The Bridge Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

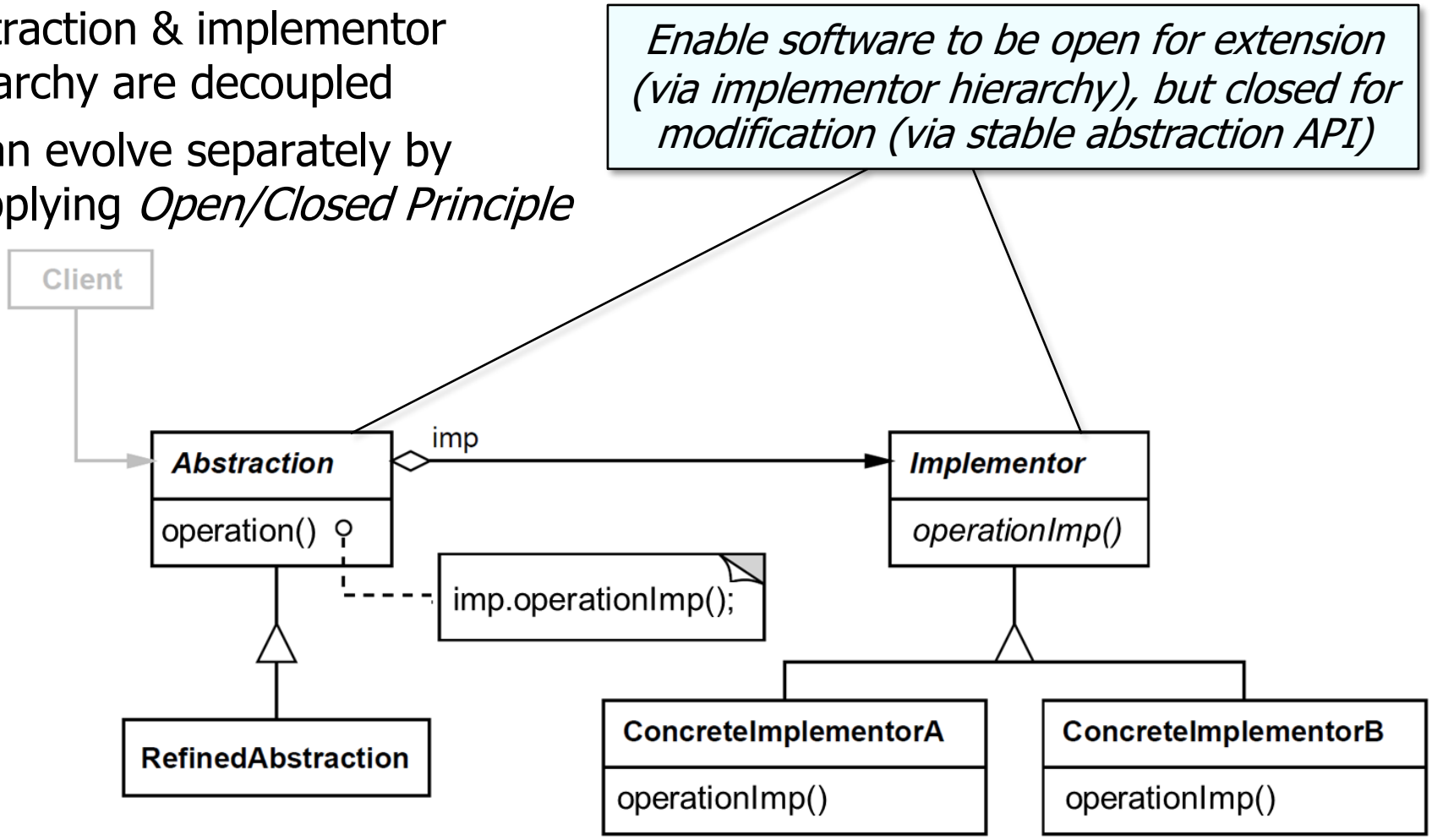
- Recognize how the *Bridge* pattern can be applied to make the expression tree structure easier to access & evolve transparently.
- Understand the structure & functionality of the *Bridge* pattern.
- Know how to implement the *Bridge* pattern in C++.
- Be aware of other considerations when applying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

