* 1. Structural Patterns
* Structural patterns are concerned with how classes and objects are composed to form larger structures
* 1. Intent
* Convert the interface of a class into another interface clients expect
* 2. Applicability
* you want to use an existing class, and its interface does not match the one you need.
* you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
* (object adapter only) you need to use several existing subclasses, but it's  impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.
* 已知: 一个existing class和一个需要用到的interface, 但是不match, 用一个adaptor 来使interface可以用到existing class功能

3. Structure

* 一个interface 叫 Target { void request(); }
* 一个已经有的class叫 Adaptee
* Adaptor implements Targer [extends Adaptee]{

Adaptee adaptee;

Void request() { adaptee.specificRequest(); }

* }
* Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

4. Consequences

1) Class adaptor (extends a class)

* adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
* lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
* introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

2) Object adaptor (use composition)

* lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
* makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

3） two way adaptors (implements both interface)

5. Implementation

* find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation
* use abstract operations
* using delegate objects
* parameterized adaptors
* 6. Related Patterns
* Bridge (151) has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object.
* Decorator (175) enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.
* Proxy (207) defines a representative or surrogate for another object and does not change its interface.
* 1. Intent
* Decouple an abstraction from its implementation so that the two can vary independently

2. Applicability

* you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
* both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
* changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
* (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
* you have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts. Rumbaugh uses the term "nested generalizations" [RBP+91] to refer to such class hierarchies.
* you want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class [Cop92], in which multiple objects can share the same string representation (StringRep).

3. Structure (接口和实现分离)

* Abstract class (Abstracion): 定义外界的接口
* Interface: Impementor { void operationImpl(); }
* Abstract class里面调用Implementor来实现功能
* Abstraction forwards client requests to its Implementor object.

4. Consequences

* Decoupling interface and implementation
* Improved extensibility (extends abstraction and implementor)
* Hiding implementation details from clients

5. Implementation

* Only one implementor
* Creating the right Implementor object
* 1. Intent
* Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
* 2. Applicability
* you want to represent part-whole hierarchies of objects.
* you want clients to be able to ignore the difference between compositions of  objects and individual objects. Clients will treat all objects in the composite structure uniformly.
* 3. Structure
* Component: interface to define a group a object
* Composite implements Component: it is a tree structure of component contains component (component has children)
* Leaf: concrete implementation of component

Collaborations

* - Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding

4. Consequences

* Defines class hierarchies consisting of primitive objects and composite objects
* Makes it easier to add new kinds of components
* makes the client simple

5. Implementations

* Related Patterns  Often the component-parent link is used for a Chain of Responsibility (223).
* Decorator (175) is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
* Flyweight (195) lets you share components, but they can no longer refer to their parents.
* Iterator (257) can be used to traverse composites.
* Visitor (331) localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.