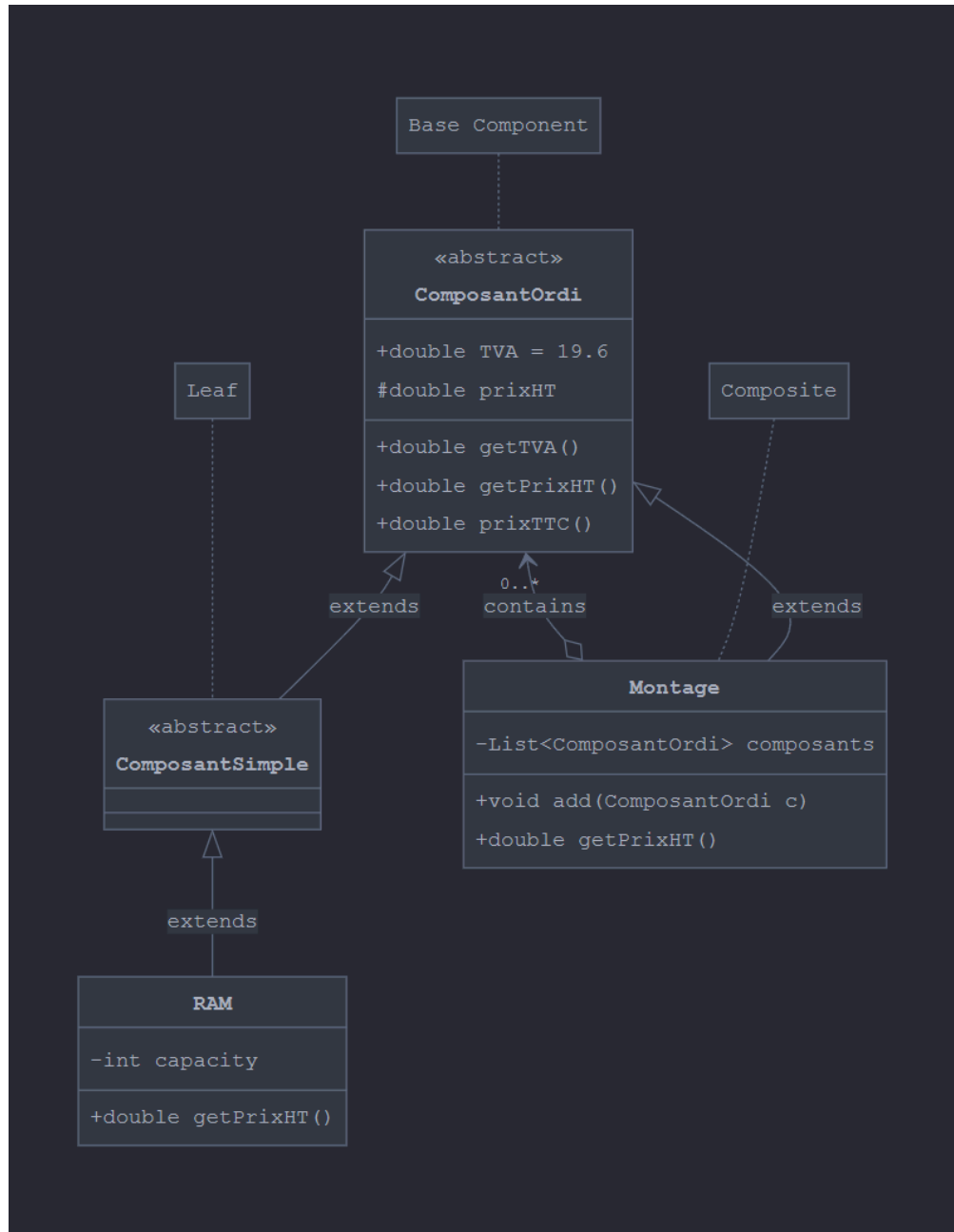


IGL s1 2022-2023
Maksym Lytvynenko

Exercise A

1. UML diagram showing the architecture using the Composite pattern since we're dealing with components that can contain other components.



2. Explain the relationship between higher-order functions and dynamic binding using the `prixTTC()` method of `ComposantOrdi` class as an example.

- Higher-Order Function Characteristics: The `prixTTC()` method is acting as a higher-order function because it uses other methods (`prixHT()` and `TVA()`) that it doesn't directly implement.
- Dynamic Binding: When `prixTTC()` calls `this.prixHT()`, the actual method that gets executed isn't determined until runtime. For example:
 - If 'this' is a `RAM` object, it will use `RAM`'s implementation of `prixHT()`
 - If 'this' is a `Montage` object, it will use `Montage`'s implementation that might sum up all its components

3. Write the code for the `Montage` class and its necessary methods.

```
public class Montage extends ComposantOrdi {
    // We use a List to store components since order might matter for assembly
    private List<ComposantOrdi> components;
    public Montage() {
        // Initialize our collection in the constructor
        components = new ArrayList<>();
    }
    // Method to add components to the assembly
    public void add(ComposantOrdi component) { components.add(component); }
    // Override getPrixHT to calculate total price of the assembly
    @Override
    public double getPrixHT() {
        // Sum up the prices of all components
        double totalPrice = 0;
        for(ComposantOrdi component : components) {
            totalPrice += component.getPrixHT();
        }
        return totalPrice;
    }
}
```

4. Extension points and control inversions in `CompOrdiFramework` exist at two main levels:

- Extension Points:
 - The abstract `ComposantOrdi` class serves as the primary extension point through abstract methods like `getPrixHT()`, enabling the addition of new component types.
 - The `Montage` class provides extension through composition via its `add(ComposantOrdi)` method, allowing the creation of complex structures.
- Control Inversions: The `prixTTC()` method in `ComposantOrdi` demonstrates control inversion: it defines the skeleton for price calculation but delegates the specific HT price calculation to subclasses through `getPrixHT()`. This Template Method pattern inverts traditional dependency by making the framework depend on concrete implementations rather than the reverse.

5. Currently, our `Montage` class doesn't allow chaining because it returns `void`. To enable the desired syntax, we need to modify the `add` method to return the object itself:

```
public class Montage extends ComposantOrdi {
    public Montage add(ComposantOrdi component) {
        super.add(component);
        return this; // Return 'this' to enable chaining } }
```

Exercise B

Coherent assemblies through using the Abstract Factory pattern.

// Abstract Factory interface

```
public interface MontageFactory {
    public Processeur createProcesseur();
    public CarteMere createCarteMere();
    public RAM createRAM();
    // ... other creation methods }
```

// Concrete factory for configuration 'A'

```
public class MontageAFactory implements MontageFactory {
    @Override
    public Processeur createProcesseur() { return new Processeur("Intel-i5"); }
    @Override
    public CarteMere createCarteMere() { return new CarteMere("ASUS-A320"); }
    @Override
    public RAM createRAM() { return new RAM("8GB"); } }
```

// The Montage class that uses the factory

```
public class MontagePredefini extends Montage {
    private MontageFactory factory;
    public MontagePredefini(char config) { switch(config) {
        case 'A': factory = new MontageAFactory(); break;
        // other configurations... }
    }
    // Automatically assemble using the factory
    this.add(factory.createProcesseur()).add(factory.createCarteMere())
    .add(factory.createRAM()); } }
```

Exercise C

1. First signature (with Montage parameter)

```
public boolean equiv(Montage c, String critere)
```

- More specific - only accepts Montage objects
- Restricts method to only compare with other Montages
- Violates Liskov Substitution Principle because it's less general than parent class method

Second signature (with ComposantOrdi parameter):

```
public boolean equiv(ComposantOrdi c, String critere)
```

- Matches parent class signature exactly
- Can accept any ComposantOrdi (more flexible)
- Follows Liskov Substitution Principle
- Allows comparing a Montage with any component

The second signature is better because it properly overrides the parent method, Maintains substitutability, Provides more flexibility.

2. `m2.equiv(m4, "x"):`

- Receiver m2 has static type Montage
- Uses Montage's equiv method

`m2.equiv(m4, "x"):`

- Same as above

`m4.equiv(m2, "x"):`

- Receiver m4 has static type ComposantOrdi
- Uses ComposantOrdi's equiv method

`m3.equiv(m4, "x"):`

- Receiver m3 has static type ComposantOrdi
- Uses ComposantOrdi's equiv method

`m4.equiv(m3, "x"):`

- Receiver m4 has static type ComposantOrdi
- Uses ComposantOrdi's equiv method

The method selection is first based on the static type of the receiver. In `m2.equiv()`, m2 is of static type Montage, while in `m4.equiv()`, m4 is of static type ComposantOrdi. This determines which method signature is considered at compile time

Dynamic Linking:

- Only comes into play after the method signature is selected through static typing
- Determines which implementation to use based on the actual object type at runtime
- Not relevant in our case because equiv is not using any polymorphic calls internally

Inclusion Polymorphism:

- Allows ComposantOrdi references to hold Montage objects
- This is why `ComposantOrdi m4 = m2` is possible
- However, the static type (ComposantOrdi) still determines which method signature is used

3. `ComposantOrdi m4 = m2`

Abstraction:

- It demonstrates programming to an interface (ComposantOrdi) rather than implementation (Montage)
- Allows for loose coupling between components

Framework Flexibility:

- Users can work with generalized types (ComposantOrdi)
- Makes it easier to extend the framework with new component types
- Enables polymorphic behavior

4. Using the Double Dispatch pattern:

```
// In ComposantOrdi
public boolean equiv(ComposantOrdi c, String critere) {
    // Let the argument handle the comparison
    return c.equivFrom(this, critere); }

// Add new method for double dispatch
public abstract boolean equivFrom(ComposantOrdi c, String critere);
public abstract boolean equivFromMontage(Montage m, String critere);
public abstract boolean equivFromRAM(RAM r, String critere);

// In Montage
@Override
public boolean equivFrom(ComposantOrdi c, String critere) {
    return c.equivFromMontage(this, critere); }

// In RAM
@Override
public boolean equivFrom(ComposantOrdi c, String critere) {
    return c.equivFromRAM(this, critere); }
```

When we call `m4.equiv(m1, "x")`:

```
// First call
m4.equiv(m1, "x") // Calls ComposantOrdi's equiv

// Inside ComposantOrdi.equiv:
public boolean equiv(ComposantOrdi c, String critere) {
    return c.equivFrom(this, critere); // 'c' is RAM, 'this' is Montage }

// Second call - goes to RAM's equivFrom because c is RAM
public boolean equivFrom(ComposantOrdi c, String critere) {
    return c.equivFromRAM(this, critere); // 'c' is Montage, 'this' is RAM }

// Final call - goes to Montage's equivFromRAM
public boolean equivFromRAM(RAM r, String critere) {
    return false; // Montage decides it can't be equivalent to RAM }
```

Exercise D

1. STATE PATTERN:

- Would represent each price variation as a state
- The RAM component would switch between states
- Each state would calculate prices differently
- Problems with this approach:
 - Can't combine multiple variations easily
 - State transitions might become complex
 - Not very flexible for adding new variations

DECORATOR PATTERN:

- Would wrap RAM components with different price variation decorators
- Each decorator adds its calculation to the base price
- Can stack multiple decorators
- Benefits:
 - Very flexible - can combine variations
 - Easy to add new variations
 - Clean separation of concerns

2.

// Base component

```
public abstract class ComposantOrdi {
    public abstract double getPrixHT(); }
```

// Basic RAM

```
public class RAM extends ComposantOrdi {
    private double basePrice; public double getPrixHT() { return basePrice; } }
```

// Base decorator

```
public abstract class VariationDecorator extends ComposantOrdi {
    protected ComposantOrdi component;
    public VariationDecorator(ComposantOrdi component) { this.component = component; }}
```

// Concrete decorators

```
public class TransportVariation extends VariationDecorator {
    private double weight;
    public double getPrixHT() {
        return component.getPrixHT() + calculateTransportCost(weight); } }
```

```
public class MaterialVariation extends VariationDecorator {
    private double materialPrice;
    public double getPrixHT() {
        return component.getPrixHT() + calculateMaterialImpact(materialPrice); } }
```

// Basic example of creating a RAM with price variations

// First create basic RAM component

```
RAM basicRAM = new RAM(100.0); // Base price 100€  
// Add transport variation based on weight  
ComposantOrdi ramWithTransport = new TransportVariation(basicRAM, 0.5); // 0.5kg  
// Add material price variation  
ComposantOrdi ramWithBothVariations = new MaterialVariation(ramWithTransport, 20.0);  
// material cost 20€  
// Calculate final price  
double finalPrice = ramWithBothVariations.getPrixHT();
```