Structural Patterns

"Code with Passion!"

Structural Patterns

Patterns that are concerned with class and object composition

- Adapter pattern
- Decorator pattern
- Bridge pattern
- Flyweight pattern
- Proxy pattern
- Composite pattern

Adapter Pattern

Adapter Pattern

- What is it?
  - The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients
- When to use it?
  - Use it when there is an existing class (possibly from a 3rd-party vendor) you need to use but it does not provide desired interface you need

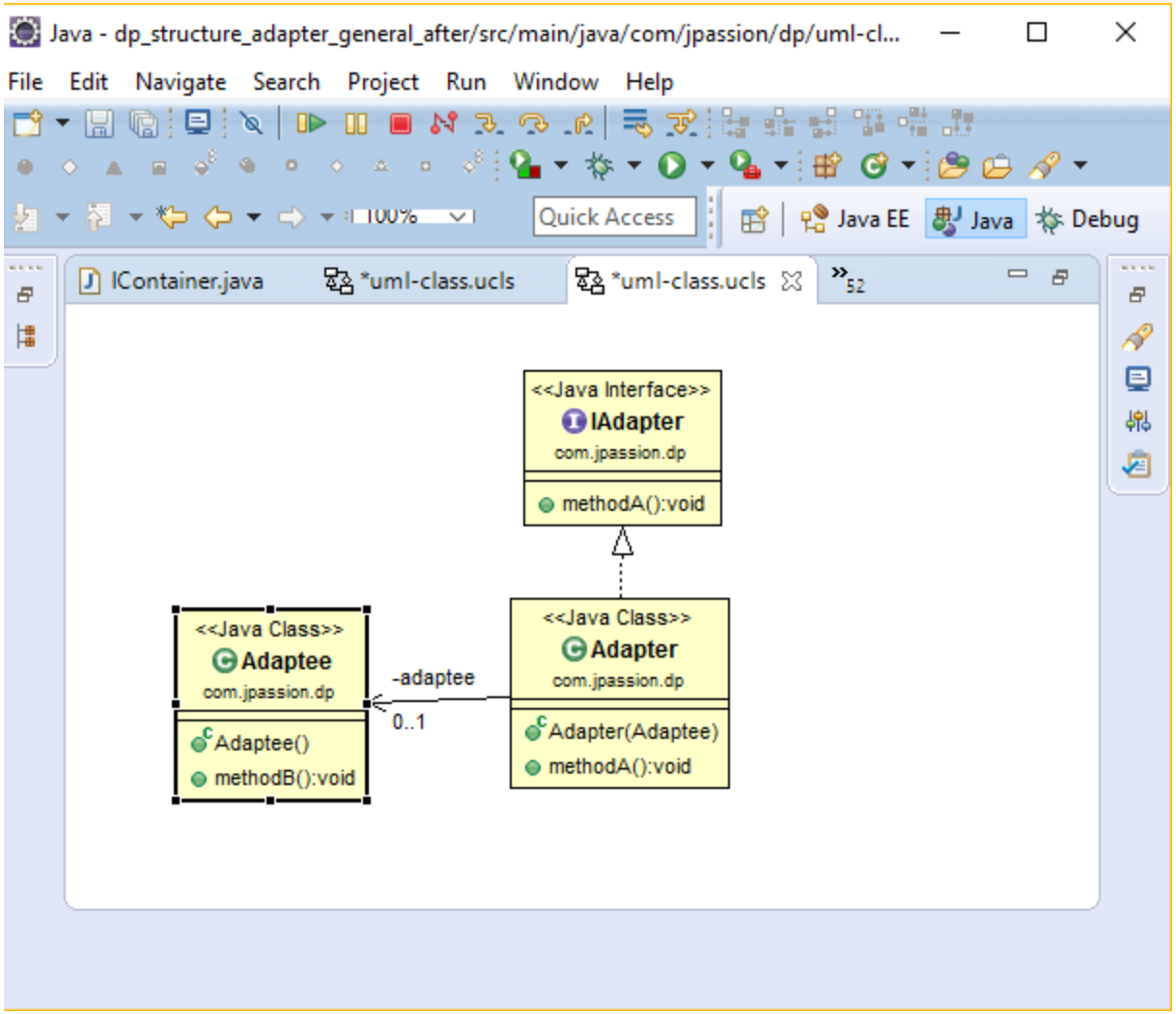
Example Scenario

- Your have bought 3rd-party library, which you don't have source code and but you want to use the functionality of it but using different interface
- Future changes
  - You might want to replace the 3rd-party library with someone else without affecting the client code

