Problem:**
  - A common, unified interface to a disparate set of implementations or interfaces is required.
  - Undesirable coupling to many things in the subsystem.
- **Solution:**
  - Define a single point of contact to the subsystem that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

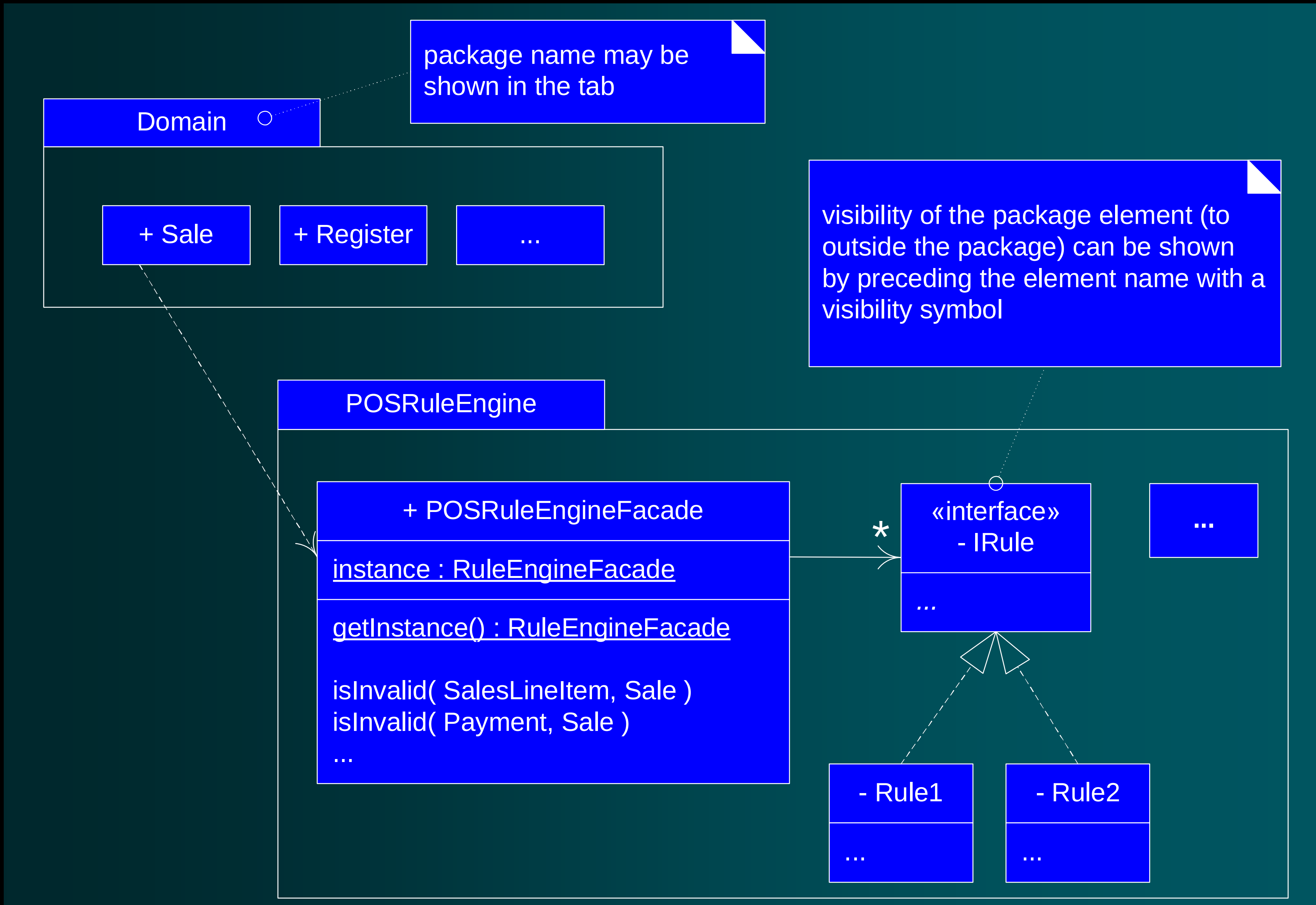
26.9. Facade (GoF)

- Facade
  - A "front-end" object that is the single point of entry for the services of a subsystem.
  - Provides Protected Variations from changes in the implementation of a subsystem.

26.9. Facade (GoF)

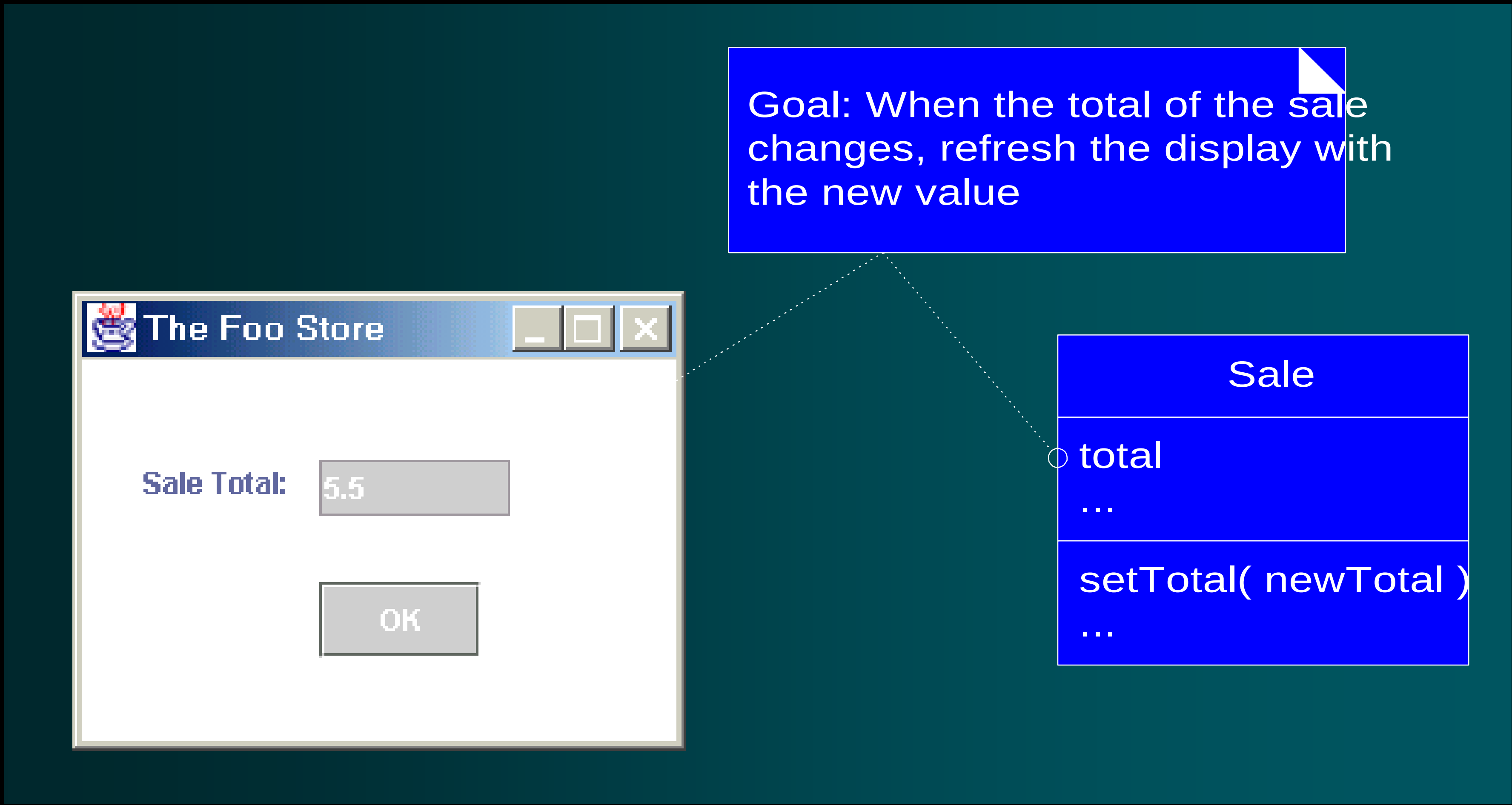
```
public class Sale {
   public void makeLineItem ( ProductDescription desc,
          int quantity ) {
          SalesLineItem sli = new SalesLineItem( desc, quantity );
          // call to the Facade
          if ( POSRuleEngineFacade.getInstance().isInvalid( sli,
                 this ) )
                 return;
          lineItems.add( sli );
   }
   // ...
} // end of class
```

26.9. Facade (GoF)



26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)

- How to refresh GUI's display of the sale total when the total changes

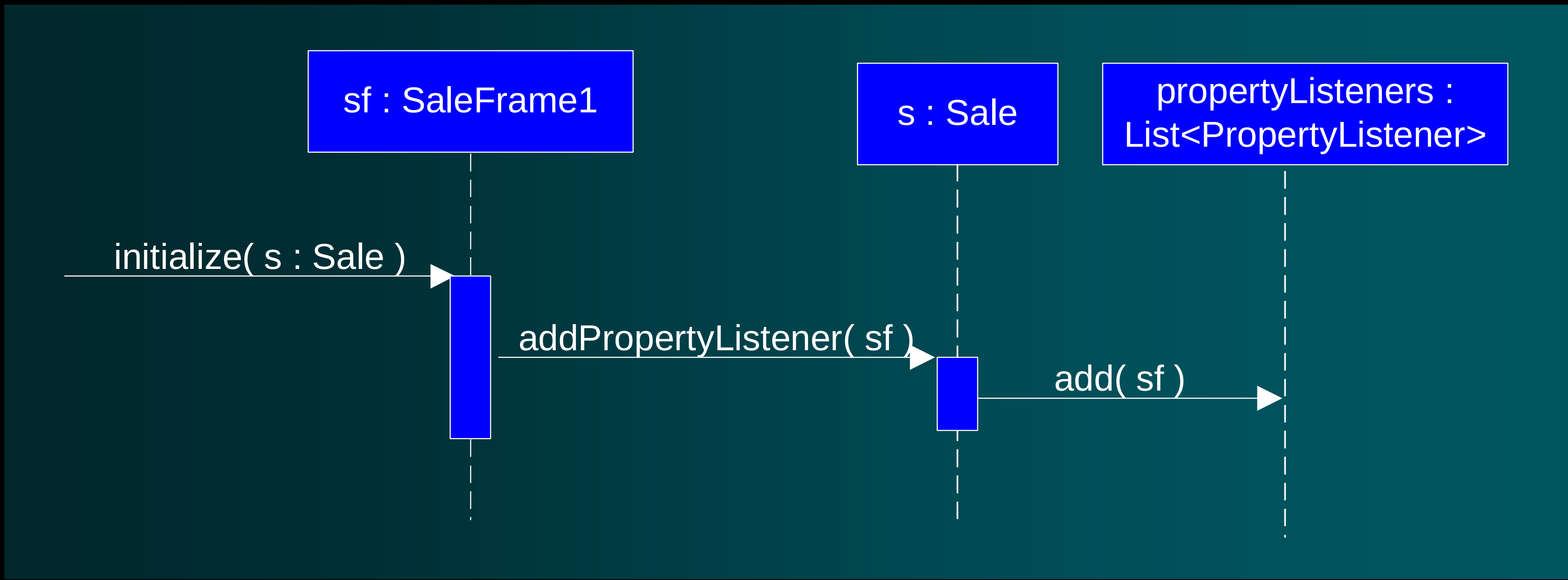


26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)

