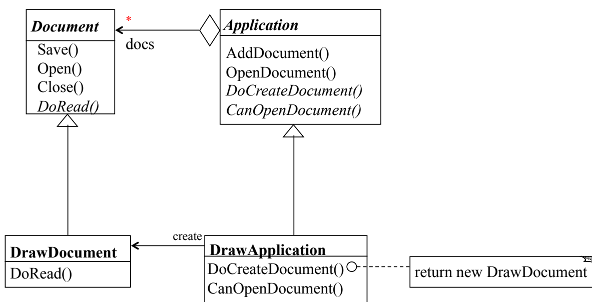
More Go4
Design Patterns

Template Method: Motivation

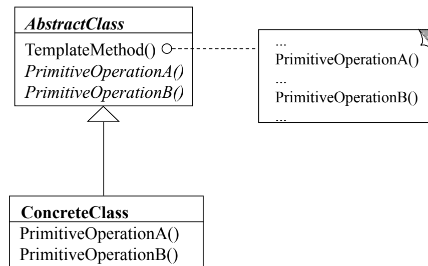
- **A template method specifies an algorithm whose steps can be overridden by subclasses**
  - » **defines the ordering but lets Application and Document subclasses vary individual steps**

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }
    Document* doc = DoCreateDocument();
    if (doc) {
        _docs->AddDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```


Template Method: Example



Template Method: Structure



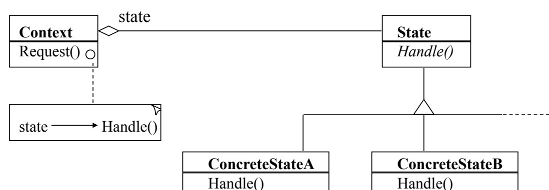
Template Method: Consequences

- Inverted control structure.
- Types of operations called by the template method
  - concrete AbstractClass operations
  - primitive operations (must be overridden)
  - factory methods (ie for creating objects)
  - hook operations (may be overridden)
- C++ access control
  - primitive operations as protected members
    - » only template method can call them, and
    - » are pure virtual.
  - template method should be a non virtual member function.
- Perversion: too many primitive operations that are overridden:
  - Must aim to minimize number of primitive operations that must be overridden.

The 'Infamous' State Pattern

State Pattern

From Gamma et al.



You must consider cost of instance creation/destruction, frequency of state changes, etc.

Efficient FSMs

- The State pattern as the recommended starting point??
  - Because other representations of FSMs are much less amenable to change
- CASE structures:
  - if states/events are implemented as objects, they need to be assigned indices for the CASE statement to be real-time efficient
  - from GSF: should be used only when the constraints on real-time performance and memory usage are critical
- Table Lookup:
  - still need indices
  - slightly slower than CASE structures
  - from GSF: should be used when real-time performance is a primary concern and the FSM is small and simple (i.e., not hierarchical)
- Double Dispatch:
  - states and events must be implemented as objects
  - slowest approach
  - from GSF: only use when run-time performance is not critical. Technique allows a high degree of flexibility and reusability, which is specially advantageous for complex FSMs.

Achieving Performance

- There is a fundamental trade-off between performance and evolution:
  - Static techniques (such as table look-up and CASE statements) and optimizations are typically more difficult to evolve than dynamic ones.
- Two *rules*:
  - Reduce the overall number of messages:
    - » this is easier said then done...
  - Run code and test performance as early as possible
- What to look for:
  - frequent messages (in particular, those that carry lots of data)
  - excessive data processing in senders or receivers due to ill-conceived data representation (typically too general):
  - passing by value rather than by reference or pointer
  - excessive creation and destruction of instances

© J.-Pierre Corriveau, 1997- present

More on Inefficient Data Access

Symptoms:

- Lots of messages used only to access data
- Unnecessary restructuring of the same data for different customers
- Excessive data deciphering in receiver
- Data organized for just-in-case rather than *actual* needs.

Issues:

- Is it OK to violate encapsulation to improve speed of access?
- Should you customize data representation for frequent/critical users?
- Should you write fast customized procedures (as opposed to slower general ones) even though they are used infrequently?
- Should you cache or (re)compute the data?
- Does the data really belong to this object?

© J.-Pierre Corriveau, 1997- present

Performance Heuristics

- Explicit your performance requirements and memory constraints!
  - use timing constraints à la UML
  - use (preferably automatic) performance modeling and metrics
- Consider the *frequency* of use-cases and of their corresponding sequences of messages.
  - don't handle the worst cases in such a way that the more frequent sequences are inefficient!
  - consider optimizing most frequently used methods
  - consider collapsing together objects that interact too much...
- Avoid excessive delegation.
- Revisit your data packaging:
  - understand the pros and cons of multiple copies of the same data

© J.-Pierre Corriveau, 1997- present

About OOPLs

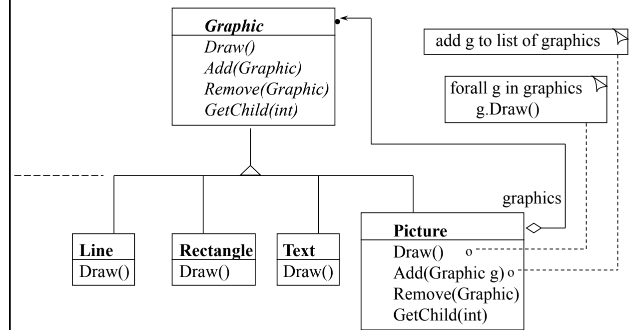
- Know your language:
  - know about the cost of a procedure call
  - understand the cost of the features of an OOPL (e.g., RTTI)
  - Understand inlining and friends in C++
- Know how and when to use primitives!
- Typical sources of slow-downs:
  - dynamic typing (i.e., variables declared without a type)
  - creation and destruction of instances
  - dynamic binding
    - » but virtual functions have *constant* overhead in C++
  - conversions and casting
  - call by value
  - class/equality checking statements
  - slow data structures in libraries

© J.-Pierre Corriveau, 1997- present

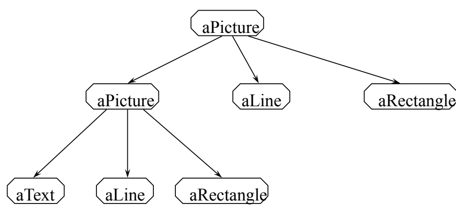
Structural vs Behavioral Pattern

Solution in the organization of classes
vs
Solution in the operations of a class

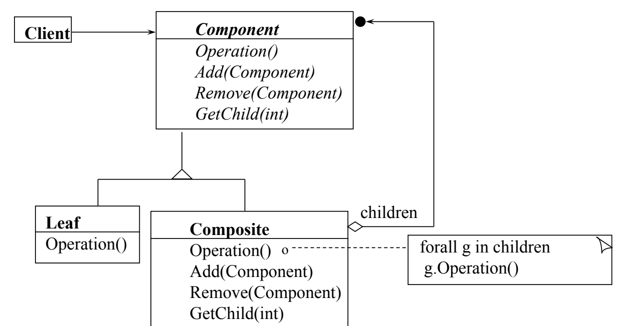
Composite Pattern



Object Structure



Composite Structure



Composite Consequences

- makes the client simpler because it can treat the composites and primitives uniformly. This avoids case statements on the type of the component.
- easier to add new kinds of components
- can't have the type system help in restricting components of a composite, but have to use run-time type checks instead

- Beware: implementation considerations are NOT trivial!!

© J.-Pierre Corriveau, 1997- present

Pattern Hatching

(see book by J. Vlissides)

Bottom line: Remember Alexander's philosophy!

Step 1: do an inventory of common practices

Step 2: allow discrimination between alternatives by analyzing the forces of patterns

force: +/- wrt FR and NFR requirements

The sad reality is that, 20+ years after G04, we're still at step 1…

© J.-Pierre Corriveau, 1997- present

Iterators (1)

- Iterators allow applications to loop through elements of some ADT *without* depending on knowledge of its implementation details.
  - There are a number of different techniques for implementing iterators, each having advantages and disadvantages.
- Design issues:
  - providing a copy of each data item vs. providing a reference to each data item
  - handling concurrency and insertion or deletion while iterator(s) are running
- There are three primary methods of designing iterators:
  1. Pass a pointer to a function
     - Not very OO... we avoid stand-alone functions.
  2. Use in-class iterators (a.k.a. passive or internal iterators)
     - requires modification of class interface
  3. Use out-of-class iterators (a.k.a. active or external iterator)
     - handles multiple simultaneously active iterators on the same instance
     - may require special access to original class internals, usually using friends

© J.-Pierre Corriveau, 1997- present

Iterators (2)

Pointer to function iterator

```
template <class T>
class Vector {
public:
/* missing details */
int apply (void (*ptf) (T &)) {
for (int i = 0; i < mysize; i++)
(*ptf) (buffer[i]); //call the function
}
};
void f (int& i) { cout << i << endl; }
Vector<int> v (100);
// ...
v.apply (f);
```

- We need to add the function "apply" to the interface of Vector.
- And the argument of "apply" is a pointer to a function that returns void and must match T…

© J.-Pierre Corriveau, 1997- present

Iterators (3)

In-class iterator

```
template <class T>
class Vector {
public:
/* missing details */
void reset (void) { i = 0; }
bool advance (void) { return i++ < mysize); }
T value (void) { return buffer[i - 1]; }
private:
/* missing details */
int i; //holds the single current position
};
Vector<int> v (100);
// ...
for (v.reset (); v.advance () != false; )
cout << "value = " << v.value () << "\n";
```

- **We had to add *reset* and *advance* to the interface of Vector.**
- **There is an implicit order to the use of *reset* and *advance*.**

© J.-Pierre Corriveau, 1997- present

Iterators (4)

Out-of-class iterator

```
#include "Vector.h"
template <class T>
class Vector_Iterator {
public:
Vector_Iterator (const Vector<T> &v): i (0), vr (v) {}
bool advance (void) { return i++ < vr.size ();}
T value (void) { return vr[i - 1]; }
private:
Vector<T> &vr;
int i;
};
Vector<int> v (100);
Vector_Iterator<int> iter (v), iter2 (v);
while (iter.advance () != false)
cout << "value = " << iter.value () << "\n";
```

- Because Vector has a [] operator and a size function, no need for friends.
- Inlining improves performance and is better than friends.
- You should check out the STL!!! (Crucial for C++ jobs!)

© J.-Pierre Corriveau, 1997- present