Participants

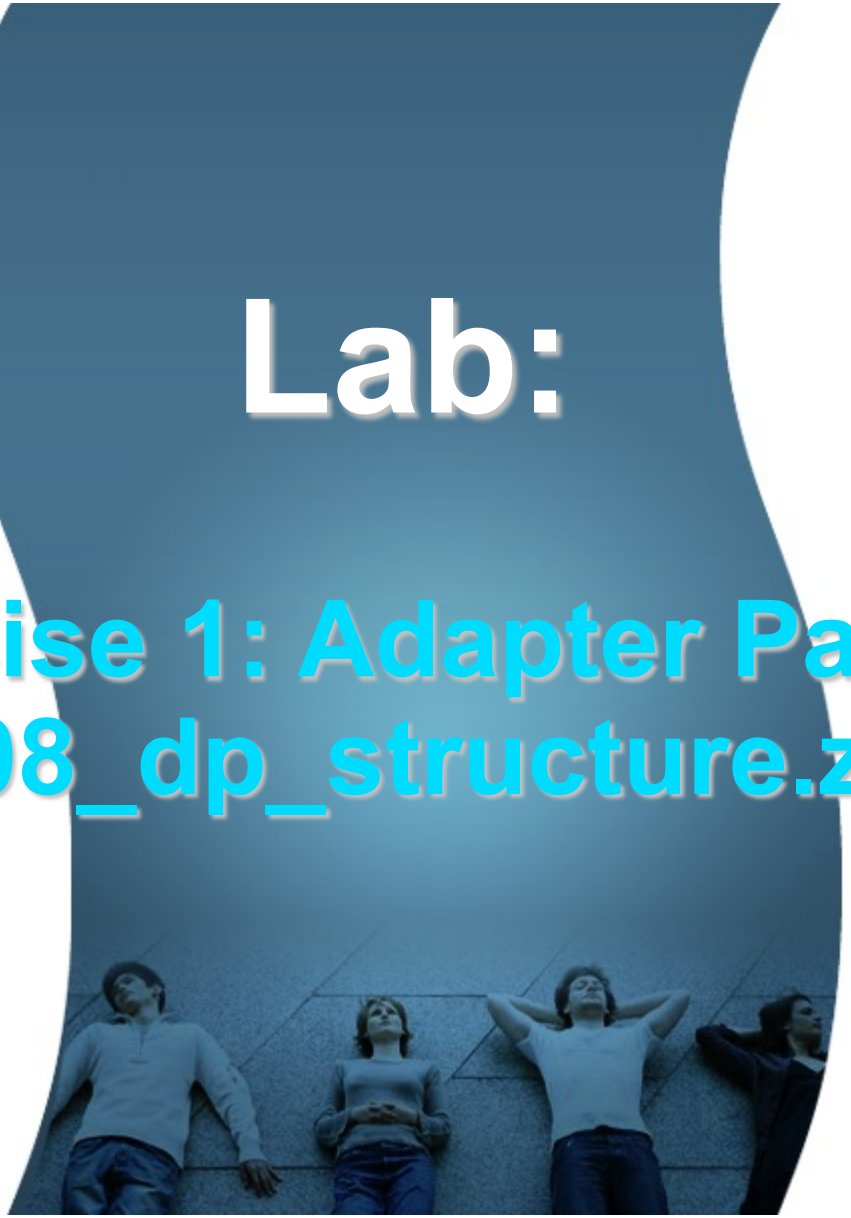
- IAdapter interface
  - Defines the interface that client uses
- Adapter class
  - Implements IAdapter interface
  - Invokes Adaptee
- Adaptee
  - Defines an existing or old interface/class that needs adapting

Adapter Pattern



Lab:

Exercise 1: Adapter Pattern
9008_dp_structure.zip



Decorator Pattern

Decorator Pattern

- What is it?
  - The decorator pattern allows extending an object's functionality dynamically (at runtime) as an object is used
- Why use decorator pattern over Inheritance for extending functionality?
  - Inheritance – you can extend an object's functionality only statically (at compile time), it could also proliferate the number of subclasses
  - Decorator pattern – you can extend an object functionality dynamically (at runtime)
- When to use it?
  - Use it when there is a need to dynamically add as well as remove functionality to a class
- Examples
  - Java IO Streams

Example Scenario

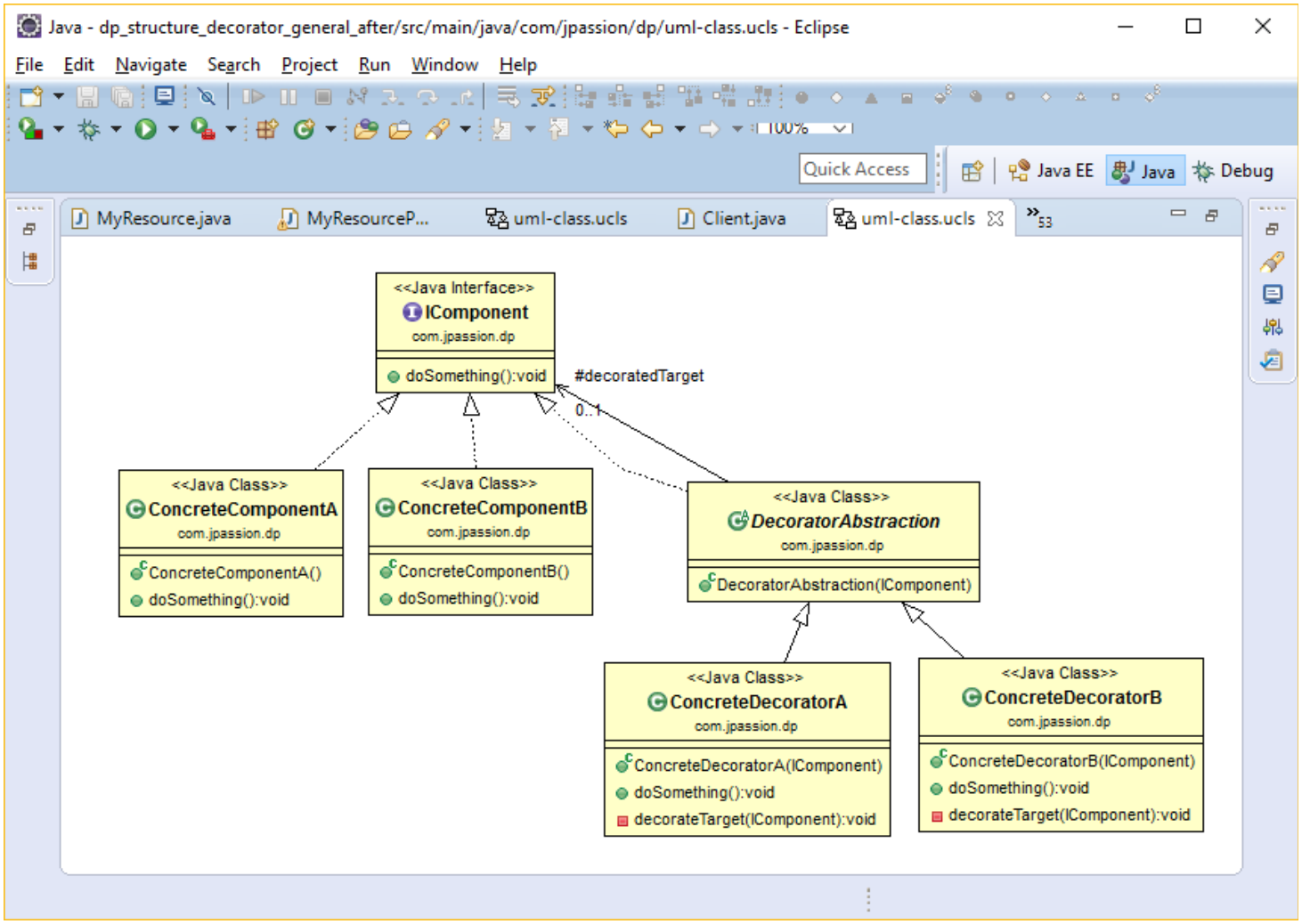
- You are an Ice cream vendor and you want your customers to choose toppings of their preference and in the order they want

