

Satisfying Open/Closed Principle

Jim Fawcett

CSE687 – Object Oriented Design

Spring 2003

Statement of Principle

- Software entities (classes, modules, functions) should be open for extension, but closed for modification.
- Definitions
 - Open:
A component is open if it is available for extension:
 - add data members and operations through inheritance.
 - Create a new policy template argument for a class that accepts policies.
 - Closed:
A component is closed if it is available for use by other components but may not, itself, be changed, e.g., by putting it under configuration management, allowing read only access.

Application Domain vs. Solution Domain

- As designers, one of our goals should be to build programs with an Executive layer and a series of libraries of reusable code.
 - The executive layer consists of an executive module and, perhaps, a few top level modules that are all application specific.
 - This top layer supports all of the application requirements.
 - The remainder of the design consists of reusable components.
 - The reusable part are solution-side components that carry out basic and often needed operations.
- This ideal is hard to realize unless we can make the reusable part extensible.
 - Seldom do we have enough foresight to predict all the needs of the application.
 - Application requirements change while we're building.

Extensions and Binding

- If our efforts to build reusable components are successful, each new project will need to build less on the reusable side and can focus on meeting project requirements.
- That can only work if the reusable components can be extended. We almost always achieve extendability by providing for application design-time binding in our library design-time designs.
 - Design for application-binding when doing library design.

How do Libraries provide Application Binding?

- Polymorphism

- Build hook classes that provide virtual functions that applications override to define application specific processing.
- Define protocols that the library uses and application designers provide by overriding in a derivation of the protocol class.
 - The library may provide derived classes to meet known requirements.
 - Application designers add new derived classes driven by increasing knowledge about the requirements or requirements changes.

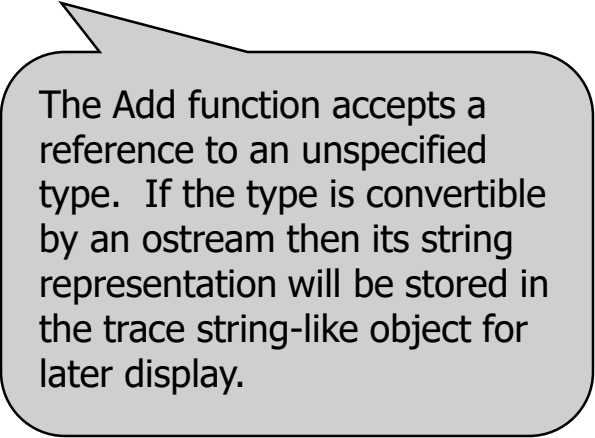
- Template policies and traits

- Build functions and classes parameterized by policy classes that allow application designers to extend by defining new policies, without changing any of the template class.
- Provide traits to make it easier for application designers to build policy classes.

Extending Functions

- Closed functions with template parameters can be extended by defining new compatible classes, used as template parameters.
- Example: trace class member

```
template <typename T> trace& Add(const T& t)
{
    std::ostringstream oss;
    oss << t;
    *this += ' ';
    *this += oss.str();
    return *this;
}
```



The Add function accepts a reference to an unspecified type. If the type is convertible by an ostream then its string representation will be stored in the trace string-like object for later display.

Extending Functions

- If we develop a user defined type that can serialize its state to a stream:
 - A `vec3D` converting its coordinates to a displayable string
 - An `html element` object serializing to a tagged string

Then this type can be used by `trace::Add(const T &t)` to save its string representation for a trace display.

- We could make `trace` effective for debugging a current project by providing serialization to string form for objects used in the implementation.

Extending Classes

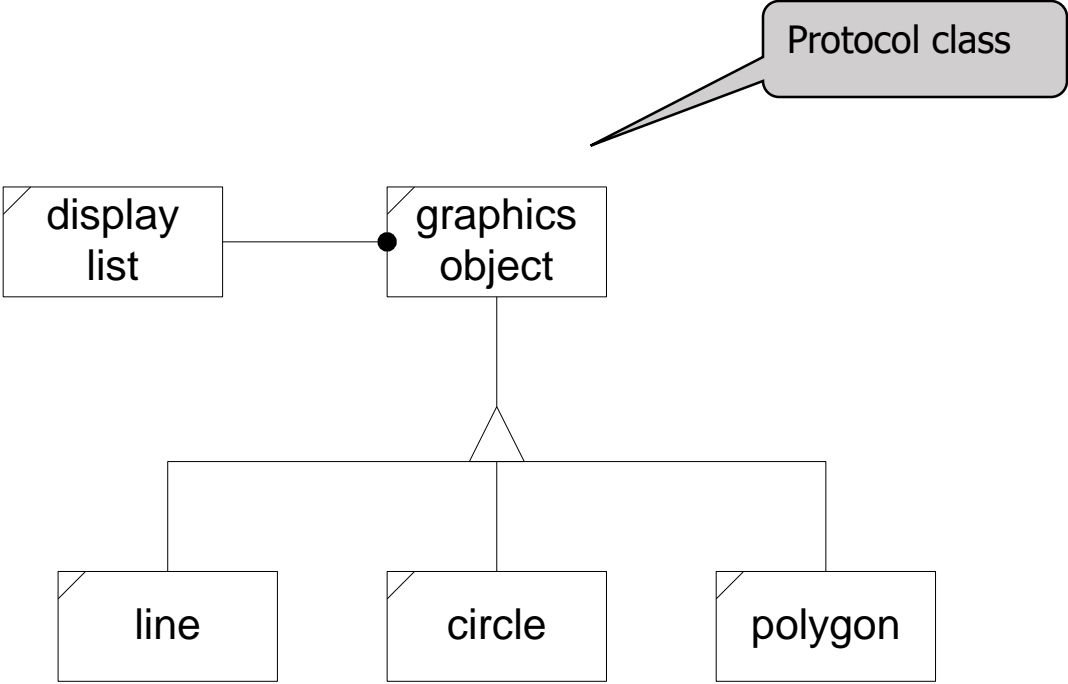
- If the library designer supported extensions by providing:

- Protocol classes
- Hook classes
- Template policy parameters and traits

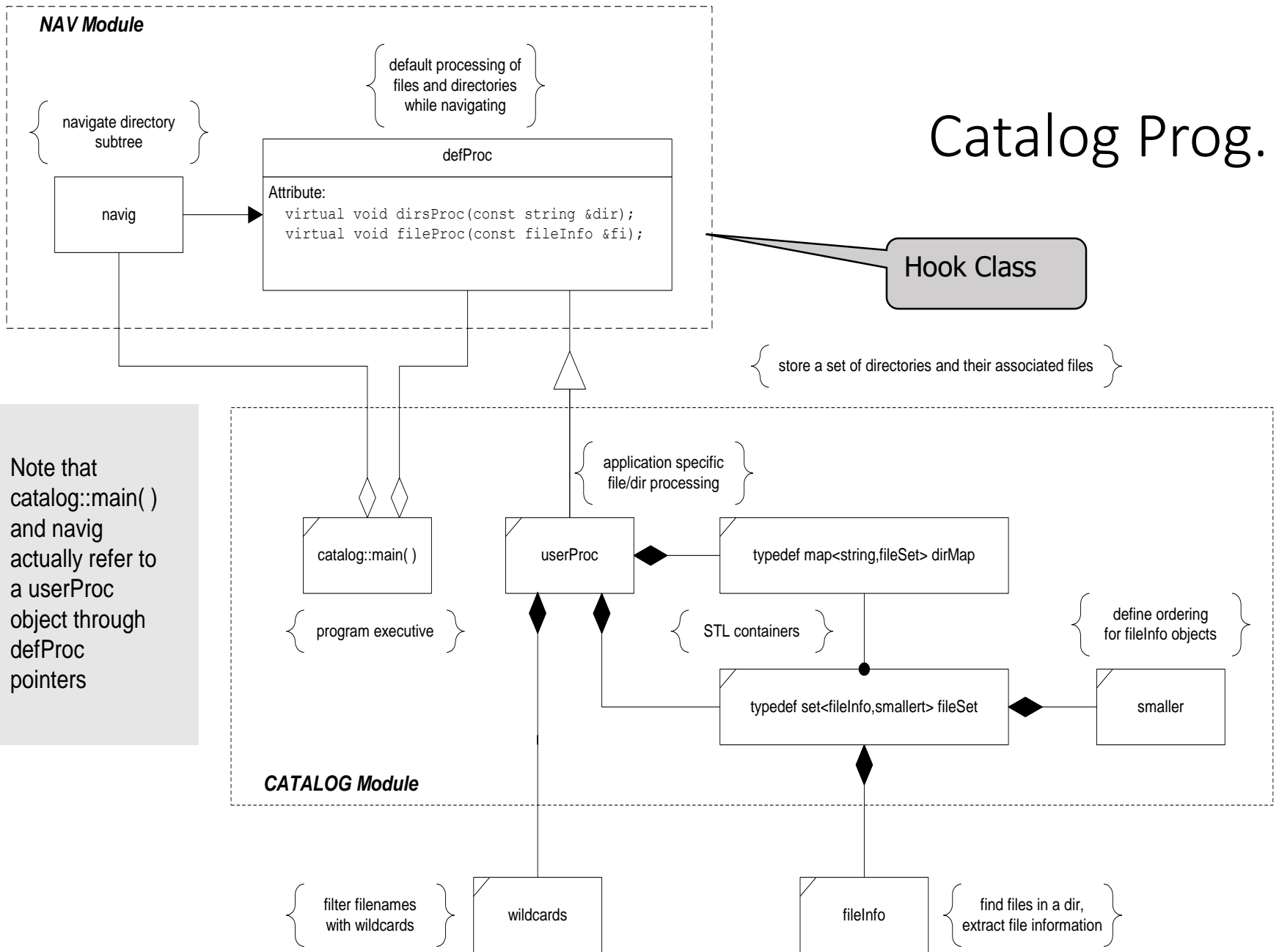
then you simply design plug-in classes to tailor operation of the library.

- Even when the library does not provide these facilities, you often can extend existing concrete classes as we did for trace.

Graphics Editor



Catalog Prog.



```
TOK.H - WordPad
File Edit View Insert Format Help
[Icons]

////////////////////////////////////
// functor that defines delimiters needed for code analysis

class codeDelimiter {

public:
    bool operator()(char ch);
};

//----< is this char a delimiter? >-----

inline bool codeDelimiter::operator()(char ch) {

    switch(ch) {
        case '[' : return 1;
        case ']' : return 1;
        case '{' : return 1;
        case '}' : return 1;
        case '(' : return 1;
        case ')' : return 1;
        case ';' : return 1;
        case ',' : return 1;
        default : return 0;
    }
}

////////////////////////////////////
// tokenizer class

template <class T> class toker {

public:
    toker(std::istream &input);
    bool getTok(std::string &buffer);
    unsigned int lines();
    unsigned int scopeLevel();
    void collectSingleQuotes(bool pred=true);
    void nestedComments(bool pred=true);
private:
    T delim;
    std::istream &in; // source of tokens
    unsigned int _lines; // number of newlines
    unsigned int _scopeLevel; // net number of open/closed braces
    enum charType { Char, NewLine, White, Punct };
    charType type(char ch);
    int isComment(const std::string &buffer);
    int isWhite(const std::string &buffer);
    int eatWhite(std::string &buffer);
    int eatComment(std::string &buffer);
    int isQuote(const std::string &buffer);
    int collectQuote(std::string &buffer);
    bool _collectSingleQuotes;
    bool _nestedComments;
};

For Help, press F1
```

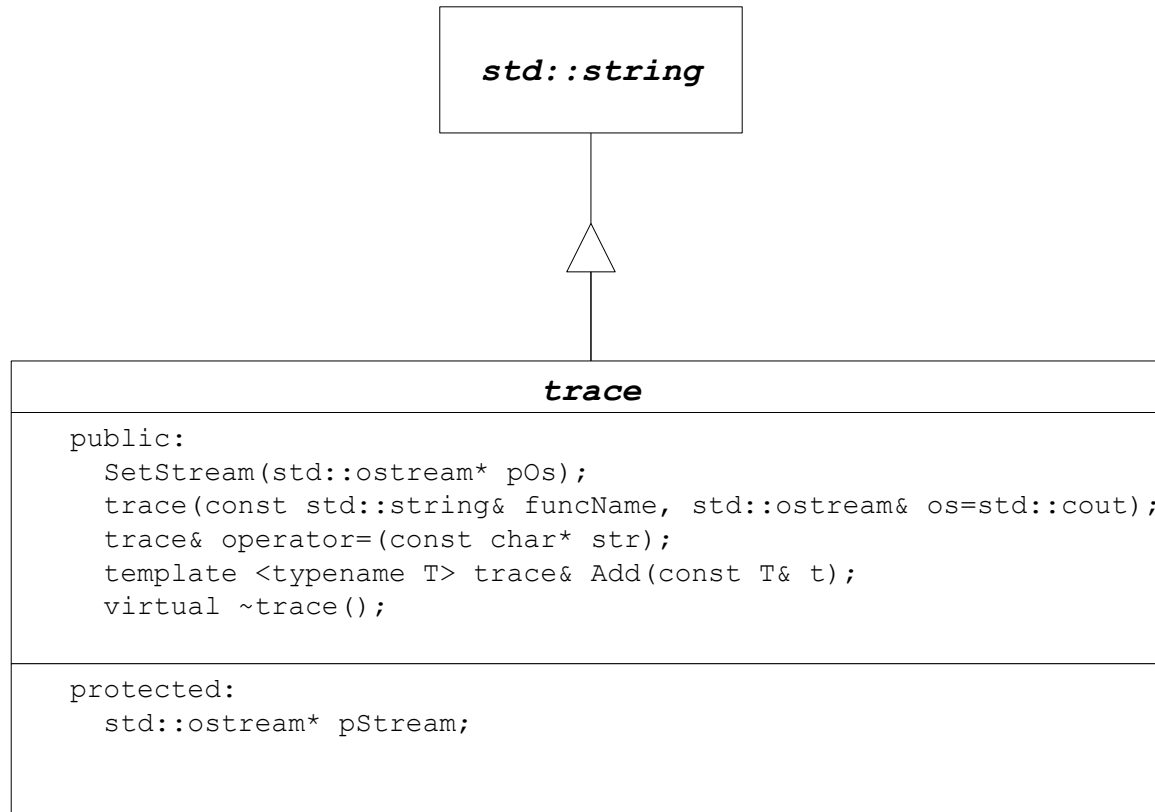
codeDelimiter
Template Policy
Class

The codeDelimiter class is a functor that defines all single-character tokens.

An object of this class is used by toker to customize how tokens are collected:

- for parsing code
- for parsing XML documents

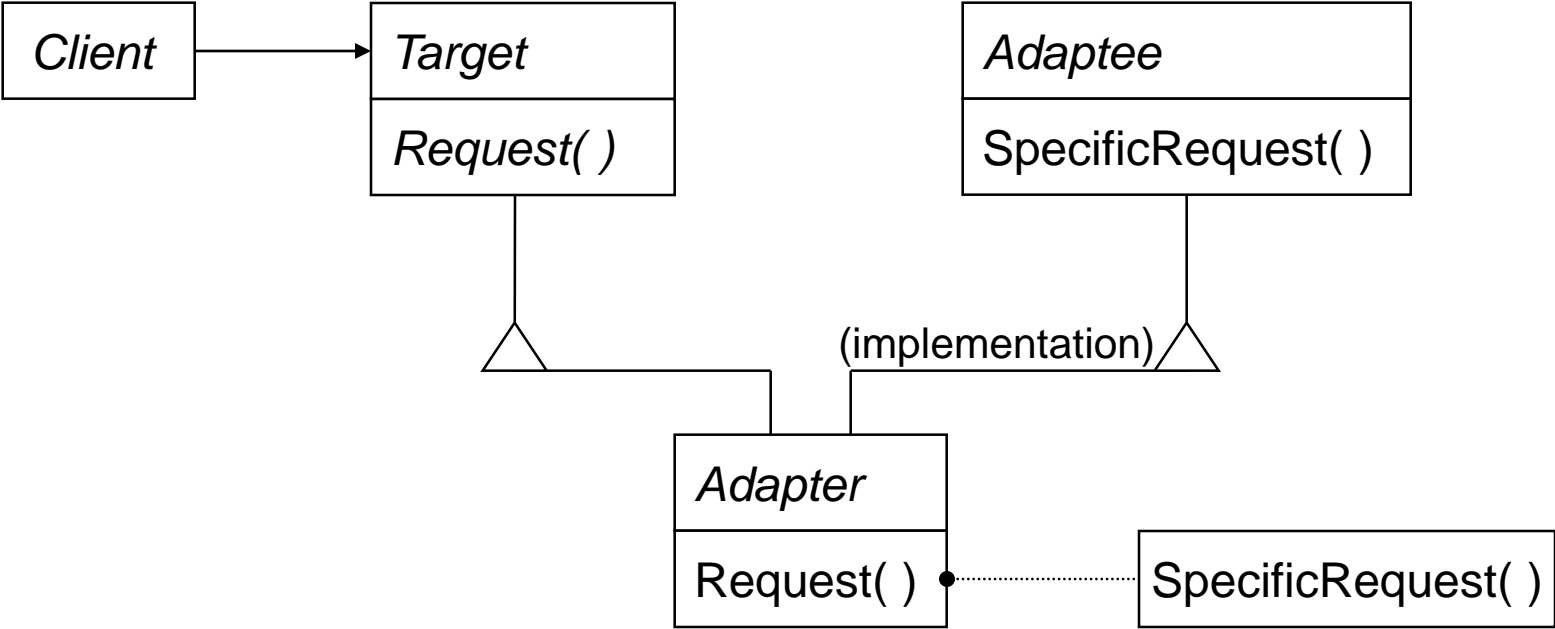
Deriving from Concrete Classes



Even this is not the End of the Story!

- The design patterns class covers about 30 patterns, some of which are designed to support extension of existing classes.
- The Adapter Pattern is typical of these. It provides a way to provide a specific required interface for a client from a class or classes that do not directly support the interface, but do provide a lot of the required functionality.
- Most of the patterns focus on ways a client can avoid binding to a concrete class.

Class Adapter



End of Presentation