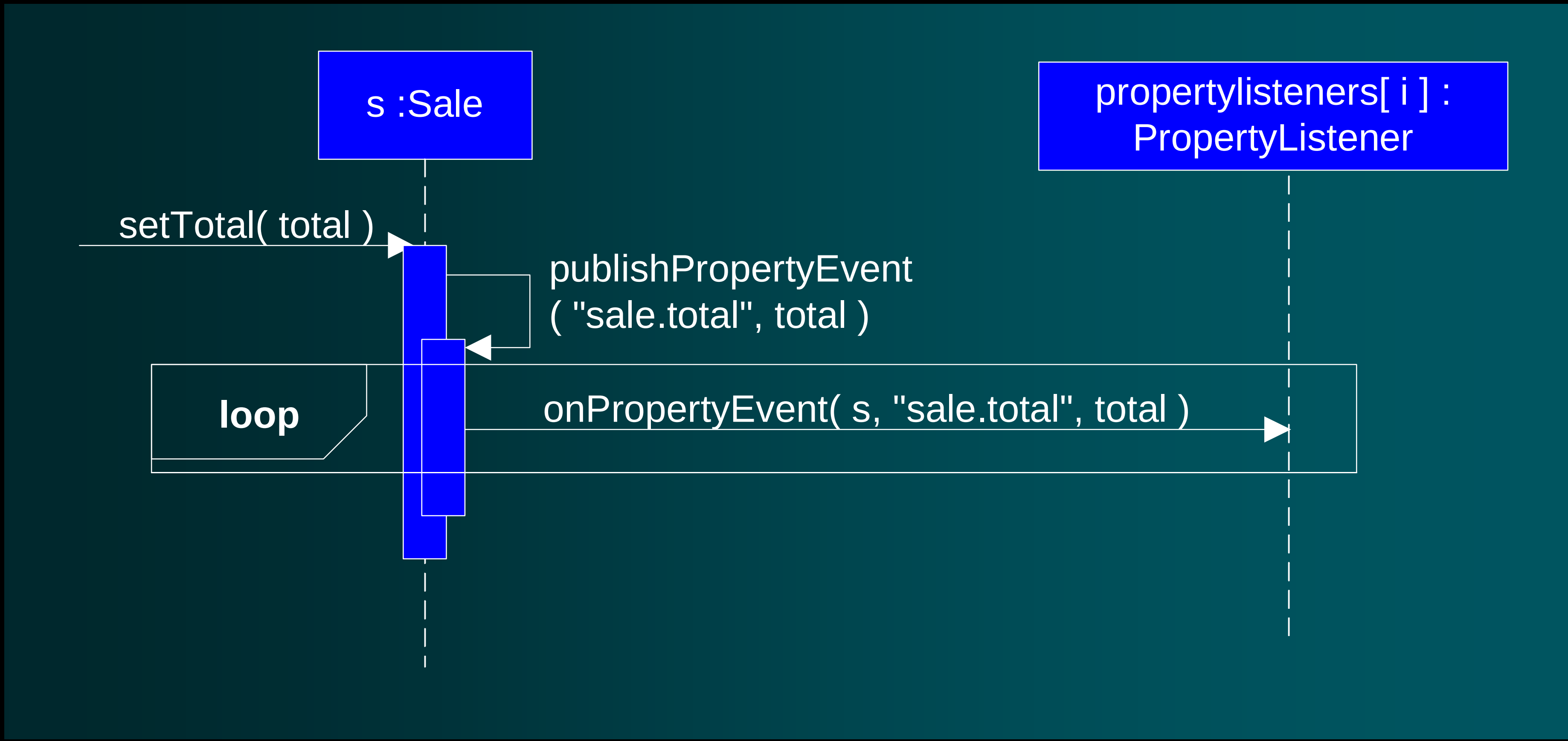
- **Problem:**
  - Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event.
- **Solution:**
  - Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.


{
for each PropertyListener pl in propertyListeners
pl.onPropertyEvent(this, name, value);
}
{
propertyListeners.add(lis);
}
Sale
addPropertyListener(PropertyListener lis)
publishPropertyEvent(name, value)
setTotal(Money newTotal)
...
{
total = newTotal;
publishPropertyEvent("sale.total", total);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
propertyListeners
*
«interface»
PropertyListener
onPropertyEvent(source, name, value)
{
if (name.equals("sale.total"))
saleTextField.setText(value.toString());
}
SaleFrame1
onPropertyEvent(source, name, value)
initialize(Sale sale)
...
{
sale.addPropertyListener(this)
...
}

