The Strategy Pattern

Other Considerations

Douglas C. Schmidt

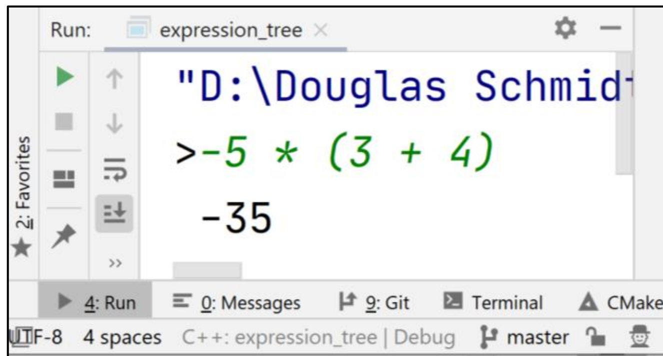
Learning Objectives in This Lesson

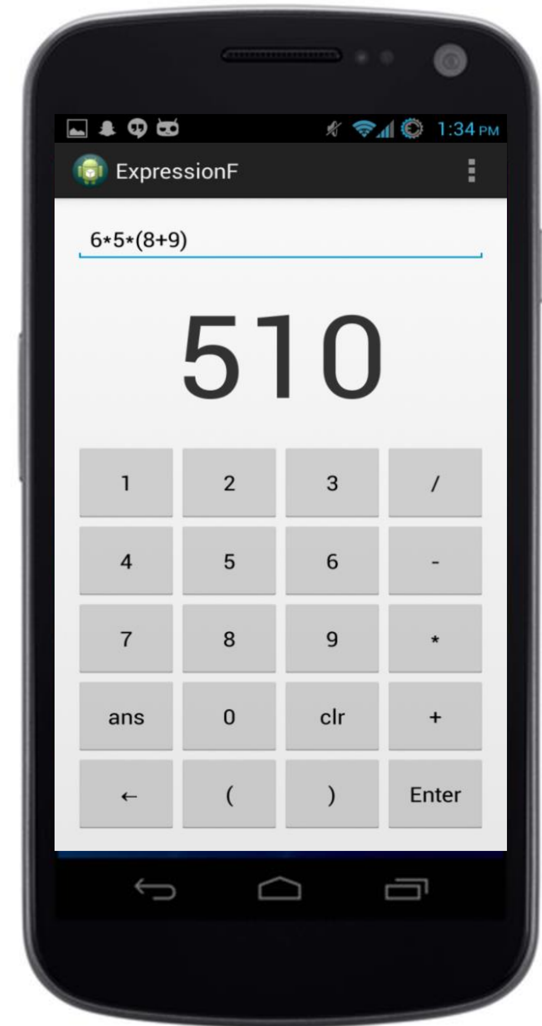
- Recognize how the *Strategy* pattern can be applied in the expression tree processing app to encapsulate variability of algorithm & platform behaviors via common APIs.
- Understand the structure & functionality of the *Strategy* pattern.
- Know how to implement the *Strategy* pattern in C++.
- Be aware of other considerations when applying the *Strategy* pattern.

Consequences

+ Greater flexibility & reuse

  - e.g., by strategizing runtime platform I/O mechanisms, most code can be reused across the Android GUI variant & the command-line variant of the expression tree processing app.

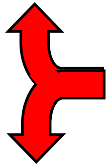




Consequences

+ Behaviors can change dynamically

```
class Expression_Tree {
  ...
  iterator begin (const std::string &traversal_order) {
    return iterator(tree_iterator_factory.make_iterator
                      (*this, traversal_order, false));
  }

  ...
```



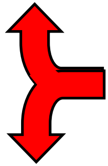
The `tree_iterator_factory.make_iterator()` method enables transparent replacement of different iterator strategies at runtime w/out breaking client code.

```
  for (auto it = expr_tree.begin("in-order");
       it != expr_tree.end("in-order");
       ++it)
    do_something_with_each_node(*it);
```

Consequences

+ Behaviors can change dynamically

```
class Expression_Tree {
  ...
  iterator begin (const std::string &traversal_order) {
    return iterator(tree_iterator_factory.make_iterator
                          (*this, traversal_order, false));
  }
  ...
```



The `tree_iterator_factory.make_iterator()` method enables transparent replacement of different iterator strategies at runtime w/out breaking client code.

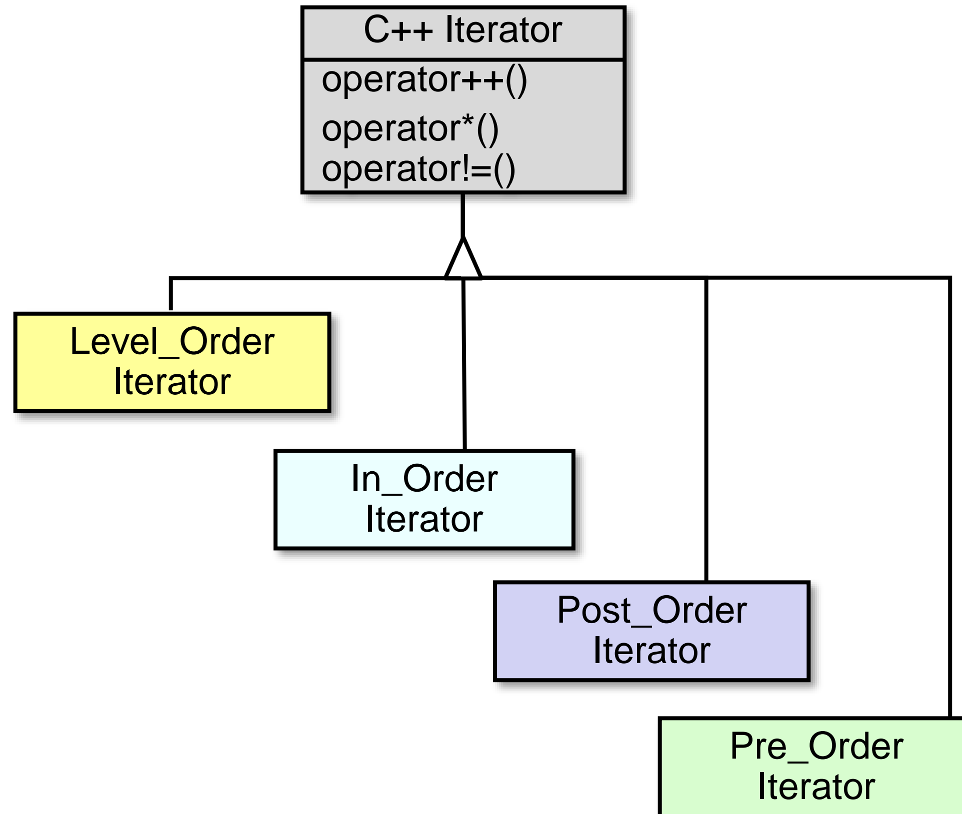
```
for (auto it = expr_tree.begin("post-order");
     it != expr_tree.end("post-order");
     ++it)
  do_something_with_each_node(*it);
```

e.g., can change from "in-order" to "post-order" traversal simply by changing this parameter

Consequences

– Overhead of strategy creation & communication

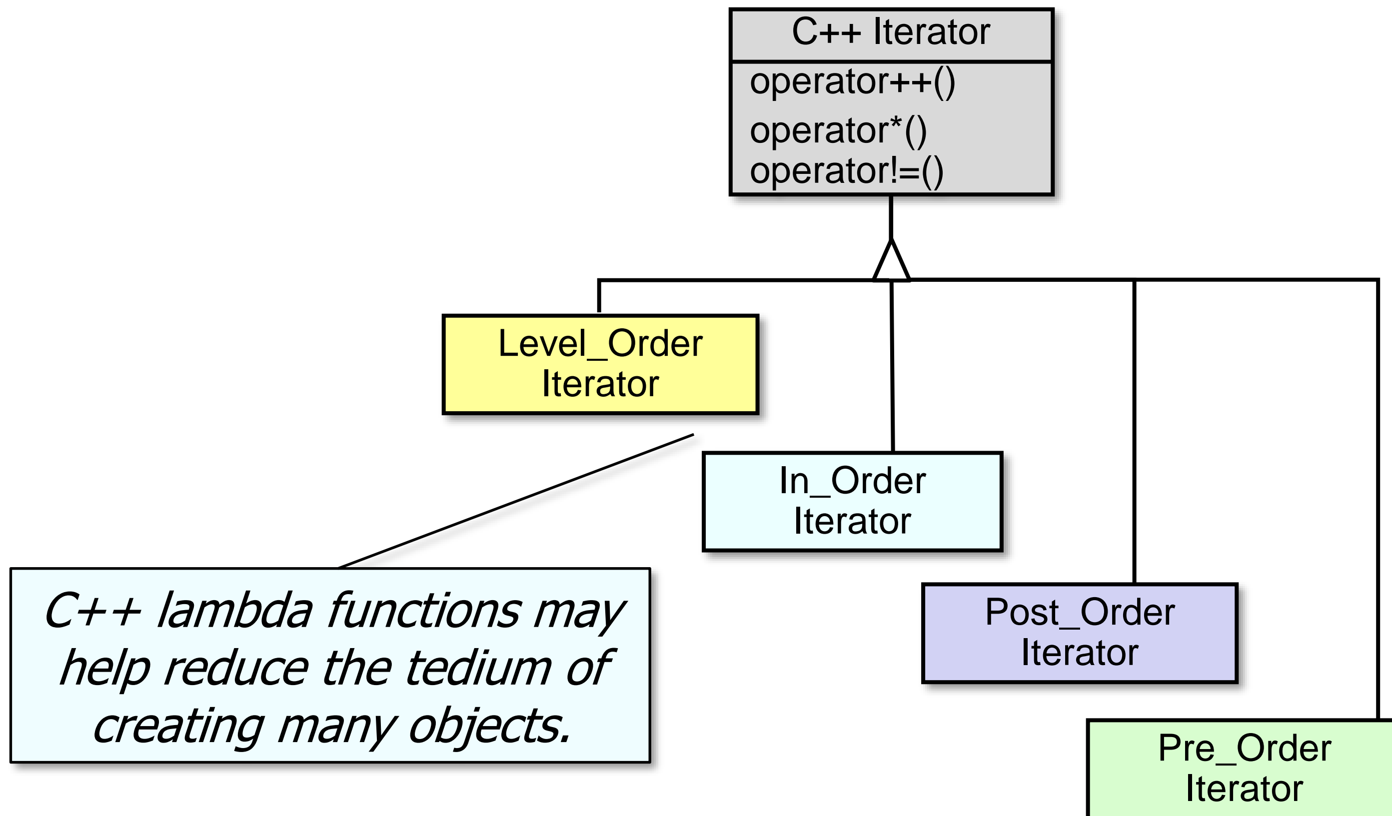
- *Strategy* can increase the number of classes/objects created in a program.