Douglas C. Schmidt

Other Considerations of the Bridge Pattern

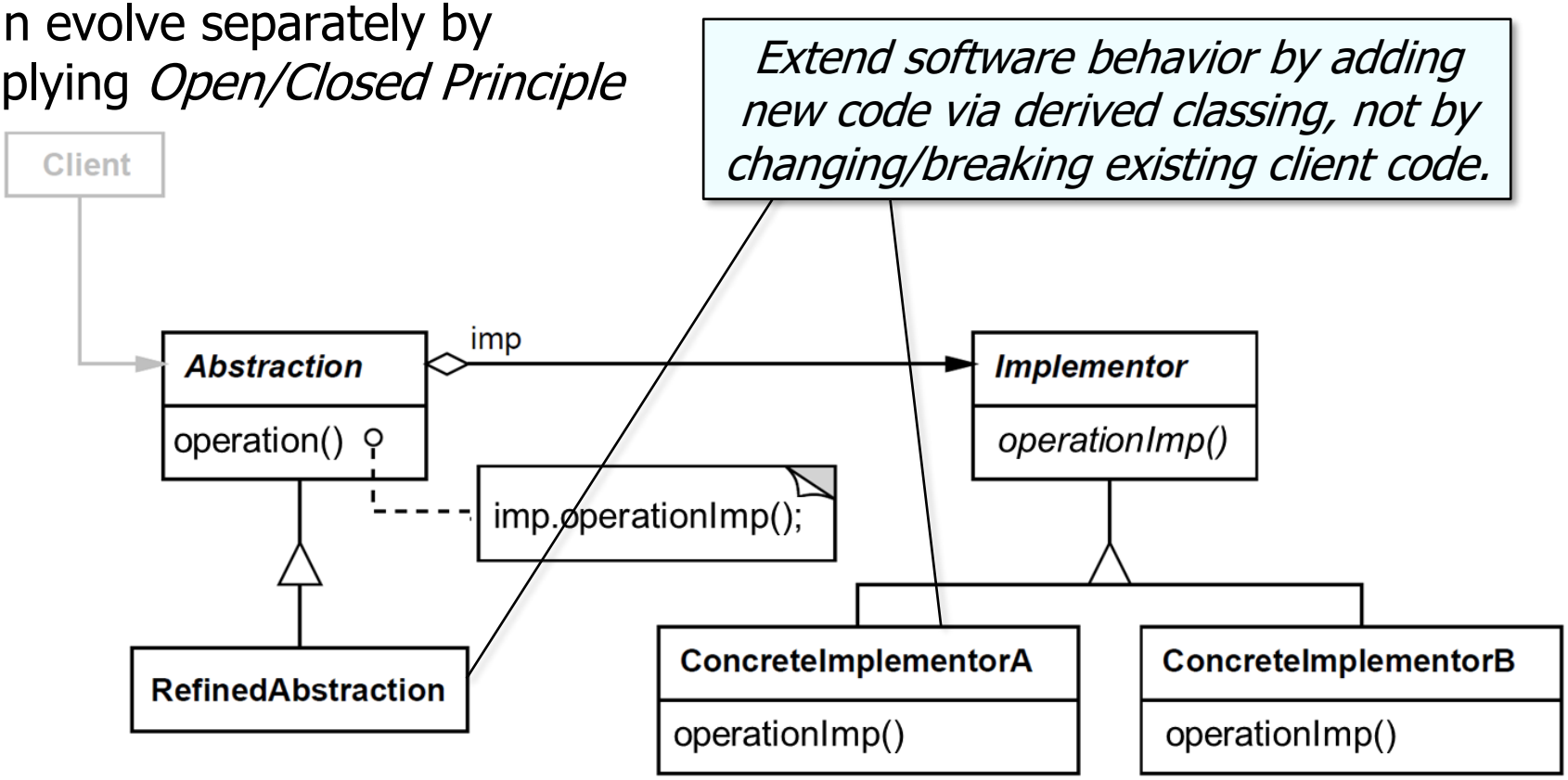
Consequences

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



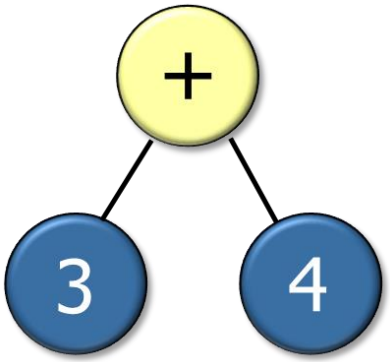
Consequences

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



Consequences

+ Implementors can vary at design-time or runtime



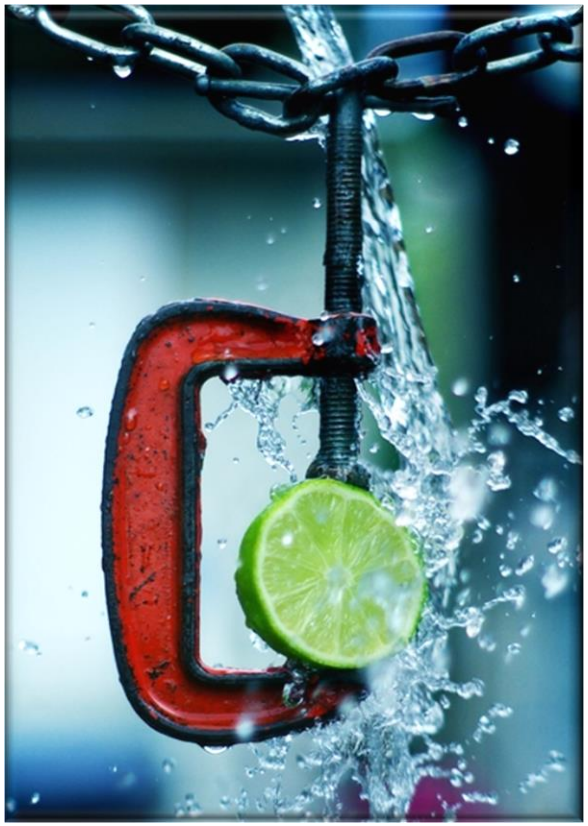

```
Expression_Tree expr_tree
  (new Composite_Add_Node
    (new Leaf_Node(3),
     new Leaf_Node(4)));
```