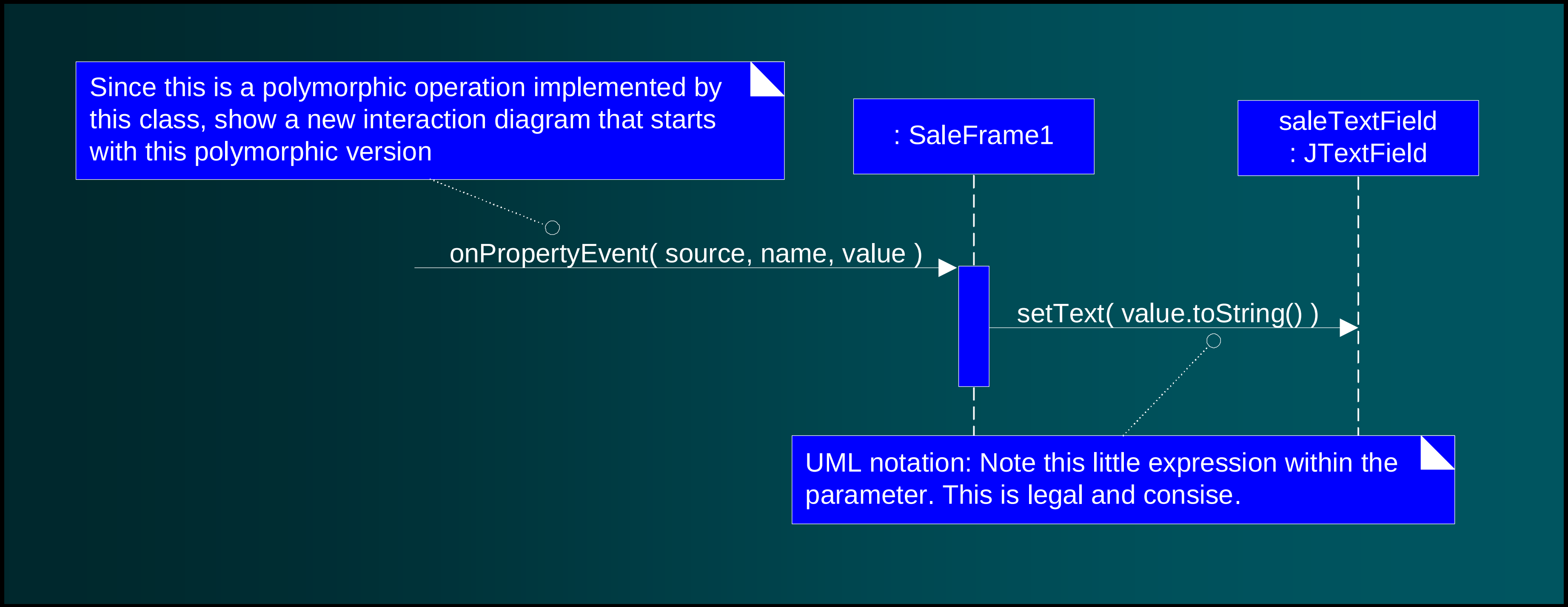
26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)



26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)

