Programming to Interfaces

Jim Fawcett

Copyright © 2004

What is an Interface?

- **First answer:**

  public members of a class

```
class fileInfo {
    friend class navig;
public:
    fileInfo();
    fileInfo(const fileInfo &fi);
    fileInfo(const std::string &path);
    ~fileInfo();
    bool firstFile(const std::string &filePattern);
    bool nextFile();
    void closeFile();
  // some members deleted for brevity
    std::string       date() const;
    std::string       time() const;
    std::string       attributes() const;
    bool isArchive()     const;
    bool isCompressed() const;
    bool isDirectory()  const;
  // members deleted
    std::string getPath(void);
    void setPath(const std::string &s);
private:
  // private members deleted
};
```

What is an Interface?

- **Second answer:**

  public members of a class plus global functions packaged with the class.

  Packaged means in the same header file and in the same namespace.

- **Interface Principle:**

  “For a class X, all functions, including [global] functions, that both:

  - Mention X
  - Are supplied with X

  Are logically part of X, because they form part of the interface of X.”

  Herb Sutter, Exceptional C++, Addison-Wesley, 2000

```
class str {
private:
  char *array;
  int len, max;
public:
  str(int n = 10);                    // void and size ctor
  str(const str &s);                  // copy ctor
  str(const char *s);                 // promotion ctor
  ~str();                             // dtor
  str& operator=(const str &s);       // assignment operator
  char& operator[](int n);            // index operator
  char operator[](int n) const;       // index operator for const str
  void operator+=(char ch);           // append char
  void operator+=(const str &s);      // append str s
  int size() const;                   // return number of chars
  void flush();                       // clear string contents
};
std::ostream& operator<<(std::ostream& out, const str &s);
```

Koenig Lookup

- This second definition is consistent with Koenig Lookup:
- Koenig Lookup:

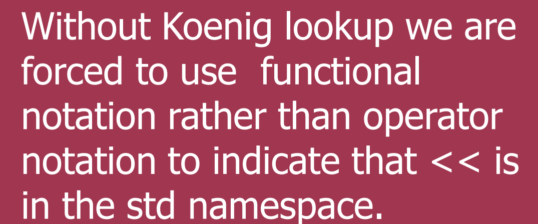
  If you supply a function argument of class type, then to find the function name the compiler is required to look not just in the local or surrounding scopes, but also in the namespace that contains the argument's type. [paraphrased from Sutter, ibid]

- For this reason, for the string class, packaged in namespace std, which has an operator<< we can say:

```
std::cout << myString;
```

instead of:

```
std::operator<<(std::cout, myString);
```



What is an Interface?

- **Third Answer:**

  Abstract class with no data members, no constructor and at least one pure virtual function.

  - An abstract class has the same role as a C# or Java interface.
  - It provides a means to use an implementation class but only binds to the abstraction provided by the interface.

```
class ITest {
public:
  virtual ~ITest() {}
  static ITest* CreateTestImpl();
  virtual std::string ident()=0;
  virtual void addString(const std::string &str)=0;
  virtual std::string getString()=0;
};
```

Note

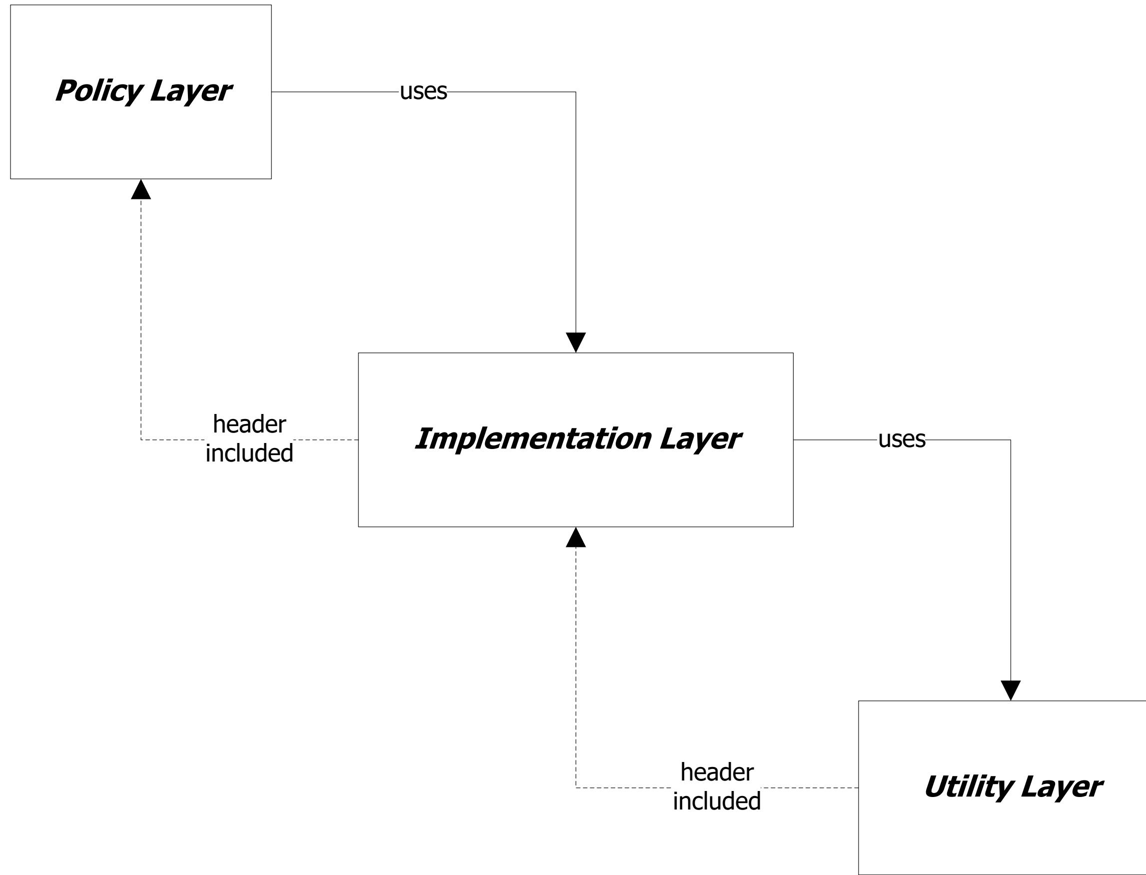
- For the remainder of this presentation, we will be using the third definition:

  - An Interface is an abstract class with no data members and no constructors.

Design Layers

- It is very typical that a software design maps onto the three layers shown on the next slide.
  - Top down design determines a policy layer and top-level partitions of the implementation layer – very little OOD at this level.
  - Classes and class relationships dominate the implementation layer. Polymorphism is an important tool to minimize coupling between components of this layer and with the utility layer.
  - Class encapsulation of data and operating system services, using bottoms up design, determines the utility layer.
  - This decomposition has the advantages that:
    - The policy layer is responsible for satisfying the application's requirements.
    - The implementation layer partitions the program's responsibilities into manageable chunks.
    - The utility layer is a rich source of reusable code. It provides a lot of small simple services used to compose the implementation.

Program Layers – Typical Decomposition



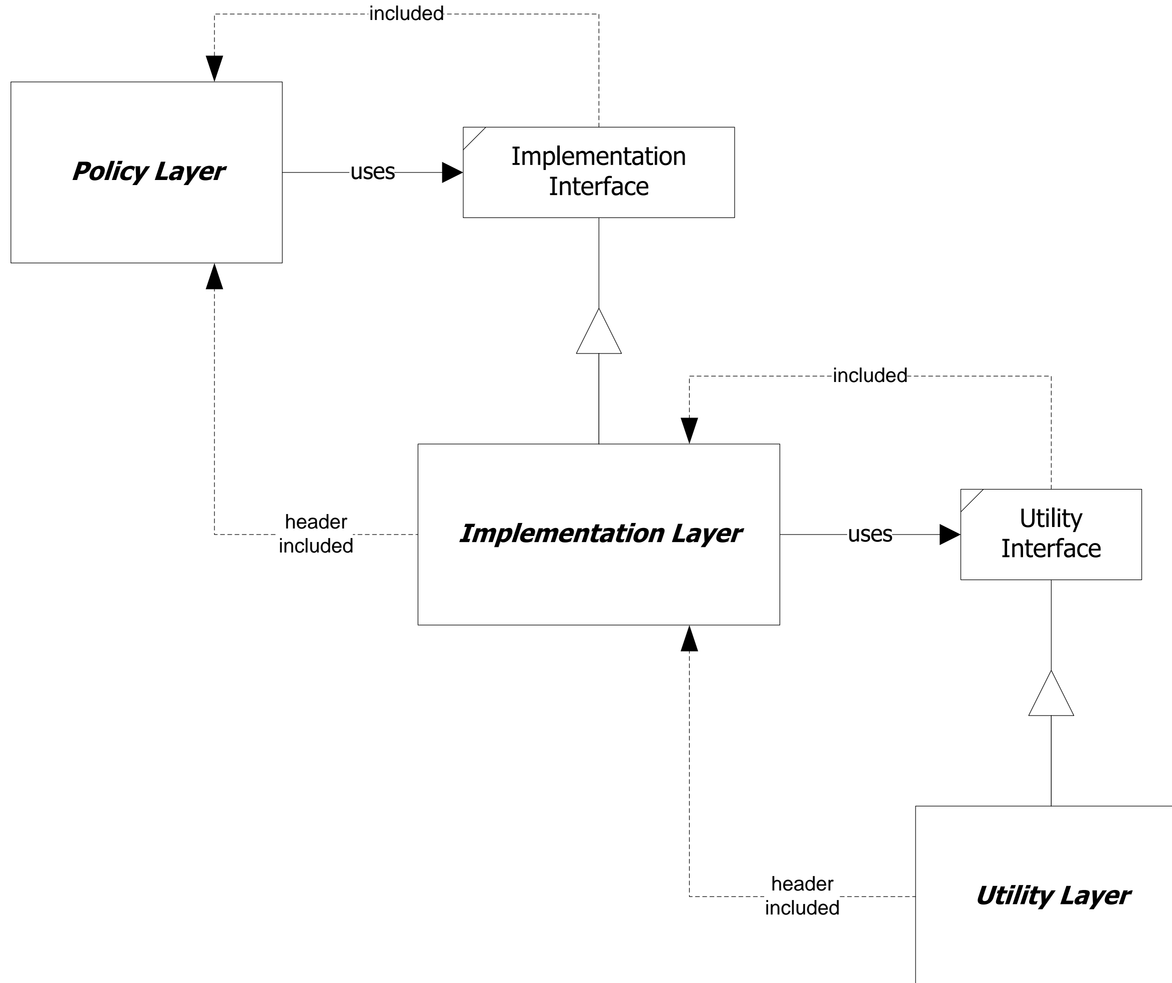
Problems with Layering

- There is one large disadvantage to this “vertical” layering:
  - Each layer is dependent on the layer below.
    - Policies need to create the objects that populate the implementation.
    - But, in order to create these objects, the policy layer must include header files of the implementation modules and so depend on the implementation of those classes.
    - But, the implementation is very volatile during development. Fixing design flaws and latent errors can introduce changes into both the policy and utility layers.
    - The same comments apply to the implementation/utility layering.
  - The need to include headers of unstable code means that the includer is also unstable and changes migrate throughtout the system – a very distressing situation for large systems.

Programming to Interfaces

- If we introduce interfaces which we strive to make stable then changes in lower layers don't affect the design of the upper layer. That layer is simply recompiled without change when the implementation (not the interface) is changed.
    - An interface is an abstract base class that has no implementation. Thus it has no latent errors or performance problems to fix. As long as the layer above only uses that interface, changes in the implementation of interface functions don't break the client's code. Note that those implementations are provided by classes that derive from the interface.
    - Upper layers still have to include the lower layer's header files in order to create the objects in that layer. This means that the upper layer must be recompiled whenever the lower layer's implementation changes, because the size of the objects are very likely to change.

Binding to Interfaces, not Implementation



Isolating Layers

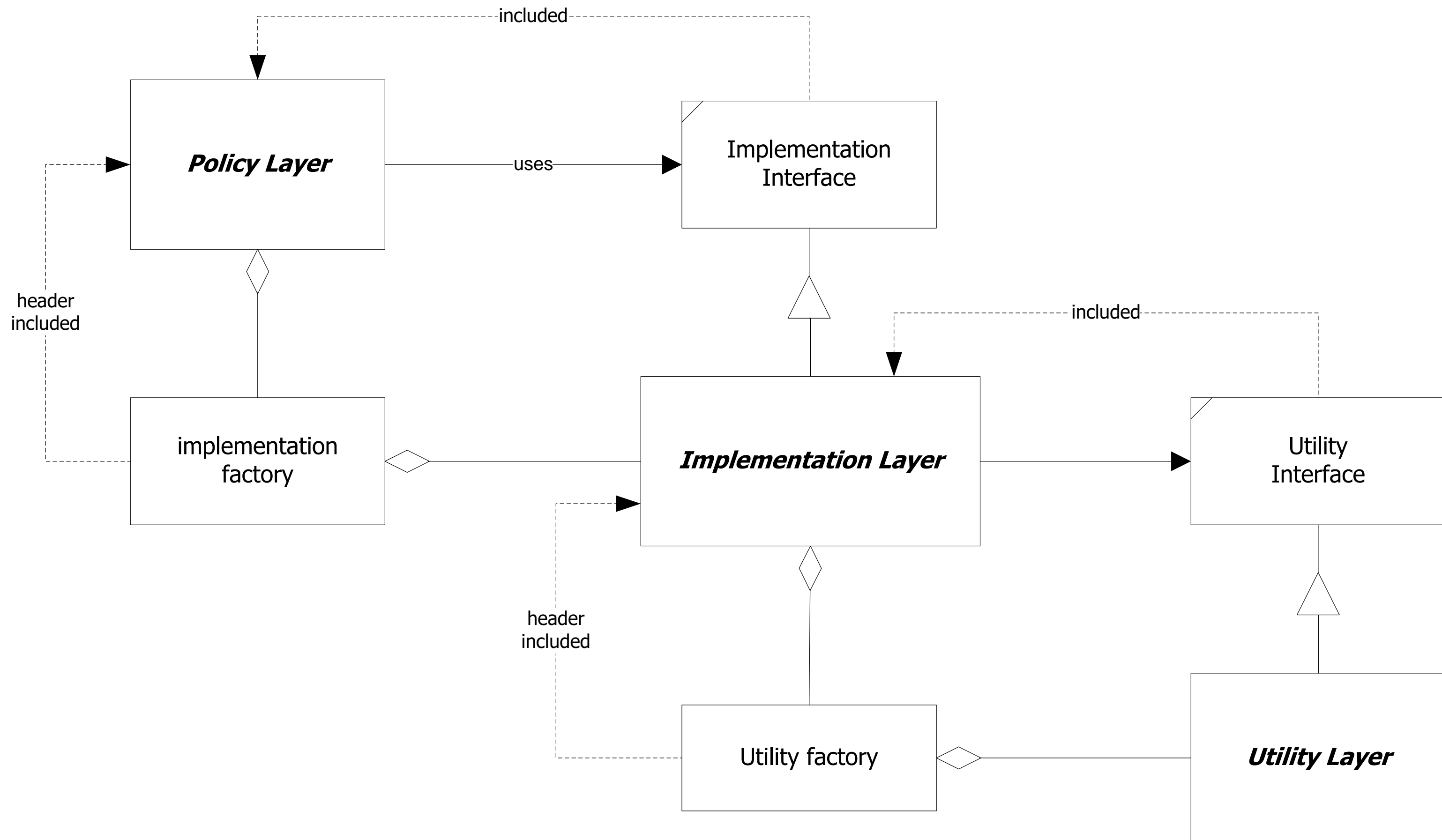
- But, with a little extra work, we can do better. By providing object factories to build all objects in an implementation, clients that program only to interfaces no longer need to include header files of the imple-mentation, they just include header files of the factories, as shown on the next page.

  The result of this architecture is that:

  - The policy layer depends only on the implementation interface and object factories, neither of which are likely to change.
  - The implementation layer depends only on its own interface and implementation and on the interface and object factories of the utility layer.
  - The utility layer depends only on its own interface and implementation.

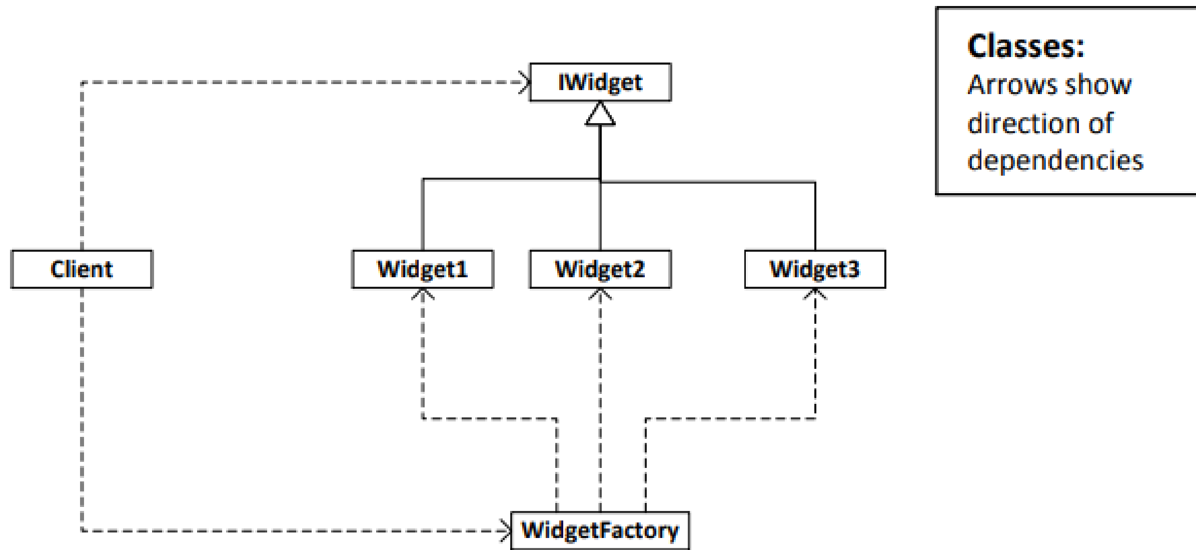
- When a change is made to the implementation, we now find that we need not recompile the policy layer.

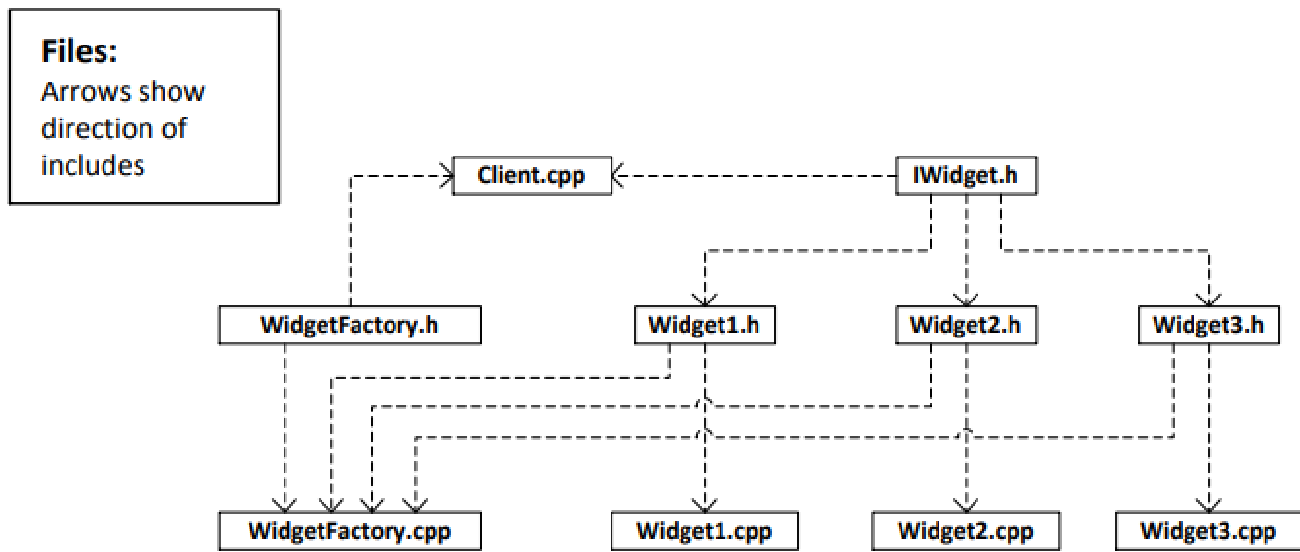
Using Object Factories to Isolate Layers



Example Application

- We will illustrate these ideas with a little prototype code.
  - The example has three implementation layer classes Widget1, 2, and 3. The implementation layer provides an IWidget interface that establishes a protocol for clients to use when interacting with the implementation.
  - The policy layer, Client, simply instantiates the Widget objects, using an object factory provided by the Widget project.
  - It then proceeds to use them by calling a function of the public interface, IWidget, on each one.

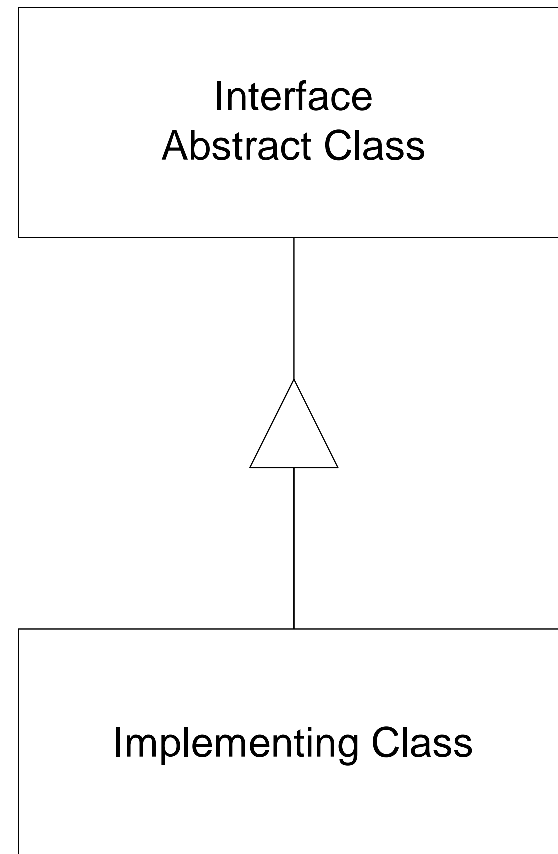

Classes:
Arrows show direction of dependencies
IWidget
Client
Widget1
Widget2
Widget3
WidgetFactory


Files:
Arrows show direction of includes
Client.cpp
IWidget.h
WidgetFactory.h
Widget1.h
Widget2.h
Widget3.h
WidgetFactory.cpp
Widget1.cpp
Widget2.cpp
Widget3.cpp

Effect of Changes

- Changes to Interface provided by component

- Changes to Implementation of component



Changes to Interface

- Changing the signature of a method:
  - Breaks the design of every client using the method.
- Changing the order of methods:
  - Forces recompilation of every client
- Adding a new method:
  - May force recompilation of every client

Changes to Implementation

- If client has reference to concrete class:
  - Forces recompilation, may break client's design.
- If client has reference to interface but creates implementing component:
  - Forces recompilation.
- If client has reference to interface and uses a factory to create implementing component:
  - Factory must be recompiled, clients simply relink.
  - If the factory holds pointers to static creational functions even factory does not have to be rebuilt.
- If component is built as a DLL, provides an interface and a pluggable factory:
  - New library is copied over the existing library. Nothing needs to be done to client.

End of Presentation