2.1 Bridge Pattern

The bridge pattern allows separation of abstraction of a class from its implementation. Therefore, the implementation can be changed at runtime. The design pattern also allows that the abstraction and implementation can vary and evolve independently.

Need to add figure!!!!!!!

Figure 2.1 Bridge Pattern Class Model

Documentation

The CROM representation of the bridge pattern by the contains three classes. This can be seen in Figure 2.1. The Compartment Type Bridge Pattern allows instantiation of the pattern. The second class GenericComponent represents objects in which the abstraction and the implementation are to be separated from each other. Furthermore, this class can also represent a client who accesses such an object.

The implementation roles are inherited by the GenericImplementation class. The role model of the design pattern is shown in Figure 2.2. The roles of the design pattern can be divided into three groups. The first is the **abstraction role**. It contains the role Abstraction and the refined abstraction roles. These are summarized in the role group RefinedAbstractions. They are equivalent to the subclass RefinedAbstraction from the representation of the Gang of Four to understand and serve the refinement of the role Abstraction. This provides an interface for the implementation roles. That's why you find the method operation in all abstraction roles. This is representative of the methods whose implementation should be separated from the abstraction of the class. The refined abstraction roles further specialize the interface of these methods. The role Abstraction also has a method to change its implementation at the time of creation or even afterwards. This can also be called from the outside. This allows, for example, that the decision on the implementation can be delegated to another object at the time of creating an abstraction. If this decision is resolved by a standard implementation, the value of the defaultImplementation attribute is used as the input parameter for this method.

The second group is the **implementation roles**. These are used for the implementation of the interfaces given by the abstraction roles. This group includes the implementation role and the concrete implementation roles. These roles are equivalent to the similar structure of the first group, a role-based representation of the inheritance structure that uses the Gang of Four. Instead of representing the inheritance structure with several classes, the refinement in this model is captured by

the roles, the role group, and the associated role equivalence. The operationImp method of these roles is representative of the implementation of the operation method from the abstraction roles. This is refined in the concrete implementation roles. Third, there is the role of the **client.** This accesses an abstraction and possesses the method callOperation.

In a meaningful instantiation of this pattern, there is at least one class that plays the role of abstraction and precisely a refined abstraction role. At the same time there is also at least one class that fulfills the implementation roles. The role of implementation and a role of concrete implementations are played. There does not need to have a client in this pattern. From this information, the occurrence constraints can be derived.

There are 2 role equivalencies in the model. The first expresses that a class playing the Abstraction role must also play the role group RefinedAbstractions, and vice versa. Along with the Role Group Constraint of the role group, the equivalence implies that a component, in addition to the general abstraction role, also has to take on exactly one more refined abstraction role. The same thing happens between the role implementation and the concrete implementation roles. Here an object of class GenericImplementation plays the role of the general implementation and exactly one concrete implementation role. There are relations in the role model between client and abstraction as well as between the role and implementation. The useAbstraction relation describes how the client accesses the abstraction of a component. The second isImplementation relation is an aggregation and detects that the instances of the Abstraction role have exactly one instance of the abstract implementation role. It also describes that abstraction further delegates the client's method call from operation to the class playing the implementation role. It calls the method operationImp. This then executes this request on its concrete implementation role.

Figure 2.2 Bridge Pattern Role Class Model

Evaluation

The following features are shown in **Gang of Four**.

1. The Bridge pattern separates the abstraction of a class from its implementation. The implementation of a class can be changed at runtime.

2. The abstraction, as well as the implementation can be refined by an inheritance structure.

3. Changes in the implementation have no influence on the code of the clients.

4. The pattern allows the classification of a class into two aspects, which is realized by the two inheritance structures.

5. A particular implementation can be shared by the design pattern of multiple objects. This feature is also described textually and with example code in the Implementation component.

6. The participatants in the design pattern: Abstraction, Implementor, the concrete implementation classes, and the refined abstraction classes.

7. There are 2 inheritance structures. One refines the class Abstraction, while the other implements the specialist. These can be extended independently of each other. This feature is also described textually.

8. The class Abstraction is abstract.

9. The class Abstraction provides the interface for the implementation classes.

10. The class Abstraction has an object of the implementation classes.

11. The class Implementor is abstract.

12. The class Implementor provides the interface for the concrete implementation classes.

13. The interface of the abstraction classes can be more complex than that of the implementation classes.

14. The concrete implementation classes implement the methods of the class Implementor.

15. A client acts with the class Abstraction.

16. For a method call in the abstraction class, it delegates the call to the implementation. This feature is also mentioned textually in the Collaborations component.

17. Separating the abstraction from the implementation saves compile time because the implementation and abstraction classes are compiled independently.

18. The separation of the abstraction from the implementation can be used for a better structuring of the system. At the highest architectural level, only the abstraction and the abstract implementation class need to be known.

19. The separation of the abstraction from the implementation can be used to hide implementation details such as reference counts.

20. Even with only one implementation, the design pattern can still be used to separate the abstraction from its implementation.

21. The decision as to which concrete implementation class is instantiated can be realized by various means.
a) An abstraction class that knows all implementation classes chooses the right one. The decision may depend on the given parameters of its constructor.
b) By default, the abstraction class instantiates an implementation class. This is then changed at runtime at certain events.
c) The decision is delegated to another object that knows the implementation classes. This variant is shown in the sample code section.

The following features are to be found in the representation of **Dirk Riehle** design patterns .

22. The design pattern includes the following roles: client, abstraction, and implementer.

23. The client does not act with the implementation roles, but with the role Abstraction. This feature is also described textually.

24. The role Abstraction has an aggregation to the role Implementer.

25. The composition constraints from the Role Relationship Matrix [Rie09, p. 34] apply.

The Feature Table is Shown as follows.