Consequences

– Overhead of strategy creation & communication

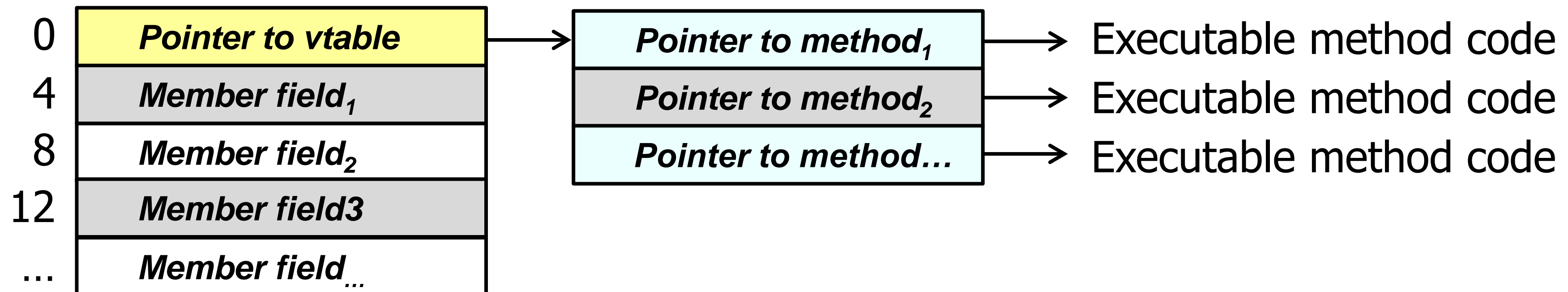
- *Strategy* can increase the number of classes/objects created in a program.



Consequences

– Overhead of strategy creation & communication

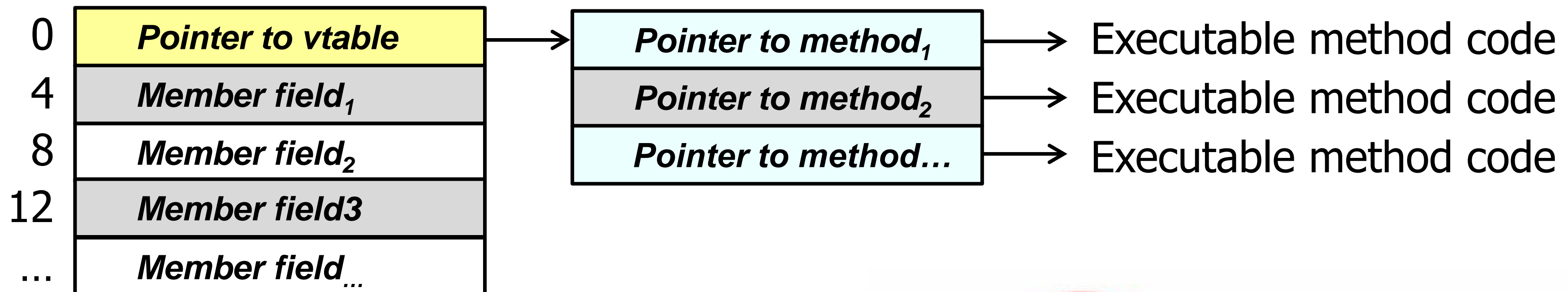
- *Strategy* can increase the number of classes/objects created in a program.
- Dynamically bound implementations of *Strategy* may incur additional virtual method call overhead.



Consequences

– Overhead of strategy creation & communication

- *Strategy* can increase the number of classes/objects created in a program.
- Dynamically bound implementations of *Strategy* may incur additional virtual method call overhead.

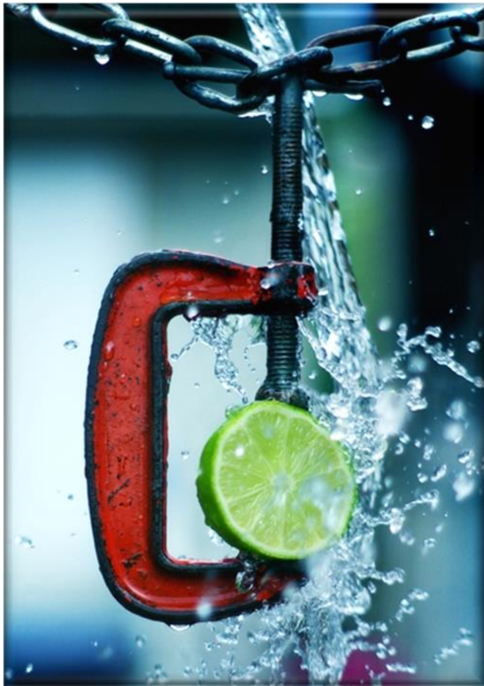


- However, modern C++ compilers optimize virtual function dispatching so it's as efficient as large switch statements or if/else chains.