versus

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

Consequences

– "One-size-fits-all" abstraction & implementor interfaces

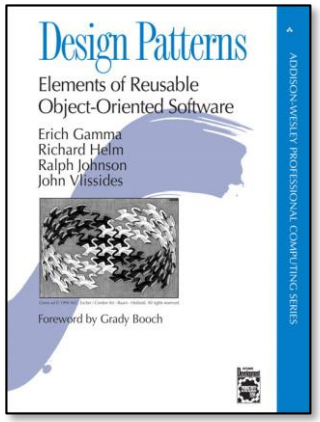


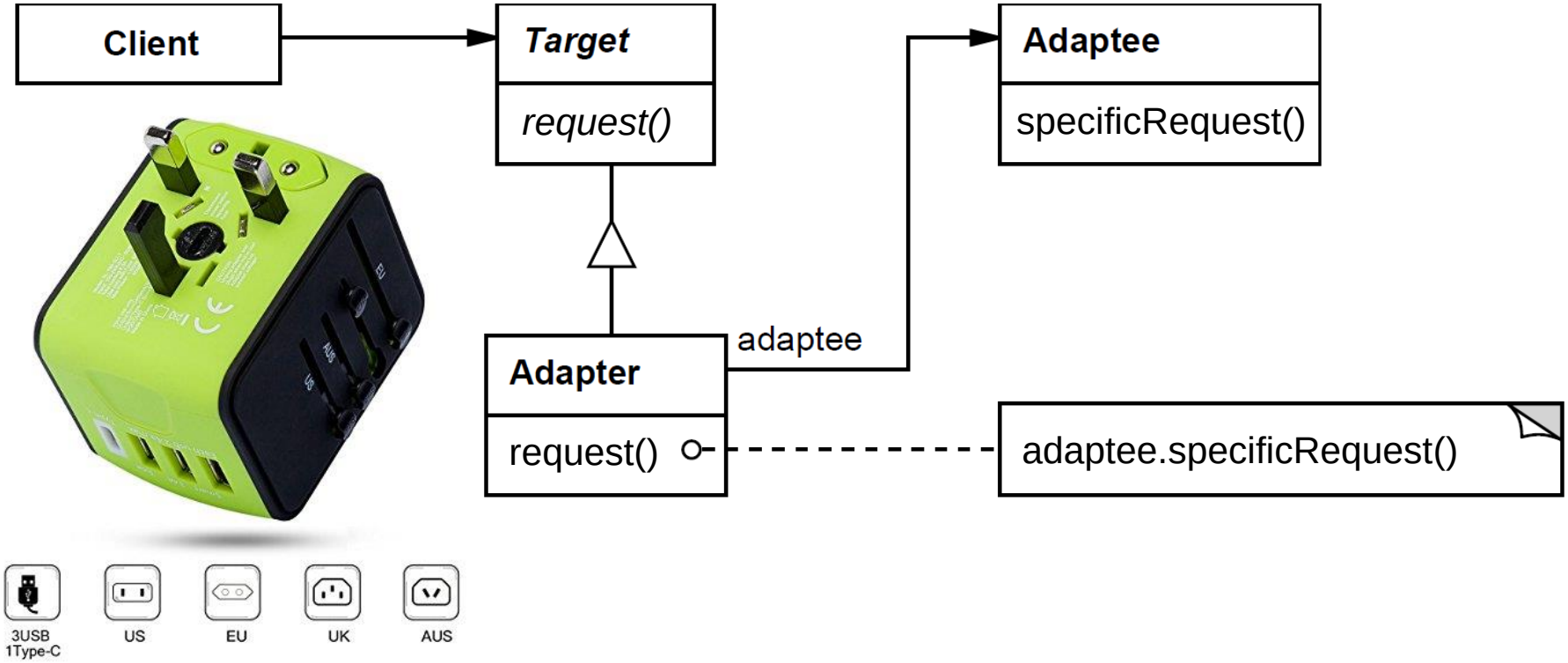
See en.wikipedia.org/wiki/Procrustes#Cultural_references

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



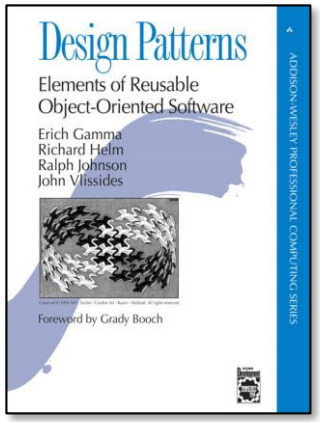


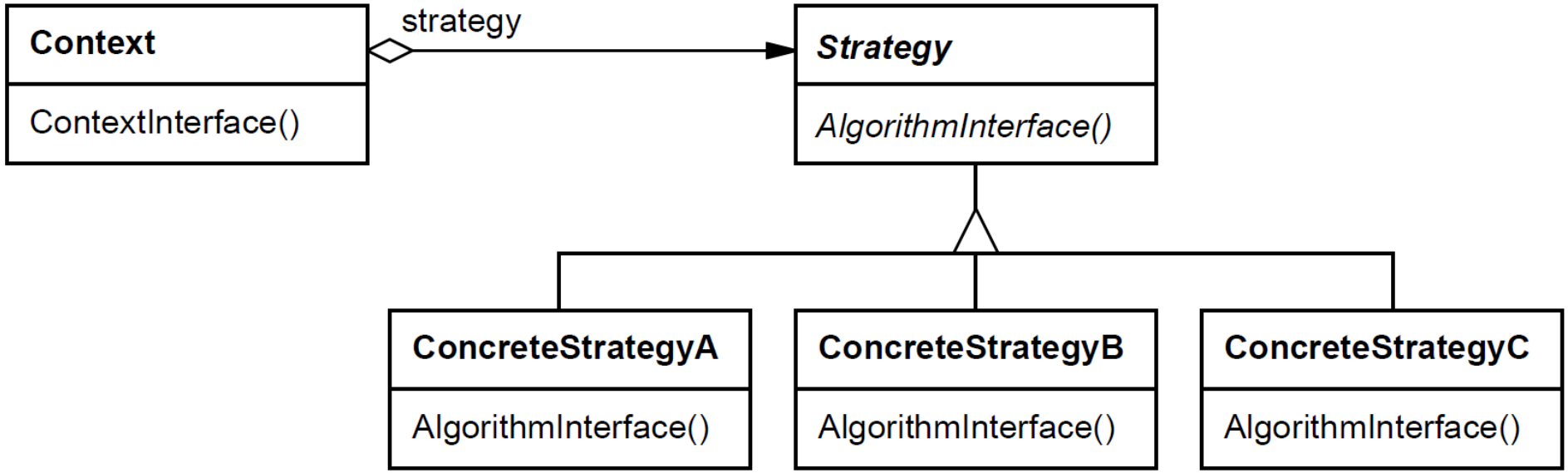
en.wikipedia.org/wiki/Adapter_pattern has more on *Adapter.*