- Future changes (without violating Open-Closed principle)
  - You want to add new toppings

Participants

- Component interface
  - Common interface for both components and decorators with methods that defines responsibilities
- ConcreteComponent
  - Implements Component interface
  - Target object to which additional functionality can be added
- DecoratorAbstraction
  - Implements Component interface
  - Maintains a reference to a Component object
- Concrete Decorators
  - Extends DecoratorAbstraction
  - Provide additional functionality

Decorator Pattern



Lab:

Exercise 2: Decorator Pattern
9008_dp_structure.zip

Bridge Pattern

Bridge Pattern

- What is it?
  - Bridge pattern "decouple an abstraction from its implementation so that the two can vary independently"
  - The abstraction is the owner of the implementors – both abstraction and implementor are Java interfaces

- When to use it?
  - Use it when there is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently maybe because they are owned by two different organizations

Example Scenario #1 of Bridge Pattern

- Manufacturing plants want to build various kinds of vehicles – we want these plants to add some functionality without being tied up with vehicle type
  - Car, Bike
  - For example, addWheels() method in ManufacturingPlant class will use the addWheels() method of the Car or Bike

- Future changes (without violating Open Close principle)
  - We want to add new manufacturing plants (without being concerned on the vehicle types)
  - We want to add few functionality to Manufacturing plants (without being concerned on the vehicle types)
  - We want to add Vehicle types (without being concerned on the manufacturing plants)

Example Scenario #2 of Bridge Pattern

- Different MVC frameworks (abstractions) want to use various persistence schemes (implementors)
  - For example, save() method of a particular MVC framework will use saveRecord() method of the Hibernate or OpenJPA implementors

- Future changes (without violating Open Close principle)
  - New MVC framework can be added
  - Various persistence schemes can be added

