Object-Oriented Patterns & Frameworks
Assignment 4b Patterns

Dr. Douglas C. Schmidt

d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of EECS
Vanderbilt University
Nashville, Tennessee
Managing Global Objects Effectively

Goals:
- Centralize access to objects that should be visible globally, e.g.:
  - command-line options that parameterize the behavior of the program
- The object (Reactor) that drives the main event loop

Constraints/forces:
- Only need one instance of the command-line options & Reactor
- Global variables are problematic in C++

% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
  > format in-order
  > expr 1+4*3/2
  > eval post-order
  7
  > quit
  % tree-traversal
  > 1+4*3/2
  7
Solution: Centralize Access to Global Instances

Rather than using global variables, create a central access point to global instances, e.g.:

```cpp
int main (int argc, char *argv[])
{
    // Parse the command-line options.
    if (!Options::instance ()->parse_args (argc, argv))
        return 0;

    // Dynamically allocate the appropriate event handler
    // based on the command-line options.
    Expression_Tree_Event_Handler *tree_event_handler =
        Expression_Tree_Event_Handler::make_handler
            (Options::instance ()->verbose ());

    // Register event handler with the reactor.
    Reactor::instance ()->register_input_handler
        (tree_event_handler);
    // ...
```
**Singleton**

**Object Creational**

**Intent**

Ensure a class only ever has one instance & provide a global point of access.

**Applicability**

- When there must be exactly one instance of a class, & it must be accessible from a well-known access point.
- When the sole instance should be extensible by subclassing, & clients should be able to use an extended instance without modifying their code.

**Structure**

```
Singleton
static instance() { singletonOperation() getSingletonData() }
static uniqueInstance singletonData

If (uniqueInstance == 0)
    uniqueInstance = new Singleton;
return uniqueInstance;
```
Singleton

Consequences
+ reduces namespace pollution
+ makes it easy to change your mind & allow more than one instance
+ allow extension by subclassing
- same drawbacks of a global if misused
- implementation may be less efficient than a global
- concurrency pitfalls strategy creation & communication overhead

Implementation
- static instance operation
- registering the singleton instance
- deleting singletons

Known Uses
- Unidraw's Unidraw object
- Smalltalk-80 ChangeSet, the set of changes to code
- InterViews Session object

See Also
- Double-Checked Locking Optimization pattern from POSA2
- “To Kill a Singleton”
  www.research.ibm.com/designpatterns/pubs/ph-jun96.txt
**Strategy**

**Consequences**
+ greater flexibility, reuse
+ can change algorithms dynamically
- strategy creation & communication overhead
- inflexible Strategy interface
- semantic incompatibility of multiple strategies used together

**Implementation**
- exchanging information between a Strategy & its context
- static strategy selection via parameterized types

**Known Uses**
- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) Real-time CORBA middleware

**See Also**
- Bridge pattern (object structural)
Strategy: object behavioral

**Intent**

define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently

**Applicability**

- when an object should be configurable with one of many algorithms,
- *and* all algorithms can be encapsulated,
- *and* one interface covers all encapsulations

**Structure**

```
Context (Composition)
  contextInterface()

Strategy (Compositor)
  algorithmInterface()

ConcreteStrategyA
  algorithmInterface()

ConcreteStrategyB
  algorithmInterface()

ConcreteStrategyC
  algorithmInterface()
```
Factory Method

class creational

Intent

Provide an interface for creating an object, but leave choice of object’s concrete type to a subclass

Applicability

when a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

Structure
Pattern & Framework Tutorial

Douglas C. Schmidt

**Factory Method**

**Consequences**

+ By avoiding to specify the class name of the concrete class and the details of its creation, the client code has become more flexible.
+ The client is only dependent on the interface.
- Construction of objects requires one additional class in some cases.

**Implementation**

• There are two choices here:
  - The creator class is abstract and does not implement creation methods (then it **must be subclassed**).
  - The creator class is concrete and provides a default implementation (then it **can be subclassed**).

• Should a factory method be able to create different variants? If so, the method must be equipped with a parameter.

**Known Uses**

- InterViews Kits
- ET++
  - WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)
- BREW
- UNIX open() syscall
Abstract Factory

Intent
create families of related objects without specifying subclass names

Applicability
when clients cannot anticipate groups of classes to instantiate

Structure
Abstract Factory

**Consequences**
- + flexibility: removes type (i.e., subclass) dependencies from clients
- + abstraction & semantic checking: hides product’s composition
- - hard to extend factory interface to create new products

**Known Uses**
- InterViews Kits
- ET++
- WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)

**Implementation**
- parameterization as a way of controlling interface size
- configuration with Prototypes, i.e., determines who creates the factories
- abstract factories are essentially groups of factory methods
**Intent**

Separate a (logical) abstraction interface from its (physical) implementation(s)

**Applicability**

- When interface & implementation should vary independently
- Require a uniform interface to interchangeable class hierarchies

**Structure**
Bridge

**Consequences**
- Abstraction interface & implementation are independent
- Implementations can vary dynamically
- Can be used transparently with STL algorithms & containers
- One-size-fits-all Abstraction & Implementor interfaces

**Implementation**
- Sharing Implementors & reference counting
  - See reusable `Refcounter` template class (based on STL/boost `shared_pointer`)
- Creating the right Implementor (often use factories)

**Known Uses**
- ET++ Window/WindowPort
- libg++ Set/ {LinkedList, HashTable}
- AWT Component/ComponentPeer