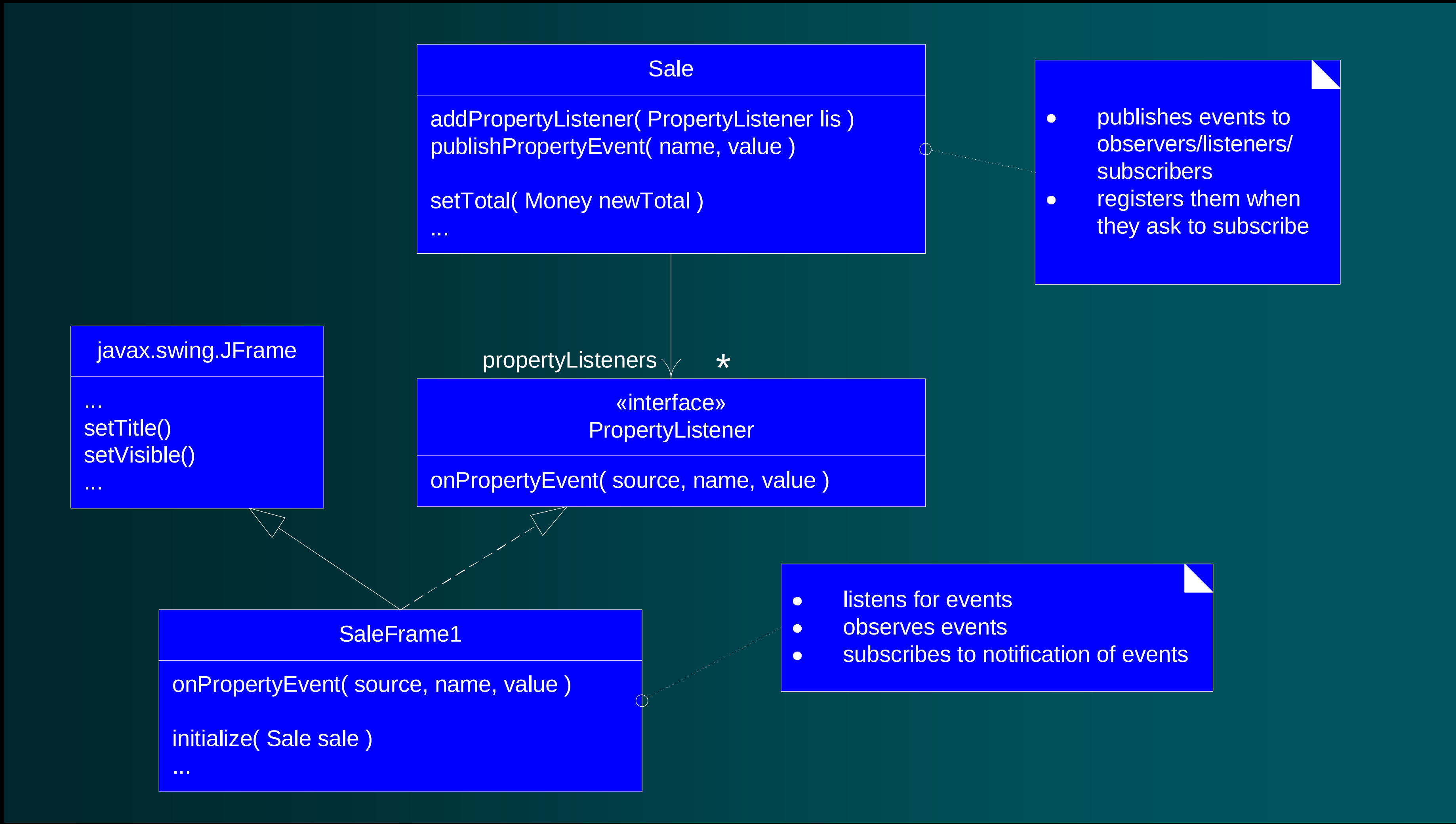


26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)



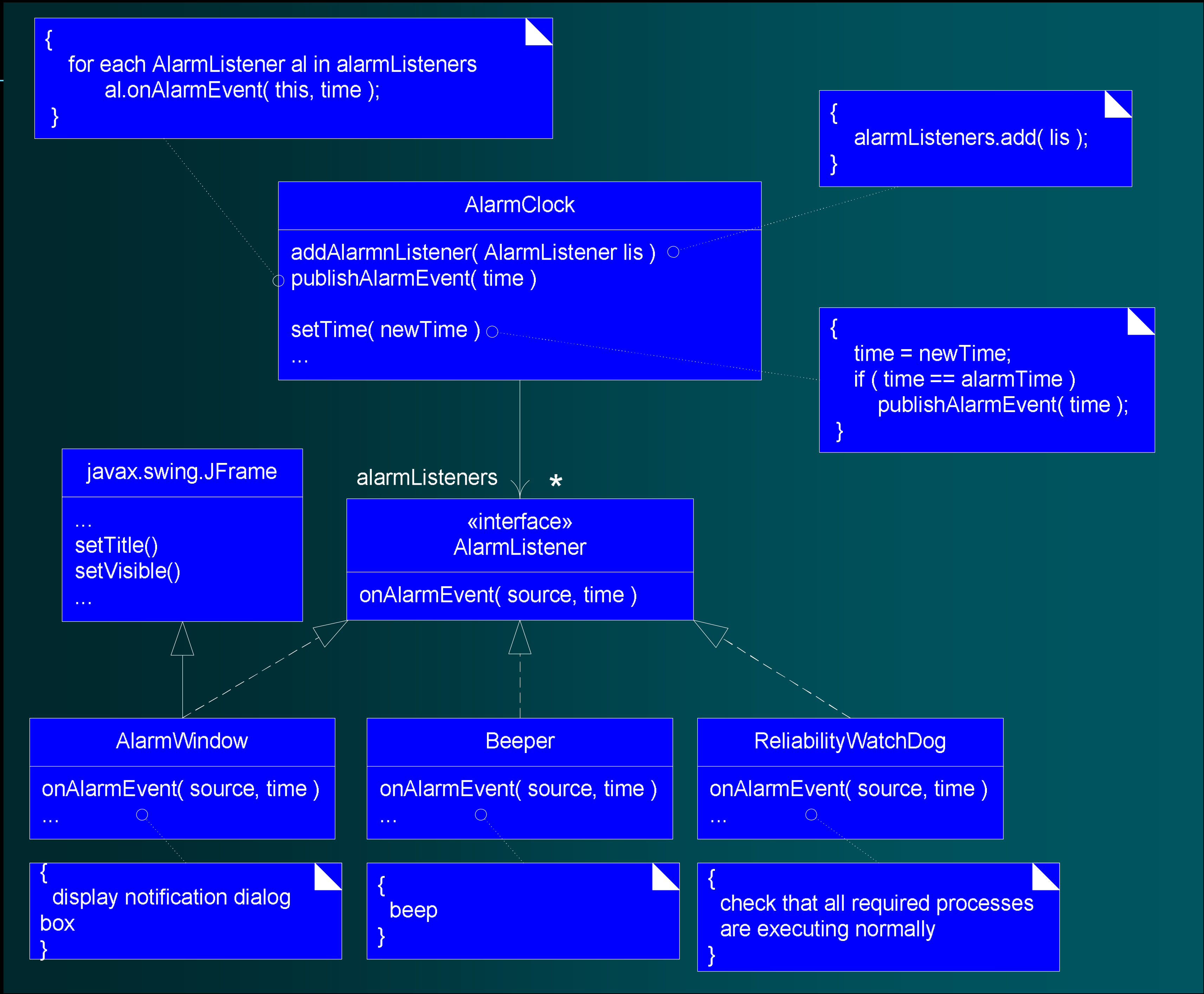
26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)

- **Observer, Publish-Subscribe, or Delegation Event Model?**



Observer Is Not Only for Connecting UIs and Model Objects

- The most prevalent use of this pattern is for GUI widget event handling
  - JButton publishes an "action event" when it is pressed.
  - Another object will register with the button so that when it is pressed
- Another example
  - AlarmClock


{
for each AlarmListener al in alarmListeners
al.onAlarmEvent(this, time);
}
{
alarmListeners.add(lis);
}
AlarmClock
addAlarmnListener(AlarmListener lis)
publishAlarmEvent(time)
setTime(newTime)
...
{
time = newTime;
if (time == alarmTime)
publishAlarmEvent(time);
}
javax.swing.JFrame
...
setTitle()
setVisible()
...
alarmListeners
*
«interface»
AlarmListener
onAlarmEvent(source, time)
AlarmWindow
onAlarmEvent(source, time)
...
Beeper
onAlarmEvent(source, time)
...
ReliabilityWatchDog
onAlarmEvent(source, time)
...
{
display notification dialog
box
}
{
beep
}
{
check that all required processes
are executing normally
}

26.10. Observer/Publish-Subscribe/Delegation Event Model (GoF)

- **Implementation**

```
class PropertyEvent extends Event {
    private Object sourceOfEvent;
    private String propertyName;
    private Object oldValue;
    private Object newValue;
    //...
}
//...
class Sale {
    private void publishPropertyEvent( String name, Object old, Object new )
    {
            PropertyEvent evt = new PropertyEvent( this,
                        "sale.total", old, new);
            for each AlarmListener al in alarmListeners
                        al.onPropertyEvent( evt );
     }
    //...
 }
```