

GoF Design Patterns (contd)

The Strategy Pattern (Behavioral)

The behavior of a class may change during the lifetime (runtime) of an object of this class.

Example:

Such a design problem in the example POS system is the complex pricing policy.

The pricing strategy (which may also be called a rule, policy, or algorithm) for sale can vary.

For example,

- Mondays, it may be 10%, and Thursdays, 5% off all sales (percentage)
- It may be 10TL off, if the sale total is greater than 200TL (absolute),
- For customers with a loyalty card, there may be other discounts (customer-based).

All these different algorithms (pricing strategies) seem to be variations of the getTotal() responsibility (behavior) of the Sale class.

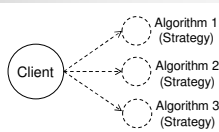
However, adding all these algorithms into the getTotal() method of the Sale using if-then-else or switch-case statements will cause coupling and cohesion problems.

All changes in pricing strategies will affect the Sale.

<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

9.1



Definition: Strategy

Problem:

How should we design for varying but related algorithms or policies?

How should we design for the ability to change these algorithms or policies?

(A certain behavior of a class may change during the lifetime of an object of this class.)

Solution:

Define each algorithm/policy/strategy in a separate class with a common interface.

The solution to the problem with different pricing strategies:

According to the strategy pattern, we create multiple SalePricingStrategy classes for different discount algorithms, each with a polymorphic getTotal method.

The implementation of each getTotal method will be different:

PercentDiscountPricingStrategy will discount by a percentage, and so on.

Each getTotal method takes the Sale object as a parameter so that the pricing strategy object can get the pre-discount price from the Sale and then apply the discounting rule.

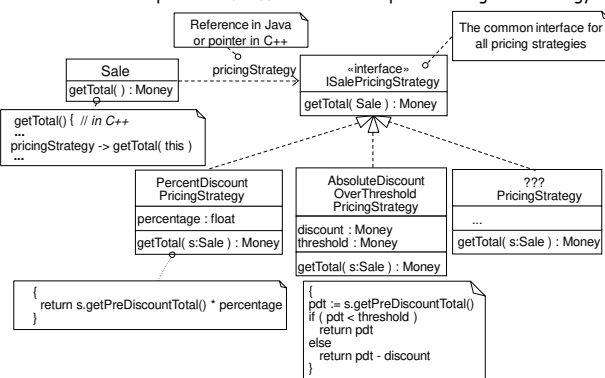
<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

9.2

Example (Different Discount Policies) :

The solution to the problem of different discount policies using the Strategy.



<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

9.3

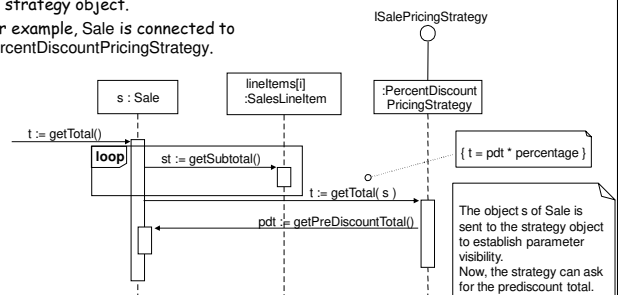
Example: The solution to the problem using the Strategy (contd)

A **strategy object** is attached to a **context object**, the object to which it applies the algorithm.

In this example, the context object is the Sale.

When a getTotal message is sent to a Sale object, it delegates some of the work to its strategy object.

For example, Sale is connected to PercentDiscountPricingStrategy.



<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

9.4

Underlying OO Principles of the Strategy pattern:

Principle: "Find what varies and encapsulate it"

This is one of the principles that the Strategy pattern is based on.

In our case study, varying parts are different discount strategies.

We separate these varying parts (pricing policies) from the stable part (Sale) of the system and encapsulate (group) them behind an abstract class (or interface in Java) (slide 9.3).

Remember "Protected Variations" (GRASP).

The details (types) of these strategies are hidden from the user (Sale).

The context object (Sale) must include a reference or a pointer to the interface (Java) or the base class (C++) of different strategies.

