The Bridge Pattern

Other Considerations

Douglas C. Schmidt
Learning Objectives in This Lesson

- Recognize how the *Bridge* pattern can be applied to make the expression tree structure easier to access & evolve transparently.
- Understand the structure & functionality of the *Bridge* pattern.
- Know how to implement the *Bridge* pattern in C++.
- Be aware of other considerations when applying the *Bridge* pattern.
Other Considerations of the Bridge Pattern

Douglas C. Schmidt
Consequences

+ Abstraction & implementor hierarchy are decoupled

- Can evolve separately by applying Open/Closed Principle

Enable software to be open for extension (via implementor hierarchy), but closed for modification (via stable abstraction API)

Consequences

+ Abstraction & implementor hierarchy are decoupled
  - Can evolve separately by applying Open/Closed Principle

Extend software behavior by adding new code via derived classing, not by changing/breaking existing client code.

en.wikipedia.org/wiki/Open/closed_principle has info on Open/Closed Principle.
Consequences

+ Implementors can vary at design-time or runtime

```
Expression_Tree expr_tree
    (new Composite_Add_Node
        (new Leaf_Node(3),
        new Leaf_Node(4)));

versus

Expression_Tree expr_tree
    (new Tree_Node
        ('+',
        new Tree_Node(3),
        new Tree_Node(4)));
```
Consequences

- “One-size-fits-all” abstraction & implementor interfaces

See [en.wikipedia.org/wiki/Procrustes#Cultural_references](en.wikipedia.org/wiki/Procrustes#Cultural_references)
**Consequences**

- “One-size-fits-all” abstraction & implementor interfaces
  - Can be alleviated via other patterns, e.g.,
    - **Adapter**—makes existing classes work with others without modifying code

```
Client → Target

Adapter
request()

Adaptee
specificRequest()

request() adaptee

adaptee.specificRequest()
```

[en.wikipedia.org/wiki/Adapter_pattern](http://en.wikipedia.org/wiki/Adapter_pattern) has more on **Adapter**.
Consequences

- "One-size-fits-all" abstraction & implementor interfaces
  - Can be alleviated via other patterns, e.g.,
    - **Adapter**—makes existing classes work with others without modifying code
    - **Strategy**—lets the algorithm vary independently from clients that use it

[Diagram of the Bridge pattern

---

[en.wikipedia.org/wiki/Strategy_pattern](en.wikipedia.org/wiki/Strategy_pattern) has more on *Strategy.*]
Consequences
– “One-size-fits-all” abstraction & implementor interfaces
  • Can be alleviated via other patterns, e.g.,
    • Adapter—makes existing classes work with others without modifying code
    • Strategy—lets the algorithm vary independently from clients that use it
    • Extension Interface—allows multiple interfaces to be exported by a component, to prevent bloating of interfaces & breaking of client code when developers extend or modify the functionality of the component

See www.laputan.org/pub/sag/extension-interface.pdf
Implementation considerations

- Creating the right abstraction or implementor
  - Often addressed by using Creational patterns
    - e.g., Factory Method or Builder

```cpp
class Multiply : public Operator {
    ...

    Component_Node *build() {
        return new Composite_Multiply_Node(left.build(),
                                            right.build());
    }
}
```

We’ll cover Builder later & show how it creates composite expression trees.
Implementation considerations

• Sharing implementors & reference counting
  • e.g., C++11/Boost shared_ptr

See en.wikipedia.org/wiki/Smart_pointer#shared_ptr_and_weak_ptr
Implementation considerations

- Dynamic uses of *Bridge* should be implemented via *Decorator*.

[diagram showing the Bridge pattern with Component, ConcreteComponent, Decorator, ConcreteDecoratorA, and ConcreteDecoratorB classes with method signatures and behavior modification through addedBehavior method]
Implementation considerations

- Dynamic uses of *Bridge* should be implemented via *Decorator*.

- *Decorator* enables client-specified embellishment of a core object by recursively wrapping it (possibly more than once) dynamically at runtime.

See [sourcemaking.com/design_patterns/decorator](sourcemaking.com/design_patterns/decorator)
Implementation considerations

- Dynamic uses of *Bridge* should be implemented via *Decorator*.

  *Decorator* enables client-specified embellishment of a core object by recursively wrapping it (possibly more than once) dynamically at runtime.

  - Java I/O is a famous example of the *Decorator* pattern.

Bridge is used more in C++ than in C++ (which uses interfaces & factories).
Known uses

- ET++ Window/WindowPort
- libg++ Set/\{LinkedList, HashTable\}
- ACE Reactor framework
- AWT Component/ComponentPeer

See [www.soberit.hut.fi/tik-76.278/group6/awtpat.html](www.soberit.hut.fi/tik-76.278/group6/awtpat.html)
Known uses

- ET++ Window/WindowPort
- libg++ Set/\{LinkedList, HashTable\}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl

Variations in how Socket is implemented

Variations in what service Socket provides

See docs.oracle.com/javase/tutorial/networking/sockets
Decouples synchronizer interface from its implementation so fair & non-fair semantics can be supported uniformly

Known uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl
- Java synchronizers

See [www.baeldung.com/java-concurrent-locks](www.baeldung.com/java-concurrent-locks)
Summary of the Bridge Pattern

*Bridge* decouples the expression tree programming API from its behavior & implementation to enable transparent extensibility.

Bridge Composite is an example of a “pattern compound.”

See [www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf](http://www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf)