Object-oriented design

• What are the goals of OO software design?
• Who needs it?
• How does OO design try to achieve its goals?
• How well does it achieve those goals?

Goal of object-oriented design

• Maintainability
• Extensibility
• Reuse

Postulate: The key to all of these is to design using loosely coupled modules.
Corollary: Designers must focus on how pieces of their designs are coupled.
Kinds of coupling

- Encapsulation

```cpp
class Employee {
    std::string m_name;
};
```

- Inheritance

```cpp
class Manager : public Employee
```

- Simple use (as client)

```cpp
Account acct1;
acct1.deposit(1000000);
```

- Template instantiation

```cpp
std::map< std::string, int > phone_dir;
```

Keeping coupling loose

- Separate interface and implementation.
- Keep implementations private.
- Provide the “right” interface - simple, yet complete.
- Use inheritance to provide a layer of abstraction between modules.
- Maintain invariants carefully.
**OO design patterns**

- There are common idioms in any design methodology.
- Learning a design methodology is mostly learning these idioms and when to use them.
- The same can be said of a programming language.
- A good exposition of OO design patterns is “Design Patterns” by “the gang of 4.”

**Patterns you’ve already seen**

- Iterators
- Abstract factory functions
- Mix-ins (a.k.a. Decorators)
Creation patterns

• One of the most important forms of coupling involves creation.
• Creation is also somewhat more difficult to do abstractly (there’s no such thing as a virtual constructor, why not?)

Pattern 1: Singleton

• Need a single instance of some class for use by many modules (similar to a global variable)
• Lifetime of this instance is for the entire program.
• May want to extend this by subclassing later.
• Want to do that without changing client code.
• Solution: use static members.
Implementing singletons

class Singleton
{
public:
    static Singleton* theSingleton();
    ...
protected:
    Singleton();
private:
    static Singleton* sm_ptr;
};

Singleton* Singleton::sm_ptr = 0;
Singleton* Singleton::theSingleton()
{
    if ( sm_ptr == 0 )
        sm_ptr = new Singleton;
    return sm_ptr;
}

Singleton details

• Constructor protected to prevent multiple instances.
• Lazy creation avoids two problems with static initialization:
  1) Order undefined
  2) Might need run-time information for initialization.
• Must not call theSingleton until main() starts!
• What if we need deallocation?
Extending Singleton

• We want to be able to substitute new kinds of singleton without changing client code.
• Clients will expect the same interface.
• Maybe select what kind of singleton based on external specification (environment variable).
• Solution: use inheritance and registration.

Extended singleton code(1)

class Singleton
{
public:
    static Singleton* theSingleton();
protected:
    static void register( string name, Singleton* ptr)
        Singleton();
private:
    static Singleton* sm_ptr;
    static map <string, Singleton*> sm_registry;
};
Extended singleton code(2)

Singleton* Singleton::sm_ptr = 0;

// Don’t call this before main()!!!
Singleton* Singleton::theSingleton()
{
    if ( sm_ptr == 0 )
    {
        string name = getenv("SINGLETON");
        sm_ptr = sm_registry[name];
    }
    return sm_ptr;
}

Extended singleton code(3)

map<string, Singleton*> Singleton::sm_registry;

void Singleton::register( string name, Singleton* ptr)
{
    sm_registry[name] = ptr;
}
Extended singleton details

- Subclasses must register themselves with the base class.
- Might do this using constructor for new class together with single static object for each class.
- This does require that all possible singletons get created, and public constructors :-<.
- To avoid this, one needs an explicit switch in theSingleton(), but still no new client code.

Pattern 2: Prototype

- Need a mechanism for creating new objects without knowing their real type.
- We do know they are derived from some common base.
- We will also already have one object of the type we want.
- Solution: something that works like a virtual copy constructor - a clone operation.
Implementing prototypes

class Item
{
public:
    virtual Item* clone() = 0;
    ...
};

class ItemA : public Item
{
public:
    virtual Item* clone();
    ...
};

Prototype details

• Provides mechanism for virtual construction
• May be used with encapsulation by client
• Instantiation of prototypes can be deferred.
• Possible to add new prototypes during runtime (and to remove them).
• Might also want a registry of prototypes.
Pattern 3: Adapter

- We have code which uses an existing abstract base class, Base.
- We want to add a new subclass for use by that code.
- We have a class, Done, that implements the behavior needed, but with the wrong interface.
- Solution: a thin layer of code around the Done class to present the Base interface.

Implementing adapters

class Adapter : public Base
{
public:
    // Base functions
    ...
private:
    Done m_done;
};

or possibly:

class Adapter : public Base, private Done
Adapter details

- Would like to build adapters when interfaces are quite similar.
- May even build two-way adapters using multiple inheritance where both are public.
- Possible to build abstract adapters using encapsulation.

Implementing adapters

// Abstract adapter
class Adapter : public Base
{
public:
    Adapter( Done* dp );
    // Base functions
    ...
private:
    Done* m_done;
};

// Two-way adapter
class Adapter : public Base, public Done
Pattern 4: Memento

- We have an object with considerable hidden state information.
- We need to be able to “checkpoint” the object, modify it, then (possibly) restore it to the checkpoint state.
- Want to do this with minimal impact on the class.

Implementing momentos

class Item
{
public:
    Momento* createMomento();
    void restoreMomento( Momento* );
    ...
};

class Momento
{
public:
    virtual ~Momento();
private:
    friend class Item;
    Momento();
    ...
};
Momento details

• Don’t want Momentos created except by Item::CreateMomento.
• Might want to allow incremental restoration via a stack of Momentos, in which case they may be much smaller.
• Possible to add a file I/O interface to Momentos to get persistence.