See lazarenko.me/devirtualization

Consequences

– Inflexible strategy interface

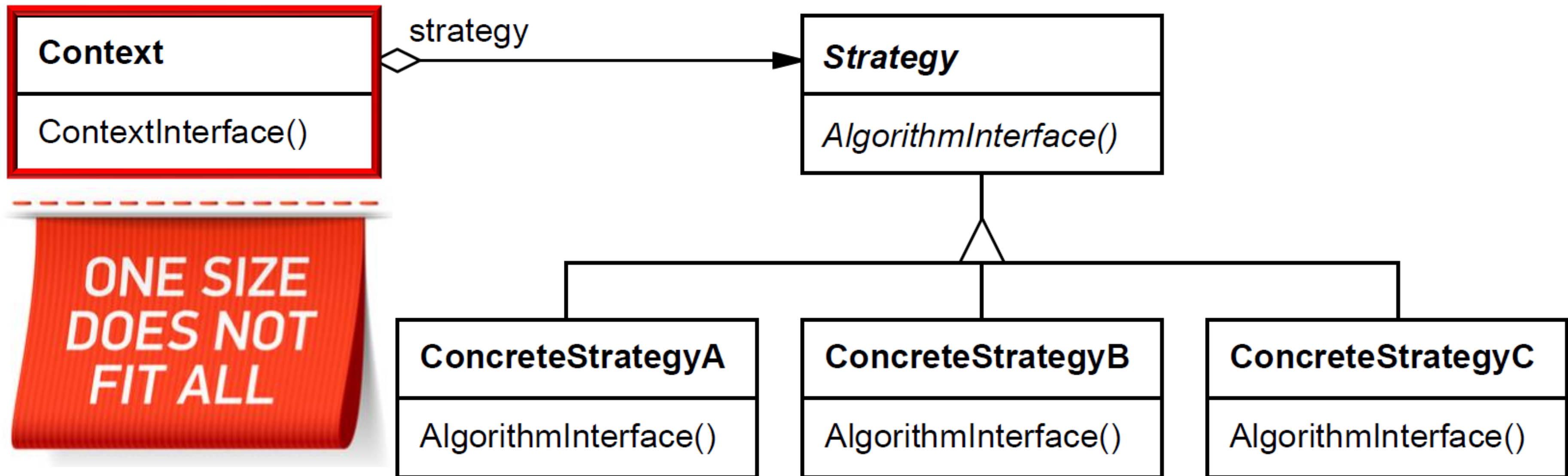


See en.wikipedia.org/wiki/Procrustes#Cultural_references

Consequences

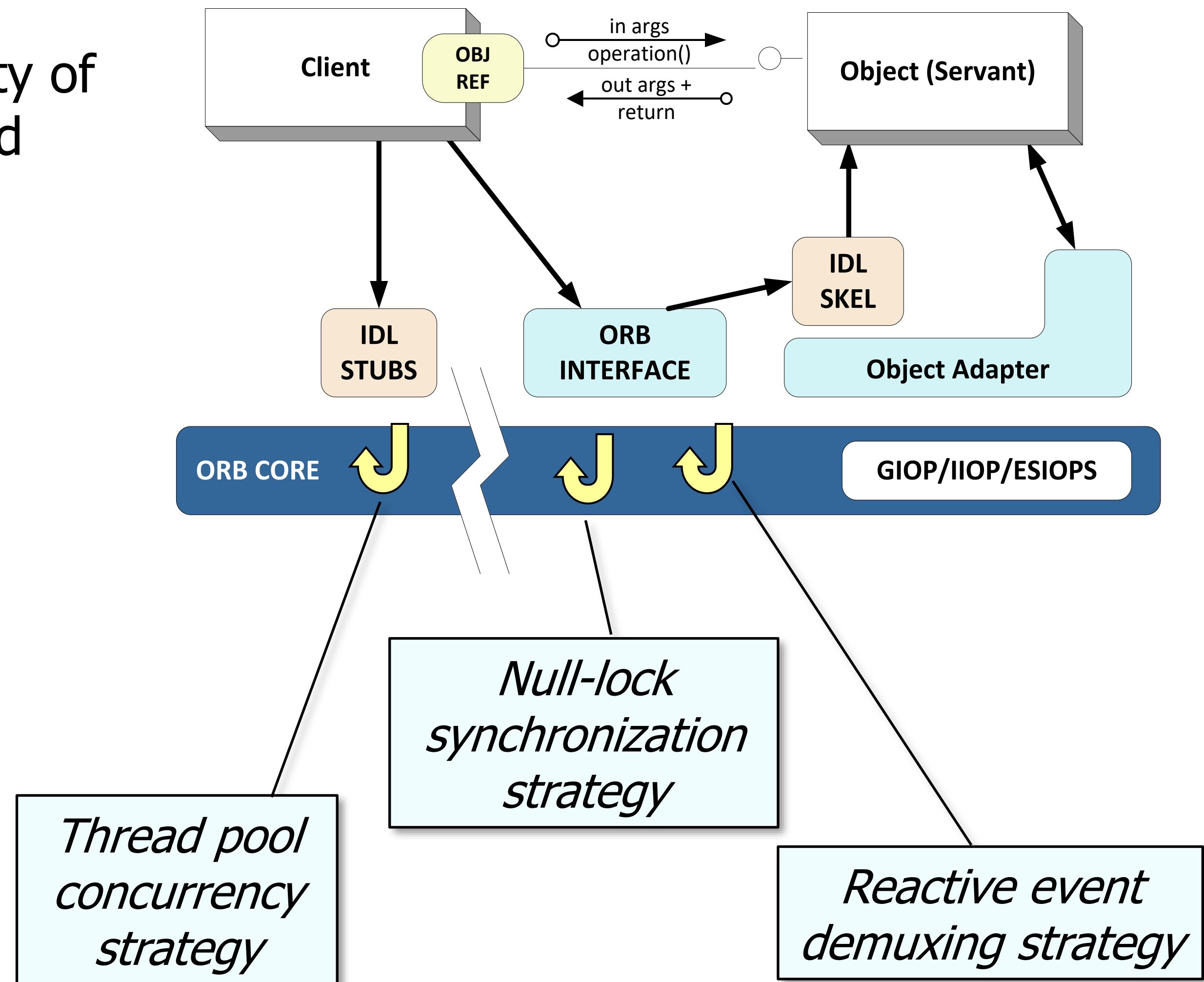
– Inflexible strategy interface

- Motivates need for *Context*, which stores values beyond one-size-fits-all interface



Consequences

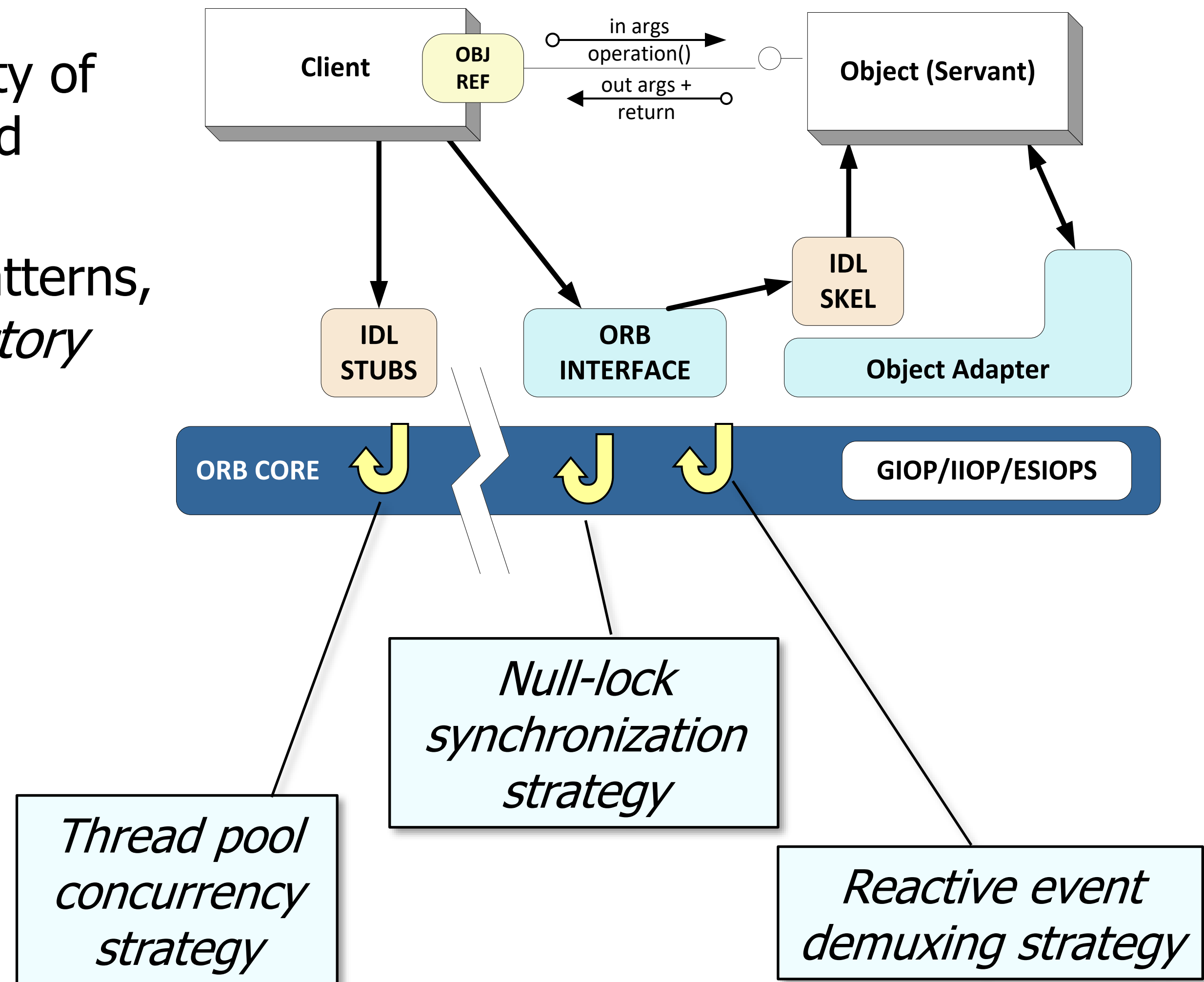
– Semantic incompatibility of multiple strategies used together inconsistently



See www.dre.Vanderbilt.edu/~schmidt/PDF/ORB-patterns.pdf

Consequences

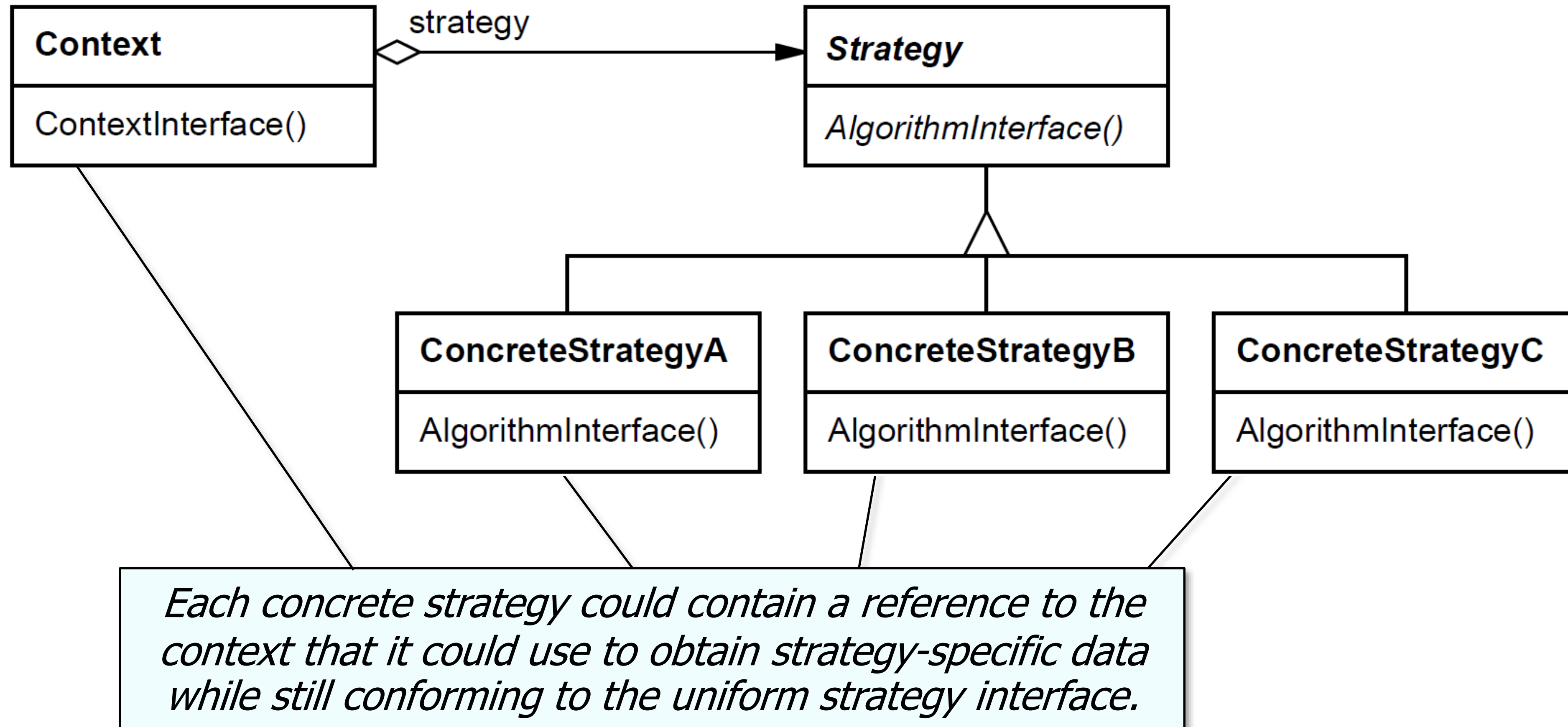
– Semantic incompatibility of multiple strategies used together inconsistently
  - May require other patterns, such as *Abstract Factory*



See en.wikipedia.org/wiki/Abstract_factory_pattern

Implementation considerations

- Exchanging information between a strategy & its context



Implementation considerations

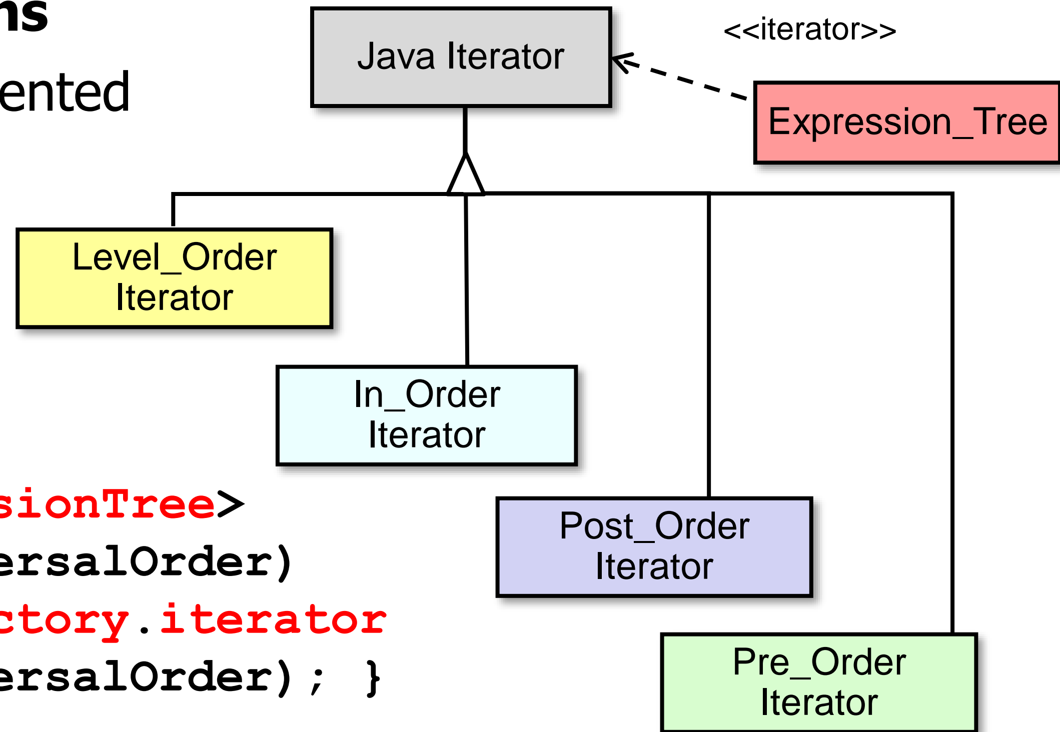
- Static binding of strategy selection
  - e.g., via Java generics or C++ parameterized types

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);

...
std::vector<int> v ({1, 6, 2, 8, 3, 9});

std::sort (v.begin (), v.end (), std::greater<int>());
```

See en.wikipedia.org/wiki/Policy-based_design for “compile-time” strategies.

Implementation considerations

- Strategies in Java often implemented with interfaces & factories
  - Rather than using the *Bridge* pattern

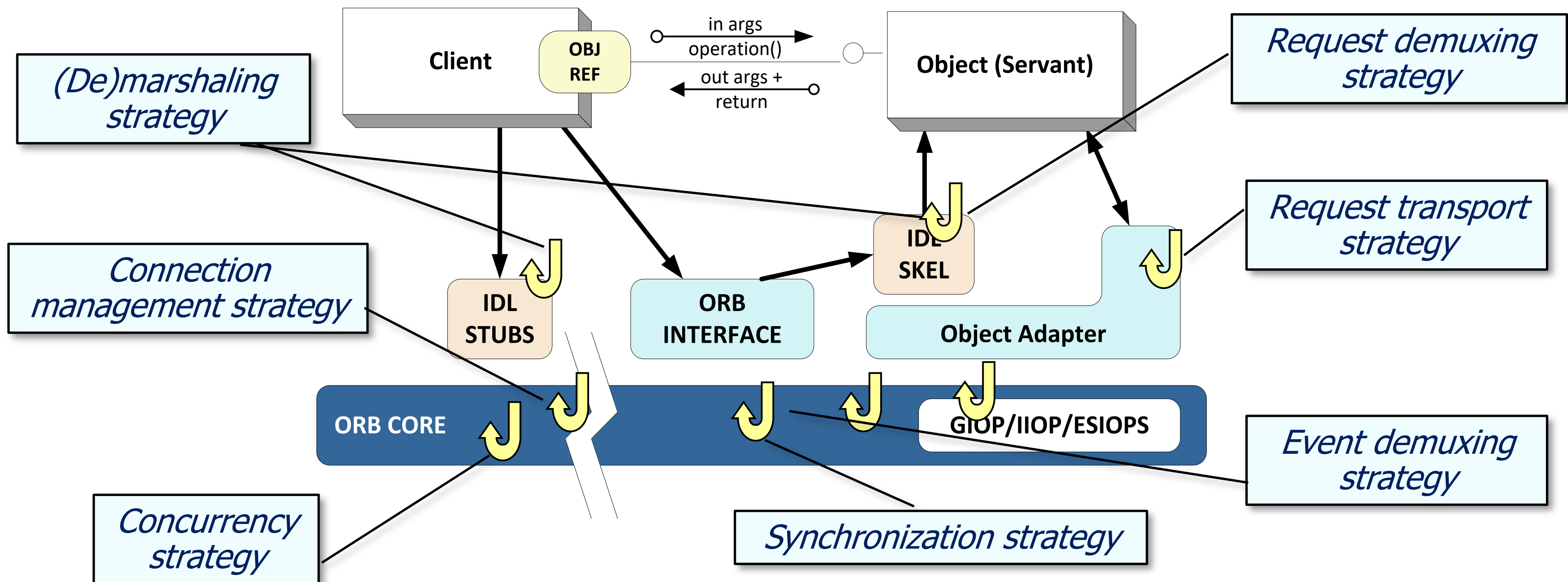



```
class ExpressionTree {
  ...
  public Iterator<ExpressionTree>
    iterator(String traversalOrder)
    { return mIteratorFactory.iterator
               (this, traversalOrder); }
}
```

Java's support for garbage collection often obviates the need for *Bridge.*

Known uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) real-time object request broker middleware



www.dre.vanderbilt.edu/~schmidt/PDF/ORB-patterns.pdf has more information.

Known uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) real-time object request broker middleware
- C++ Standard Template Library (STL)
  - *Strategy* can be applied to more than "algorithms"

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
           RandomAccessIterator last,
           Compare comp);

...
std::vector<int> v ({1, 6, 2, 8, 3, 9});

std::sort (v.begin (), v.end (), std::greater<int>());
```

Comparison strategy (functor)

See en.wikipedia.org/wiki/Function_object#In_C_and_C++

Known uses

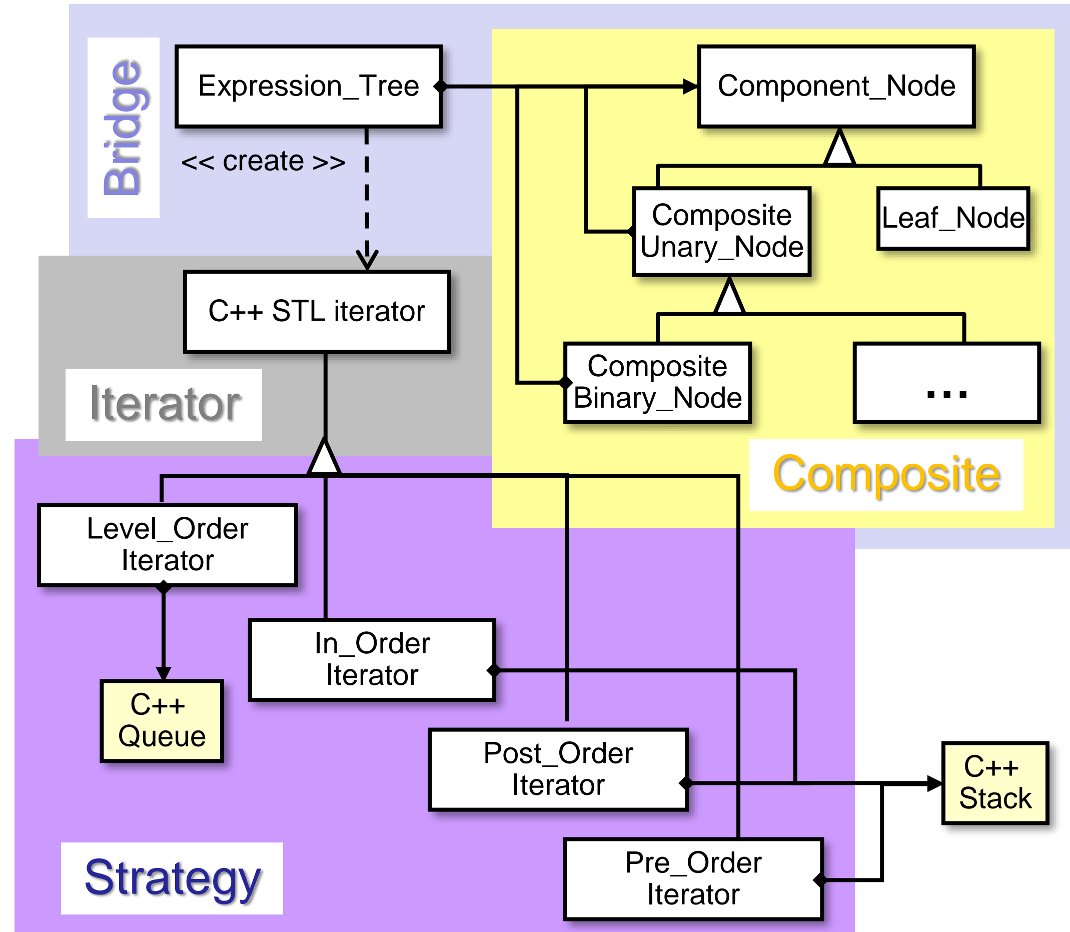
- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) real-time object request broker middleware
- C++ Standard Template Library (STL)
- Java JDK class libraries

```
String[] nameArray = {"Barbara", "James", "Mary", "John",
                      "Robert", "Michael", "Linda", "james", "mary"};

Arrays.sort(nameArray, String::compareToIgnoreCase);
```

See en.wikipedia.org/wiki/Function_object#In_Java

Summary of the Strategy Pattern

- *Strategy* encapsulates the variability of behaviors via a common API whose implementations can be changed transparently with respect to clients.



Strategy decouples the interface of a behavior from its implementations.