So, it gets attribute visibility to its strategy and can be connected to different strategy objects in runtime.

<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

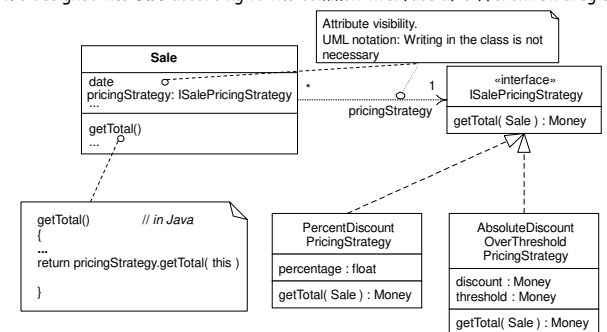
9.5

• Principle 1: "Find what varies and encapsulate it"

We found varying strategies and encapsulated them.

• Principle 2: "Design to interface, not to an implementation"

We designed the Sale according to the common interface of different strategies.

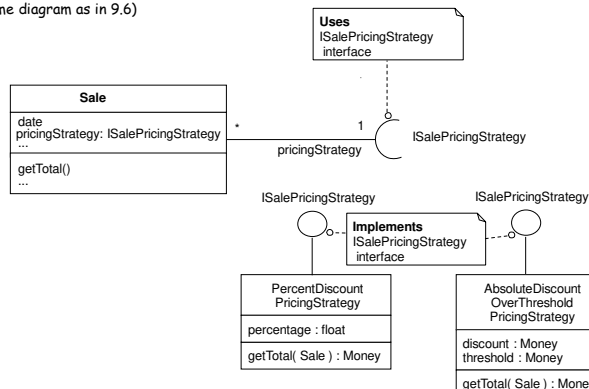


<http://akademii.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA

9.6

UML 2.X notation for interface *implementation* and *usage*.
(Same diagram as in 9.6)



<http://akademi.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA 9.7

Creating strategies (Factory):

There are two problems in the POS system related to the pricing rules.

1. Discount policies (algorithms) for sale can vary (percentage, absolute, etc.).
 2. Conditions to select the policies can vary (Monday, total > 200TL, customer).
- The first problem is solved using the strategy pattern.

Details of the second problem:

- How to decide which strategy to use (create).
- Where should the code be about the conditions?
- How to establish visibility between the context object and the strategy object.

The **Factory** pattern can be applied to create the necessary strategy object.

A PricingStrategyFactory can be responsible for creating strategies.

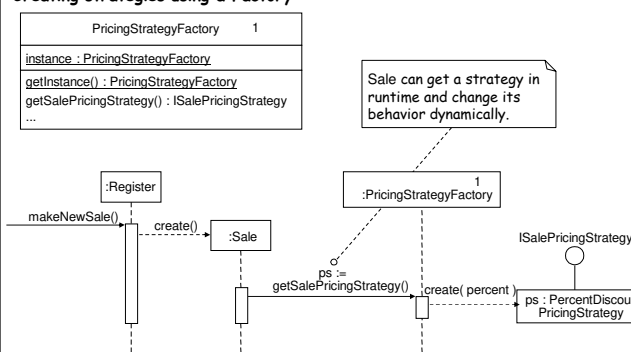
The new factory is different from the `ServicesFactory` (for adapters).

This supports the goal of High Cohesion; each factory is focused only on creating a related family of objects.

<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA 9.8

Creating strategies using a Factory:

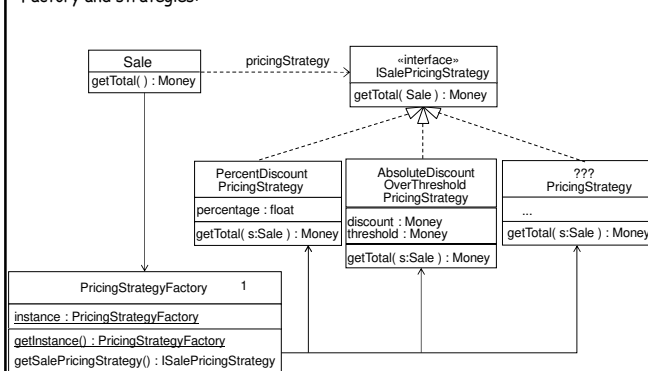


Now, `Sale` is connected to an object `PercentDiscountPricingStrategy`.
`Sale` can access the discount algorithm using the pointer `ps`, as shown in 9.4.

<http://akademi.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA 9.9

Factory and strategies:



<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>

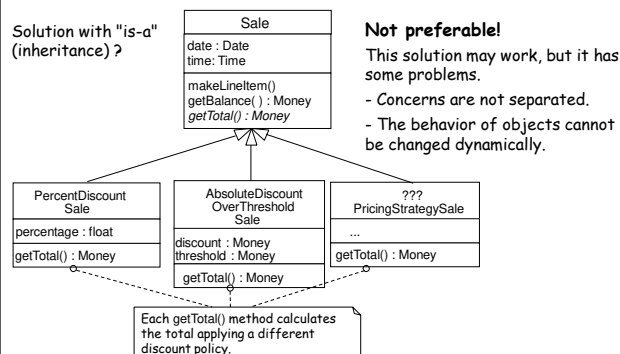
© 2012-2023 Feza BUZLUCA 9.10

Discussion: *Favor composition over inheritance* principle

Is it possible to solve the same problem using Inheritance?

We assume that we have different Sale classes with various pricing policies.

Solution with "is-a"
(inheritance) ?



<http://akademi.itu.edu.tr/en/buzluca/>
<http://www.buzluca.info>

© 2012-2023 Feza BULUTLUGA 9.1

Discussion: Favor composition over inheritance principle

Disadvantages of the solution with inheritance (is-a relation):

- Concerns are not separated: In the real world, we have only Sale classes (not different types). Various tasks are mixed in the same class.
- Inflexibility: If we create a Sale object of a specific type (for example PercentDiscountSale), we cannot change its behavior dynamically.
- We must decide on the pricing strategy during the creation of the sale.
- If we want to use another pricing strategy, we must delete the existing object and create a new one.

There is a strong connection between the base class and the derived classes.

Advantages of the solution with composition (has-a relation) (Strategy):

- Separation of concerns: Each class focuses on its own task (Sale - Pricing Strategy)
- Flexibility: Sale can request a new strategy from the factory at any time and change its behavior dynamically.

There is a weak connection (only a pointer or reference) between the context object (Sale) and strategies.

<http://akademi.itu.edu.tr/en/buzluca>
<http://www.buzluca.info>

© 2012-2023 Feza BUZLUCA 9.15

The Open-Closed Principle

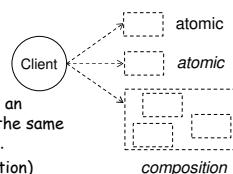
"Software entities (classes, modules, functions, etc.) should be **open for extension** but **closed for modification**".

- We should strive to write code that does not have to be changed every time the requirements change or new functionalities are added to the system.
- We should create flexible designs to take on new functionality to meet changing requirements without modifying the existing code.

The OOP concept polymorphism and the principles "Find what varies and encapsulate it" and "Design to interface not to an implementation" support the "Open-Closed Principle".

Remember the Shape Library in slide 7.13. We can add a new shape arc without changing the existing code.

Similarly, we can get services from new external systems (using adapters) or add new policies (using strategies) to our system without modifying the existing code. Later, we will cover other patterns based on the "Open-Closed Principle".

The Composite Pattern (Structural)

Sometimes a client object may get a service from an individual (atomic) object; sometimes, it may get the same service from a composition (collection) of objects.

The client object treats them (atomic or composition) identically (polymorphically) and does not have to make this distinction.

Definition:

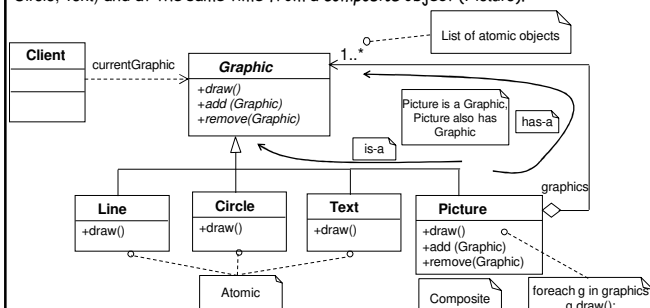
Problem: How to treat the composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

Solution: Define classes for composite and atomic objects so that they implement the same interface.

Add a list in the composite class that can include individual (atomic) objects.

Example: (From the book of the GoF)

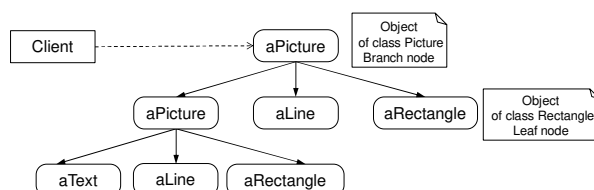
In this example, we have *atomic* shape objects (Line, Circle, Text) and a *composite* object (Picture) that consists of atomic shape objects. The class Client can get the same service (draw) from an *atomic* object (Line, Circle, Text) and at the same time from a *composite* object (Picture).

**Example: (contd)**

Objects can be composed into tree structures to represent part-whole hierarchies.

Composite lets clients treat individual objects (leaf nodes) and compositions of objects (branch nodes) uniformly.

The following diagram shows a typical composite object structure of recursively composed Graphic objects.



Atomic objects are leaf nodes, and composite objects are branch nodes of the tree.

Example: (From Larman)

How do we handle the case of multiple conflicting pricing policies?

For example, suppose that a store has the following policies:

- On Monday, there is 10TL off purchases over 100TL
- Preferred customer discount of 15%.
- Buy the product of the day, and get a 5% discount on everything.

If a preferred customer buys the product of the day and spends 150TL on Monday, what pricing policy should be applied?

Components of the problem:

1. Objects of the Sale class are sometimes connected to a single pricing strategy (atomic) and sometimes to a collection (composition) of strategies. The composite strategy solves this part of the problem.
2. The pricing strategies depend on different attributes of the Sale: Date, total, customer type, and a particular line item product.
3. Different strategies are conflicting.

We need to find solutions also for 2 and 3.

Solution:

We create a composite class CompositePricingStrategy derived from the same base class (ISalePricingStrategy) as the atomic strategies.

This composite class can also contain other ISalePricingStrategy objects.

A list in the CompositePricingStrategy class contains currently valid pricing strategies. (Composite pattern)

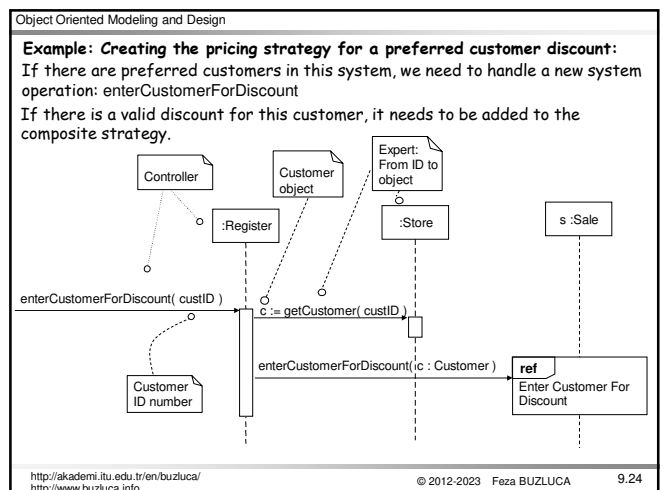
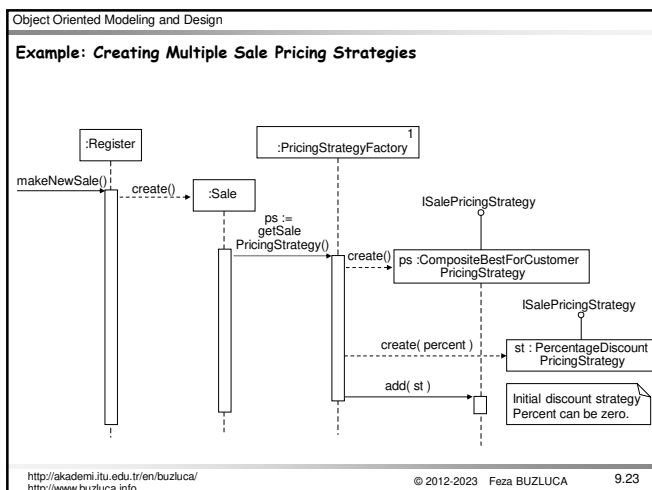
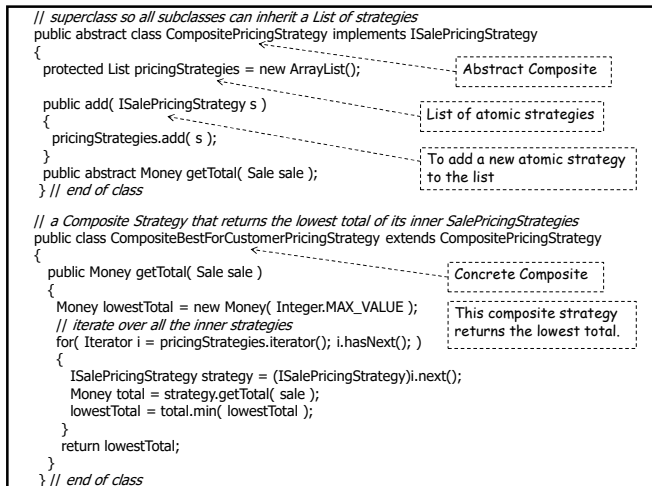
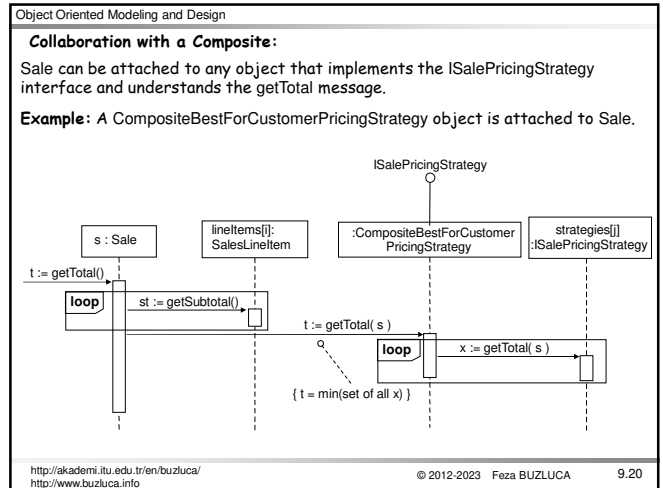
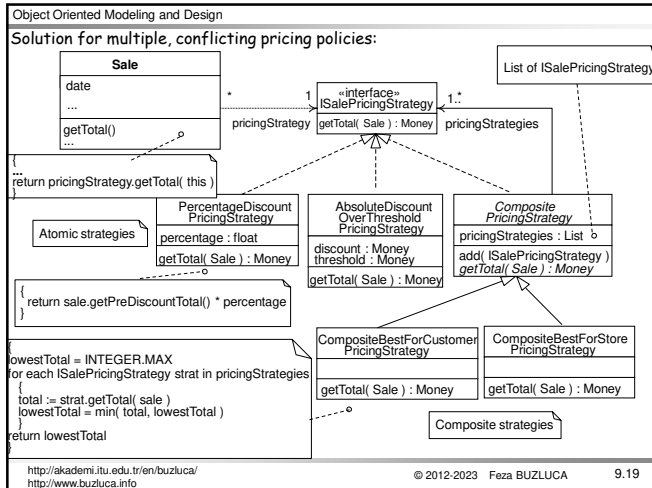
How to handle different conflicting strategies in the composite object is another strategy. (Strategy pattern again)

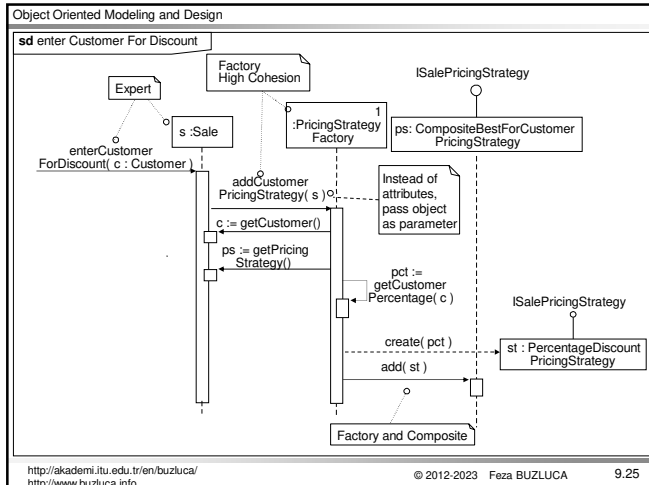
For example, the CompositeBestForCustomerPricingStrategy can try all strategies in its list and apply the strategy which returns the lowest total.

Another composite strategy (not so realistic) can be CompositeBestForStorePricingStrategy, which returns the highest total.

We can attach either a composite CompositeBestForCustomerPricingStrategy object (which contains other strategies inside of it) or an atomic PercentDiscountPricingStrategy object to the Sale object.

The Sale does not know or care if its pricing strategy is atomic or composite; they look the same to the Sale object because they are all derived from the same base class ISalePricingStrategy.





Object Oriented Modeling and Design

Considering principles and patterns in the design about customer discount

- Why does the Register not send a message to the PricingStrategyFactory, to create this new pricing strategy and then pass it to the Sale? The reason is to support *Low Coupling*. The Sale is already coupled with the factory. Furthermore, the Sale is the *Information Expert* that knows its current pricing strategy.
- Why should we transform the customerID (perhaps a number) into a Customer object? It doesn't have a pattern name, but this is a common practice in object design to transform keys and IDs for things into actual objects. Having an actual Customer object containing information about the customer, which can have functions, becomes beneficial and flexible as the design grows. For example, itemID is transformed into a ProductDescription object in the enterItem operation.
- customerID is transformed into a Customer object by the Store. Reason: By *Information Expert* and the goal of the *low representational gap*, the Store can know all the Customers. The Register asks the Store because the Register already has attribute visibility to the Store (from earlier design work).

http://akademii.itu.edu.tr/en/buzluca/
http://www.buzluca.info

© 2012-2023 Feza BUZLUCA 9.26

Object Oriented Modeling and Design

Considering principles and patterns in the design (contd)

- Passing aggregate object as a parameter:
In the addCustomerPricingStrategy(s: Sale) message, we pass a reference to the Sale object s to the factory. Then the factory asks for the Customer and PricingStrategy from the Sale.
Why d not we send just these two parameters to the factory?
Principle: Instead of individual attributes or child objects, pass the aggregate object (actually the reference) that contains child objects (or attributes). Reason: Following this principle increases flexibility because the factory can collaborate with the entire Sale in ways we may not have previously considered necessary.
In future design steps, new parameters (attributes) may be necessary. In this case, we don't need to change the interfaces of our methods; the factory can get them from Sale by calling the necessary get functions.

Note: The composite pattern is not used only with the strategies. This pattern provides that a client object treats individual objects (atomic) and group of objects (composition) identically (polymorphically), and does not have to make this distinction.

http://akademii.itu.edu.tr/en/buzluca/
http://www.buzluca.info

© 2012-2023 Feza BUZLUCA 9.27

Object Oriented Modeling and Design

The General Structure of the Composite Pattern:

```

public class Client {
    private Component myComponent; // reference (or pointer)

    public job()
    {
        .....
        myComponent.operation(); // it can be atomic (leaf) or composite (branch node)
    }
}

```

http://akademii.itu.edu.tr/en/buzluca/
http://www.buzluca.info

© 2012-2023 Feza BUZLUCA 9.28