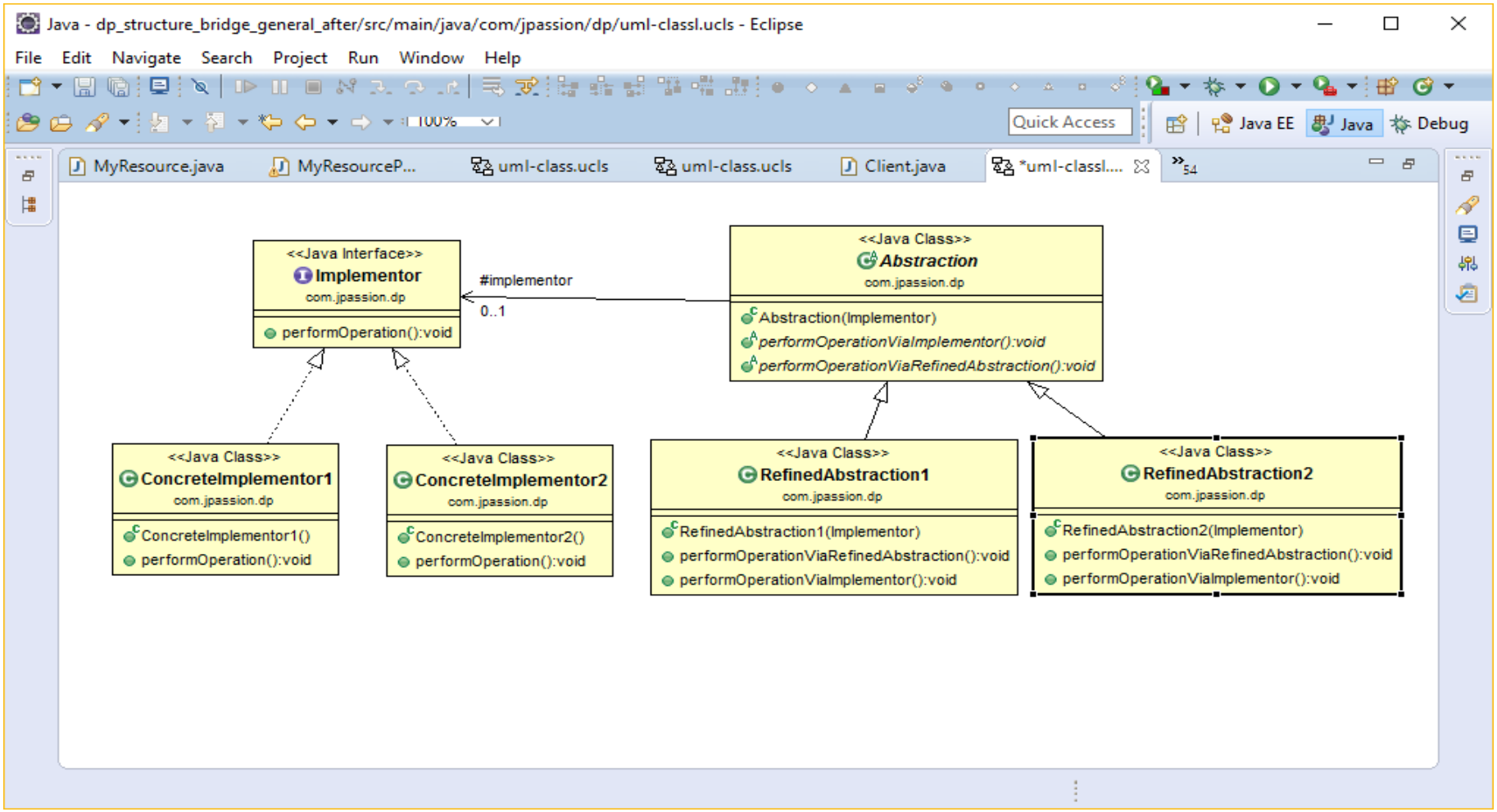
Example Scenario #3 of Bridge Pattern

- Different OS platforms (abstractions) want to use various GUI libraries (implementors)
  - Windows wants to support GUI1, GUI2, etc libraries
  - MacOS wants to support GUI1, GUI3, etc libraries
  - For example, minimizeScreen() or maximizeScreen() methods of the OS platform might in turn call mininizeScreen() or maximizeScreen() methods of the GUI implementors
- Future changes
  - We want to add new OS platforms
  - We want to add new GUI libraries

Participants

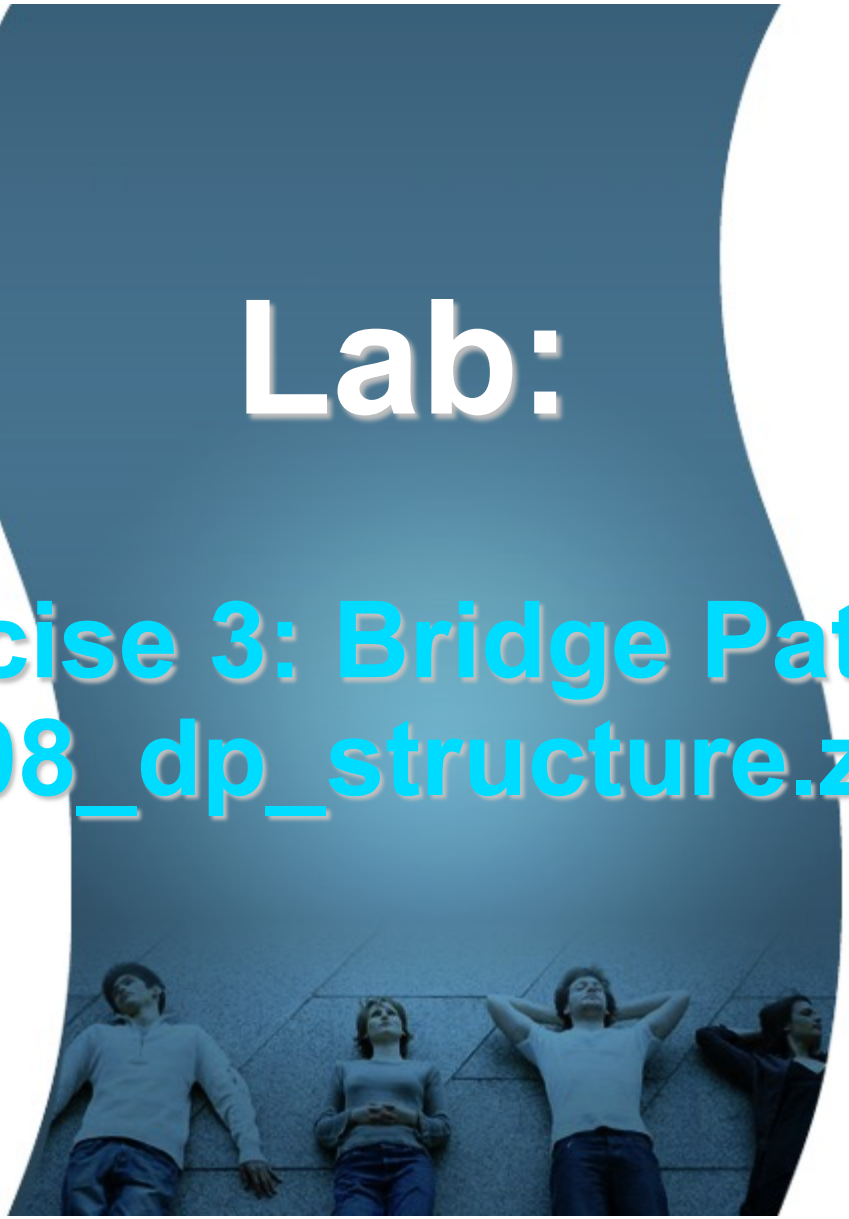
- Abstraction (abstract class or interface)
  - Abstraction defines abstraction interface for client
  - Holds reference to Implementor .
- RefinedAbstraction (normal class)
  - Extends the interface defined by Abstraction
- Implementor (abstract class or interface)
  - Defines the interface for implementation classes
- ConcreteImplementor1, ConcreteImplementor2 (normal class)
  - Implements the Implementor interface

Bridge Pattern



Lab:

Exercise 3: Bridge Pattern
9008_dp_structure.zip



Flyweight Pattern

Flyweight Pattern

- What is it?
  - Flyweight pattern tries to reuse already existing similar objects by storing them and creates new object only when no matching object is found
- When to use it?
  - Use it to reduce the number of objects created and to decrease memory footprint and increase performance

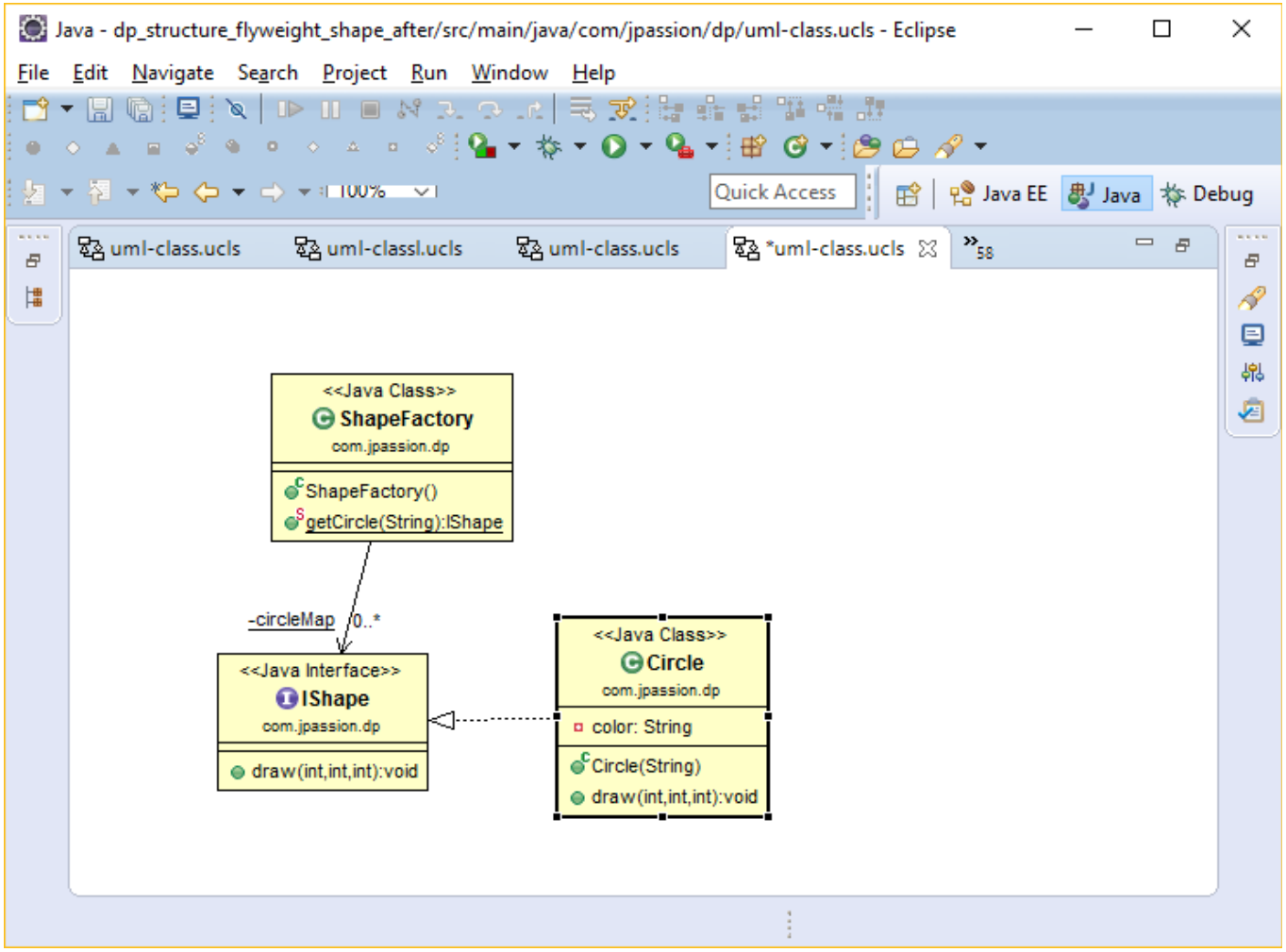
Example Scenario

- Graphical User Interface uses many common widgets and you do want to avoid the overhead of creating/destroying these widget instances
  - Text fields, buttons

Participants

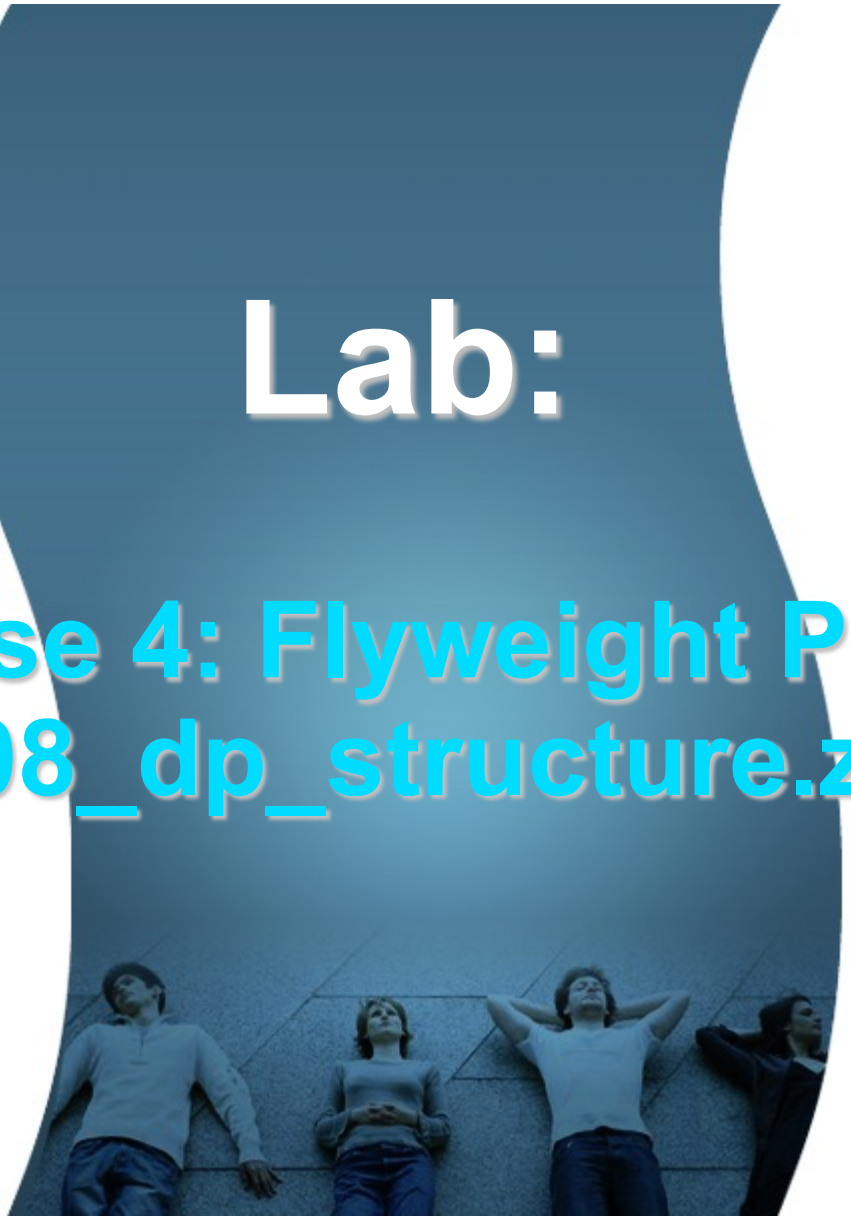
- Flyweight
  - Declares an interface through which flyweights can receive and act on external state
- ConcreteFlyweight
  - Implements the Flyweight interface and stores intrinsic state
  - A ConcreteFlyweight object must be sharable
- FlyweightFactory
  - The factory creates and manages flyweight objects.
  - In addition the factory ensures sharing of the flyweight objects - the factory maintains a pool of flyweight objects and returns an object from the pool if it is already created, adds one to the pool and returns it in case it is new.

Flyweight Pattern



Lab:

Exercise 4: Flyweight Pattern
9008_dp_structure.zip



Proxy Pattern

Proxy Pattern

- What is it?
  - In proxy pattern, a surrogate or placeholder is provided for another object to control access to it
  - In proxy pattern, an extra level of indirection can be added to support distributed, controlled, or intelligent access to real objects
- When to use it?
  - Depends on the type of Proxy (see next slide)

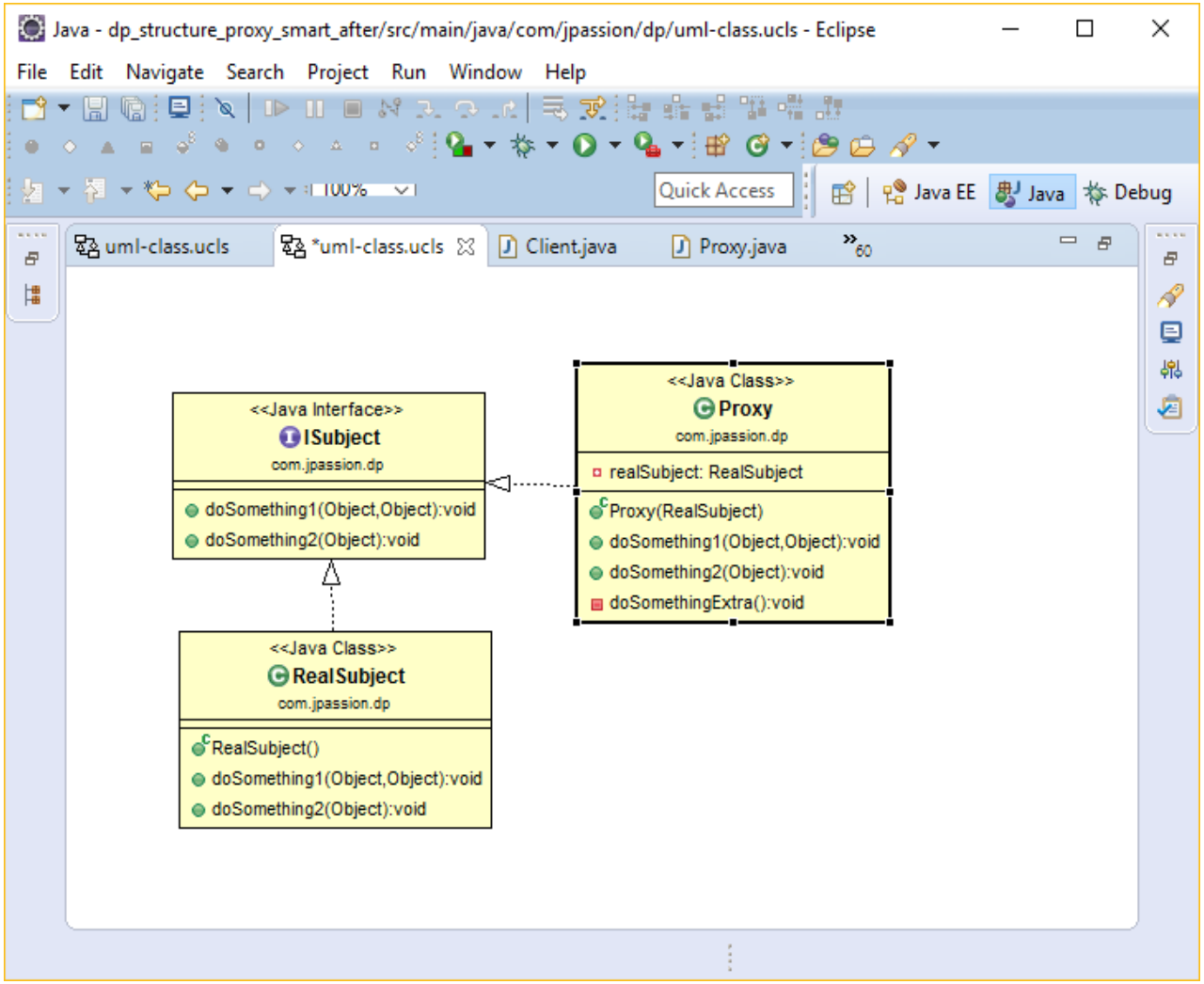
Types of Proxy

- A virtual proxy
  - A placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
- A remote proxy
  - Provides a local representative for an object that resides in a different address space. This is called "stub".
- A protective proxy
  - Controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
- A smart proxy
  - Interposes additional actions when an object is accessed.
  - Example: Counting the number of references to the real object so that it can be freed automatically when there are no more references
  - Example: Loading a persistent object into memory when it's first referenced

Participants

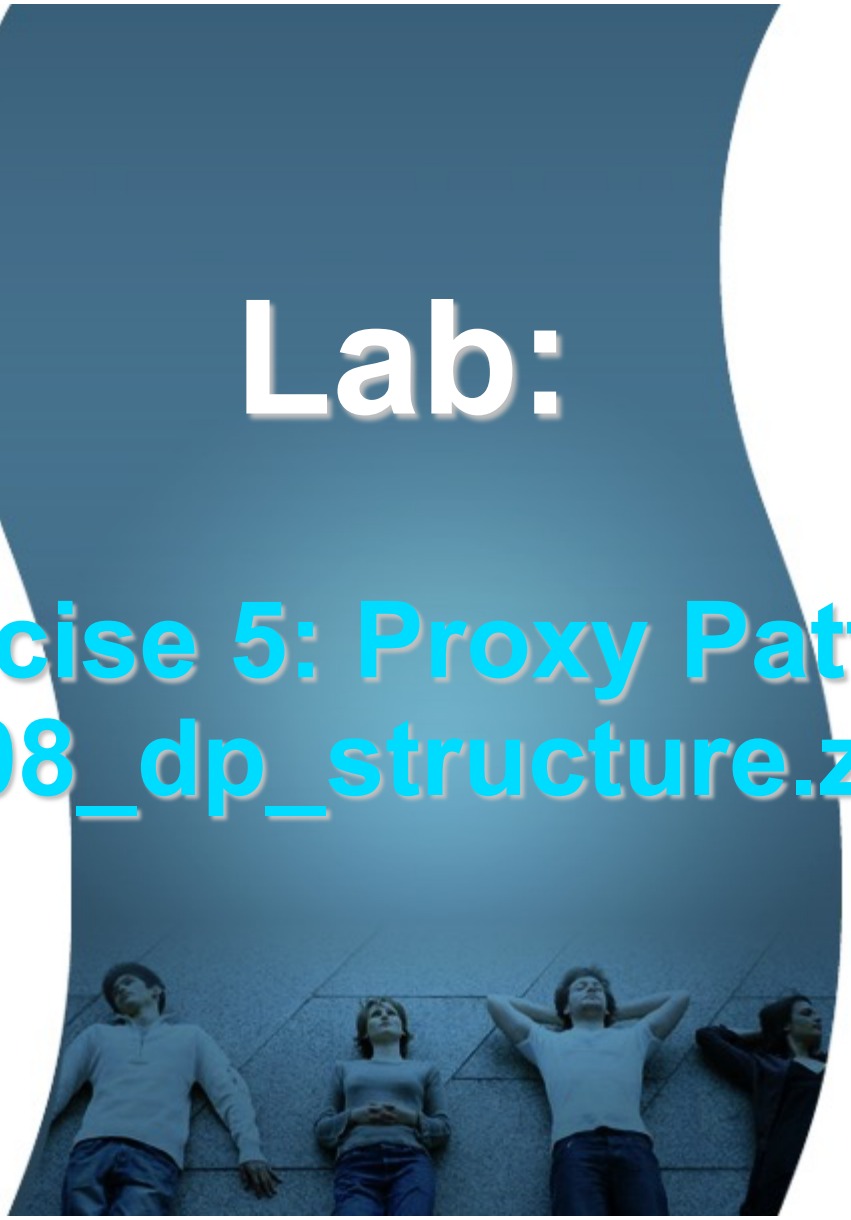
- Subject interface
  - Interface implemented by the Proxy and RealSubject
- Proxy
  - Maintains a reference to the RealSubject.
  - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
- RealSubject
  - The real object that the proxy represents.