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



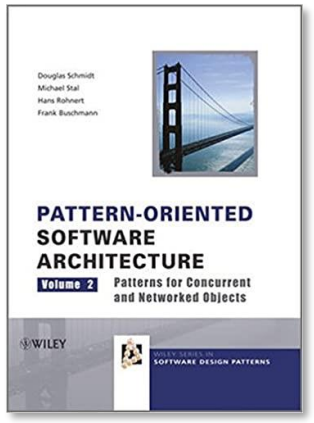


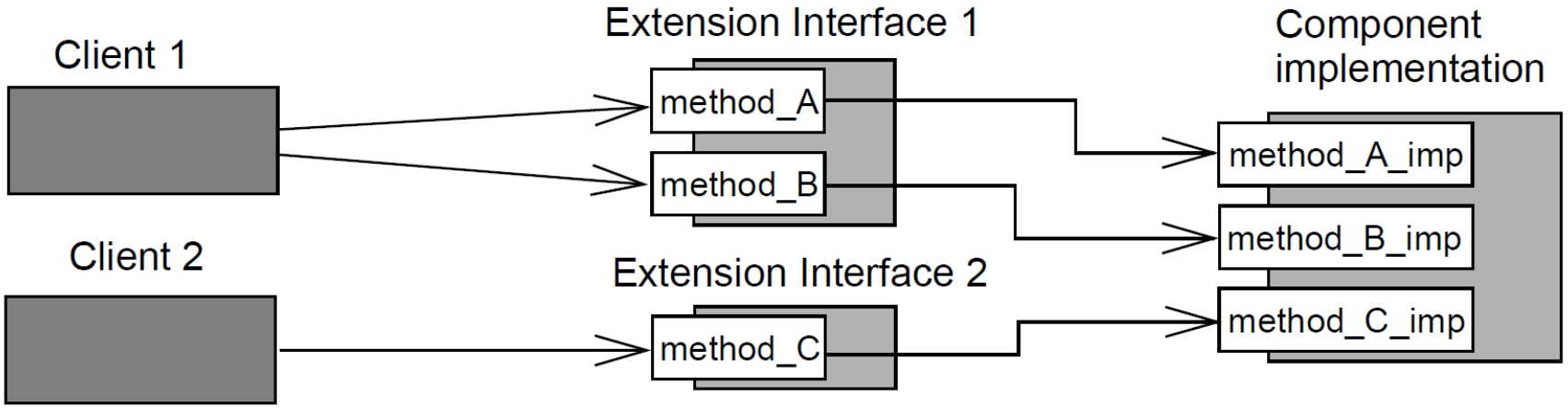
en.wikipedia.org/wiki/Strategy_pattern has more on *Strategy.*

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

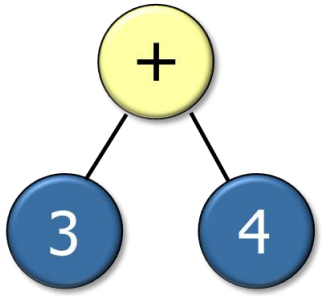




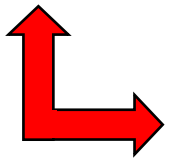
See www.laputan.org/pub/sag/extension-interface.pdf

Implementation considerations

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




```
class Multiply : public Operator {
  ...
  Component_Node *build() {
    return new Composite_Multiply_Node(left.build(),
                                       right.build());
}
```



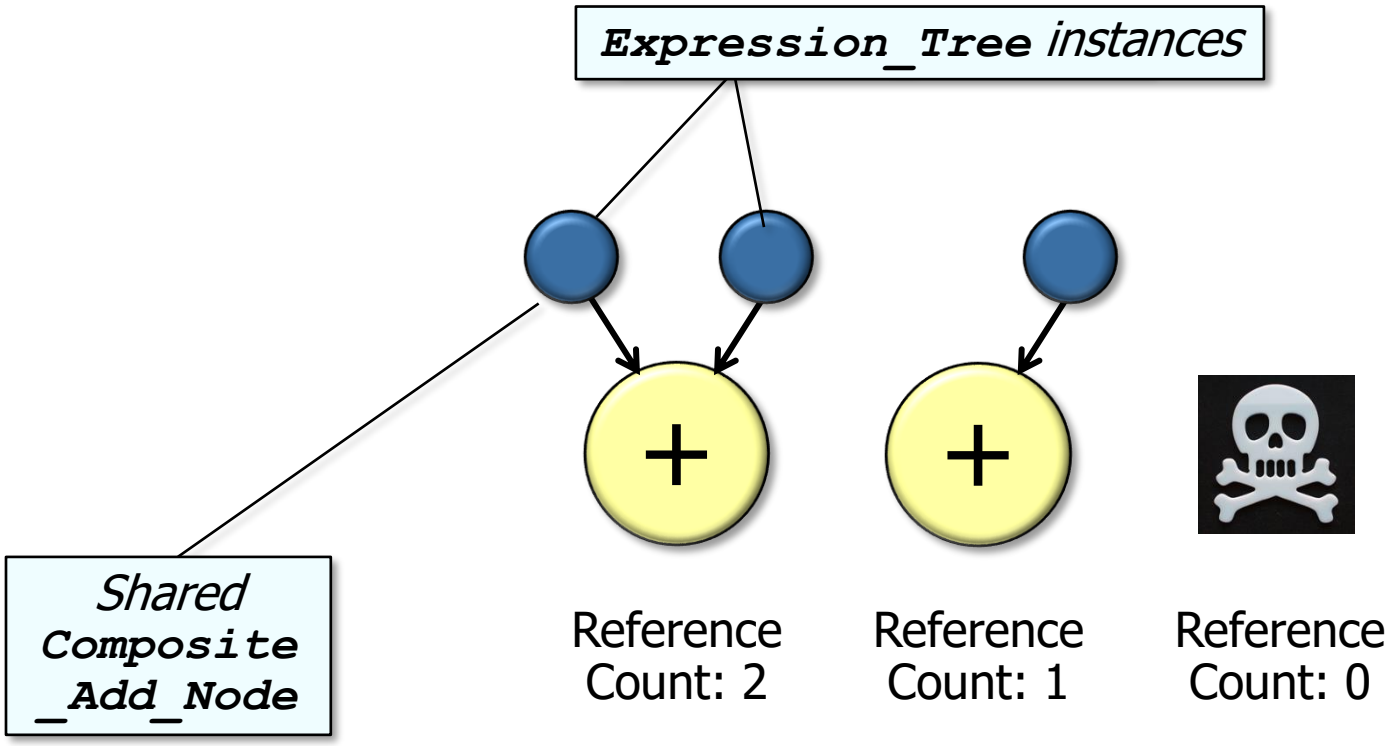
```
class Number : public Expr {
  ...
  Component_Node *build()
  { return new Leaf_Node(item); }
```