Proxy Pattern



Lab:

Exercise 5: Proxy Pattern
9008_dp_structure.zip



Composite Pattern

Composite Pattern

- What is it?
  - Composite pattern lets you compose objects into a tree structure while treating individual objects and compositions of objects uniformly
- When to use it?
  - Use it when there is a part-whole hierarchy of objects and a client needs to deal with objects uniformly regardless of the fact that an object might be a leaf or a branch

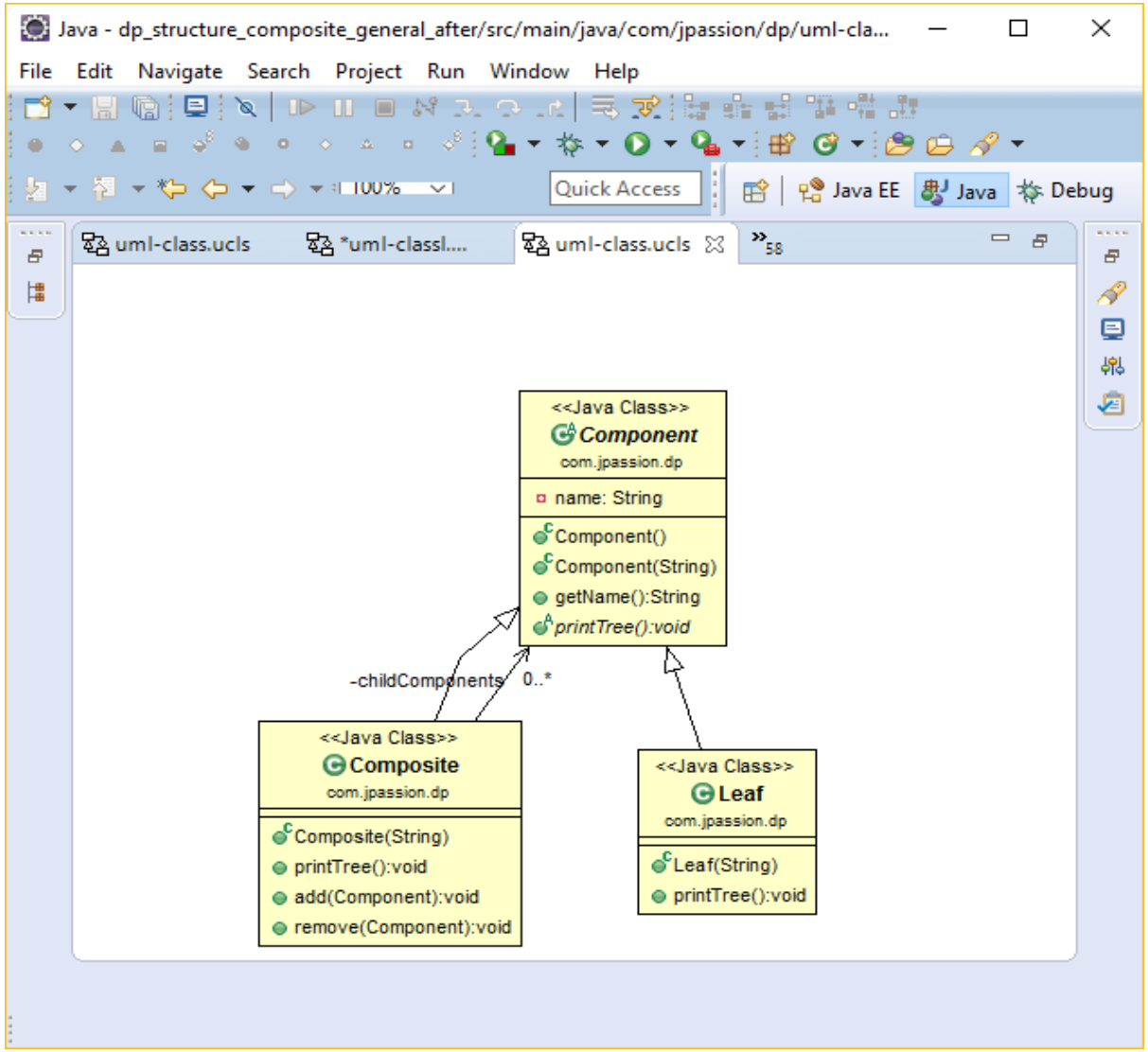
Example Scenario of Composite Pattern

- Organization structure
  - Individual contributors work for a manager, who in turn works for a director
  - We want to retrieve management information on a particular role (using the same method)
- Future changes
  - We want to add new role to the organization structure

Participants

- Component interface
  - Abstraction for leafs and composites
  - Defines the interface that must be implemented by the objects in the composition
- Leaf
  - Leafs are objects that have no children. They implement services described by the Component interface.
- Composite
  - A Composite stores child components in addition to implementing methods defined by the component interface
- Client
  - The client manipulates objects in the hierarchy using the component interface

Composite Pattern



Lab:

Exercise 6: Composite Pattern
9008_dp_structure.zip

Code with Passion!