Build corresponding component nodes

We'll cover *Builder* later & show how it creates composite expression trees.

Implementation considerations

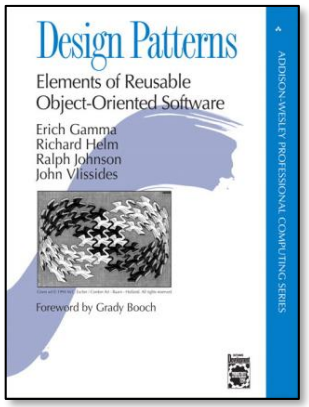
- Sharing implementors & reference counting
  - e.g., C++11/Boost **`shared_ptr`**



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

Implementation considerations

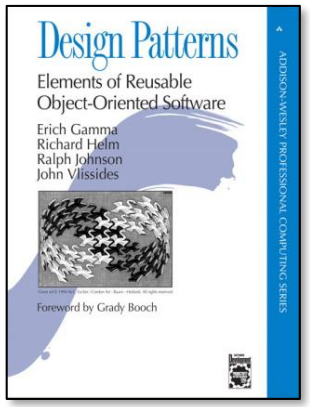
- Dynamic uses of *Bridge* should be implemented via *Decorator.*



en.wikipedia.org/wiki/Decorator_pattern has more on *Decorator.*

Implementation considerations

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

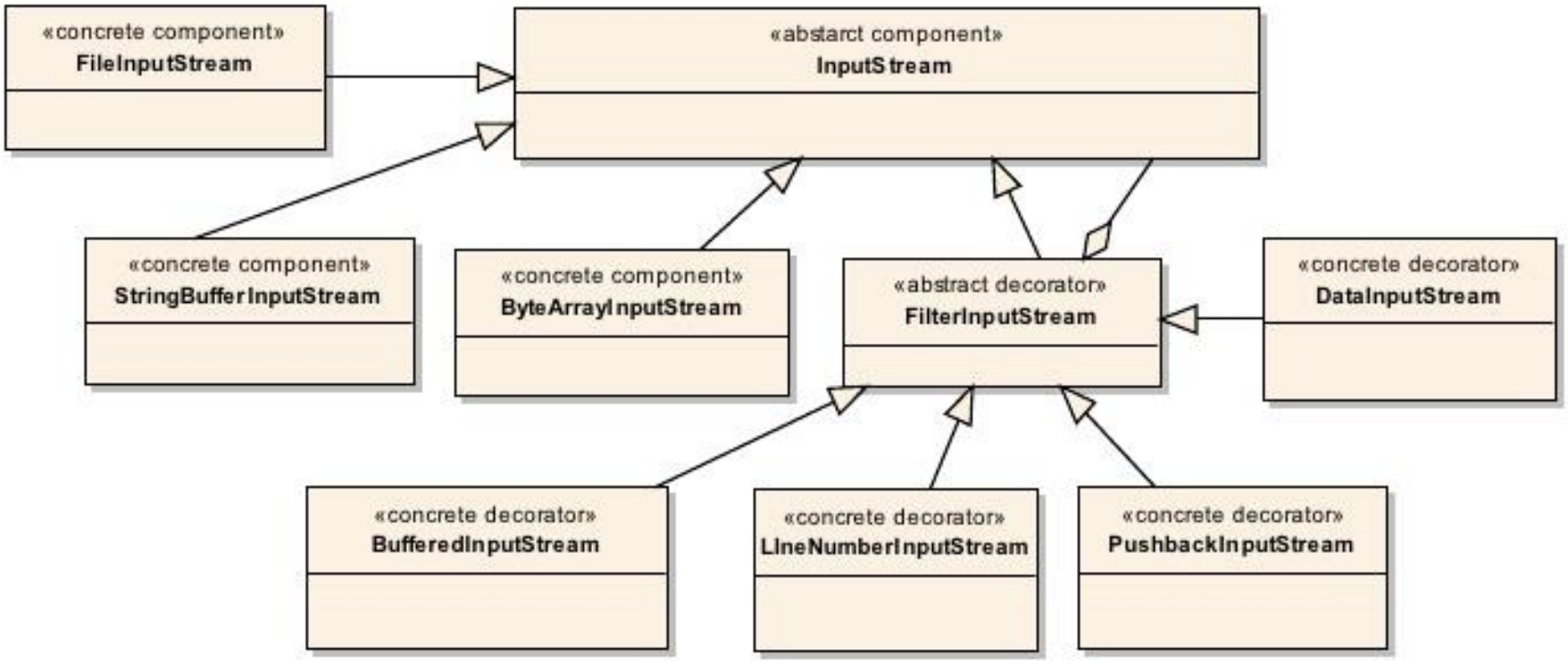


- *Decorator* enables client-specified embellishment of a core object by recursively wrapping it (possibly more than once) dynamically at runtime.

See sourcemaking.com/design_patterns/decorator

Implementation considerations

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

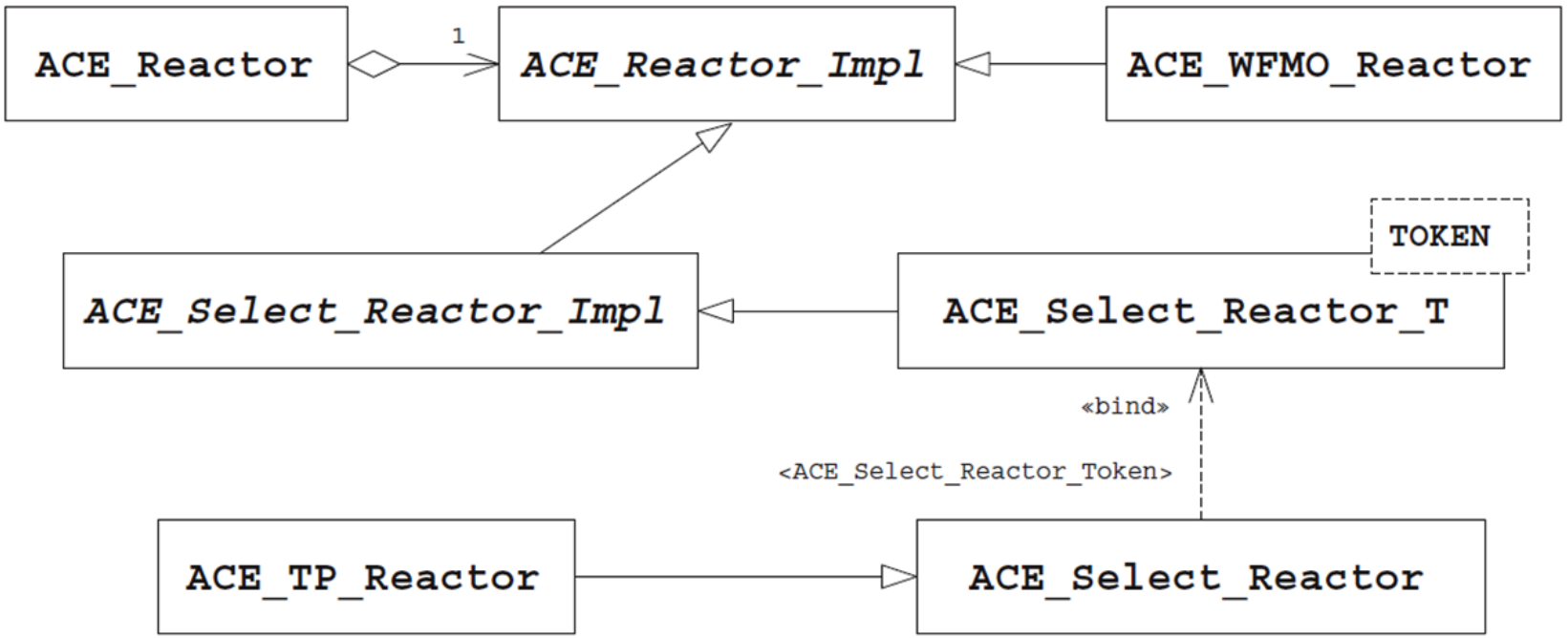


- *Decorator* enables client-specified embellishment of a core object by recursively wrapping it (possibly more than once) dynamically at runtime.
  - Java I/O is a famous example of the *Decorator* pattern.

See kymr.github.io/2016/11/27/Decorator-Pattern

Known uses

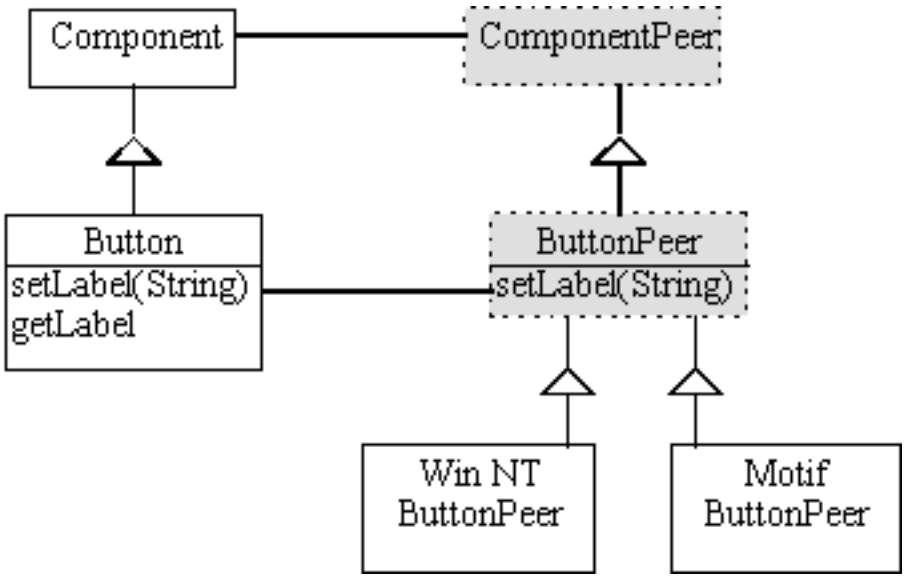
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework



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

Known uses

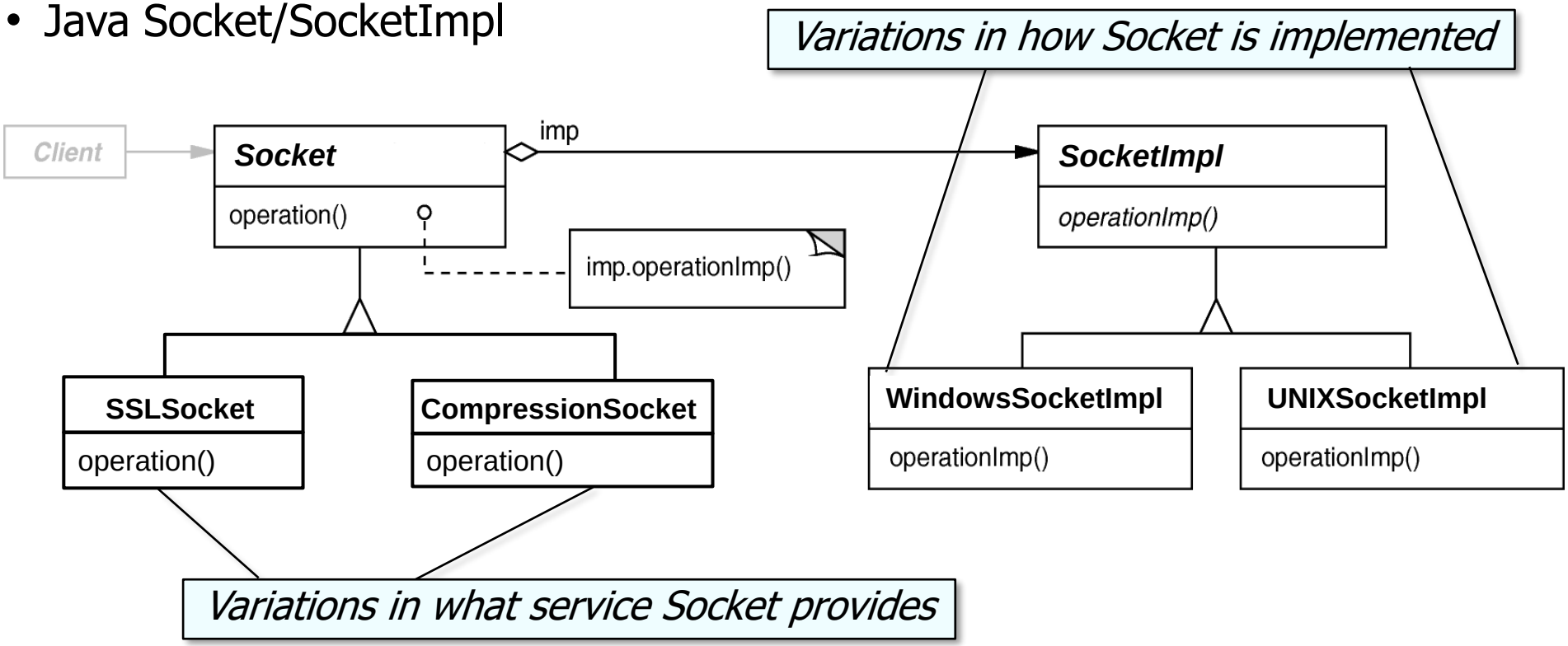
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer



See www.soberit.hut.fi/tik-76.278/group6/awtpat.html

Known uses

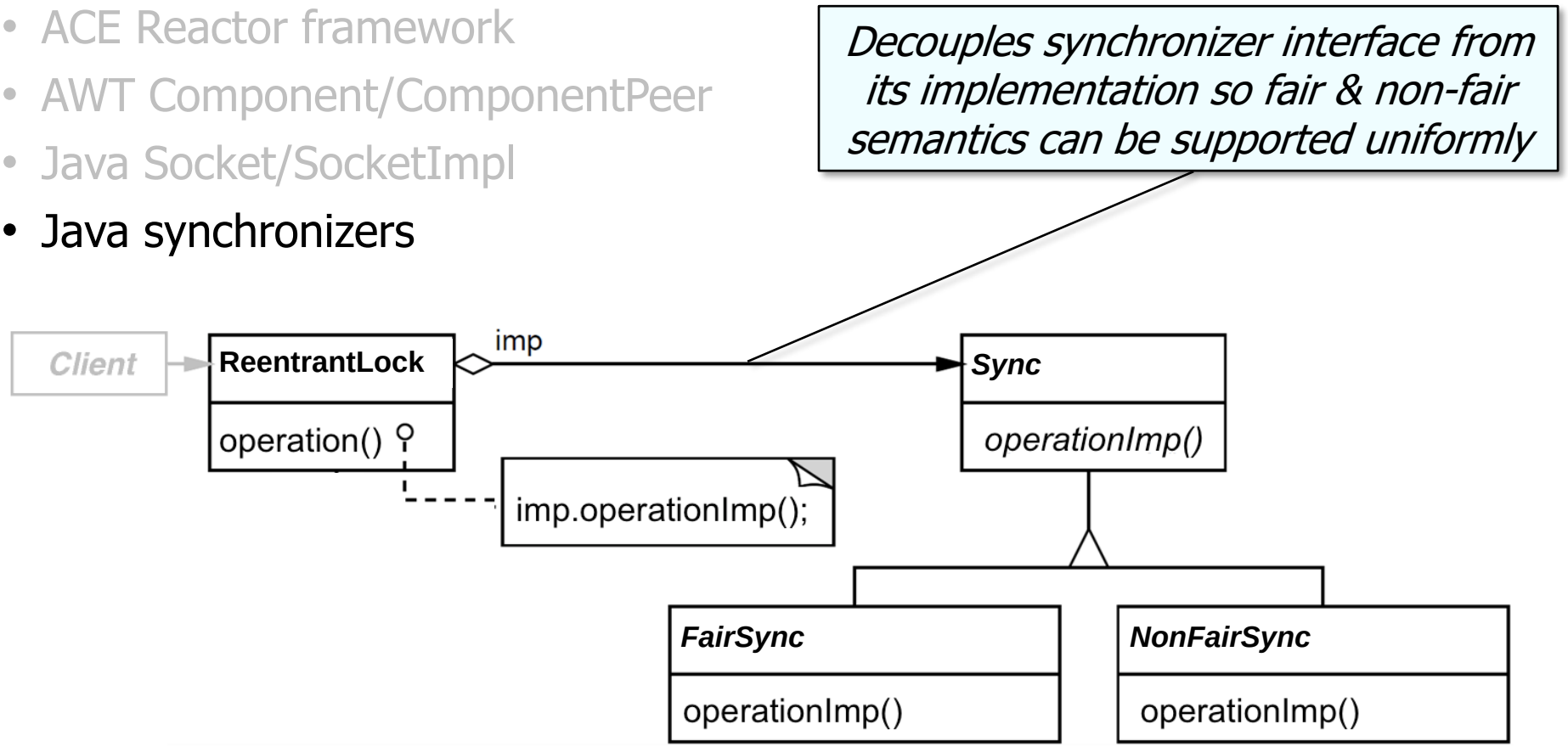
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl



See docs.oracle.com/javase/tutorial/networking/sockets

Known uses

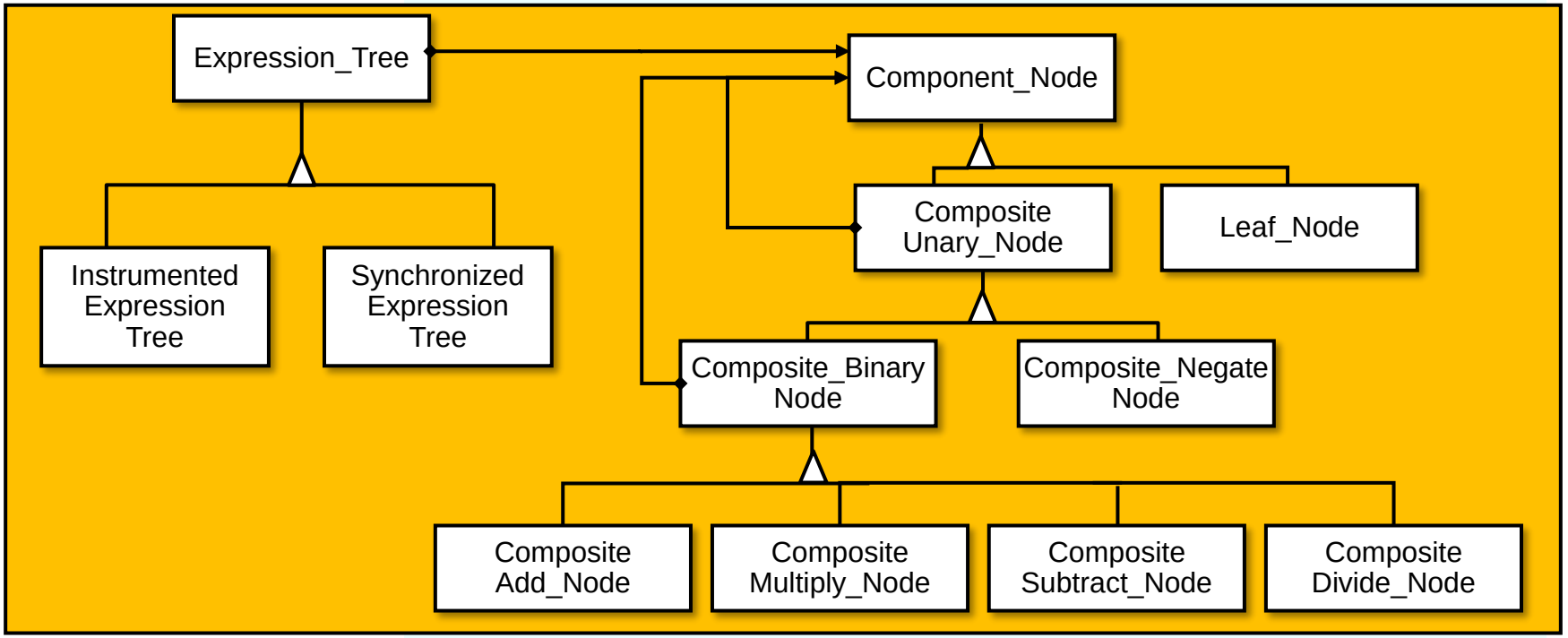
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl
- Java synchronizers



See www.baeldung.com/java-concurrent-locks

Summary of the Bridge Pattern

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



Bridge

Composite

Bridge Composite is an example of a "pattern compound."

See www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf