A Novel Use of `friend`

Michael D. Borghardt

"The Attorney-Client Idiom"

friend

“Friendship breaks encapsulation”

“Indicators of poor design”

“Avoid friendship like the plague”

“Best to use friends with a guilty conscience”

Introduction

- Keeping encapsulation
- Refactoring existing code

Agenda

- Encapsulation
- What is a friend
- Controlling friendship

How do you get access to foo?

```
class Client
{
private:
  void foo() {};
  void bar() {};
};
```

```
int main()
{
  Client c;

  c.foo(); // error
}
```

3 ways to access foo

1) Make Client::foo() public
2) Refactor your design
3) Use `friend`

Making it `public`

- Compromises encapsulation
- Negative consequences for testing
- Negative consequences for documentation

Refactoring

- Move Method
- Move Field
- Extract Class Remove Middle Man

Martin Fowler Refactoring: Improving the Design of Existing Code, 2000

Using `friend`

- Designing two or more classes to work together collaboratively
- During maintenance one class “needs” access to some private member data of another
- One class “needs” access to some private member function(s) of another

Guideline using friend functions

```
if (f virtual)
   f should be a member function of T
else if (f is << or >> operator)
   f should be a non-member function
   if (f needs access to non-public members of T)
        f should be a friend of T
else if (f needs type conversion on its left-most argument)
   f should be a non-member function
   if (f needs access to non-public members of T)
        f should be a friend of T
else if (f can be implemented via T's public interface)
   f should be a non-member function
else
   f should be a member function of T
```

Scott Meyers CUJ February 2000

The Three Ps – Cline et al

- Position – operation on left-hand side
- Promotion – type conversion on left-hand side
- Perception – esthetic judgment

Marshal Cline et al C++ FAQs 2nd 1999

Vocabulary

- Grantor – a class granting friendship, either to a free function, a class member, or a whole class
- Friend Method – a class member function that has been declared as a friend by some grantor
- Friendly Class – a friend class, or a class containing one or more friend methods

Guideline for friend classes and class members

- Only private, non-virtual members of a friendly class should be declared as friend methods

Guideline for friend classes and class members continued

- A friend should only be granted access to private member functions (virtual or non-virtual) of the grantor

Template Method

```
class Base
{
public:
  DoSomething() { preAmble(); StandardThing(); postAmble()};
private
  void StandardThing();
  virtual void preAmble() = 0;
  virtual void postAmble() = 0;
};

class Derived: public Base
{
private:
  virtual void preAmble();
  virtual void postAmble();
};
```

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Erich, et al, 1995

Virtual Friend Function idiom

```
class Base {
 public:
   friend std::ostream& operator<< (std::ostream& o, const Base& b);
   ...
 protected:
   virtual void printOn(std::ostream& o) const;
 };

 inline std::ostream& operator<< (std::ostream& o, const Base& b)
 {
   b.printOn(o);
   return o;
 }

 class Derived : public Base {
 protected:
   virtual void printOn(std::ostream& o) const;
 };
```

Marshal Cline et al C++ FAQs 2nd 1999

Consigliere

A close, trusted friend and confidant.

Wikipedia.org

Consigliere

```
class Client
{
private:
    void foo() {};
    void bar() {};
};
```

```
int main()
{
  Client c;

  c.foo(); // error
}
```

Consigliere

```
class Client
{



private:
  void foo() {};
  void bar() {};

};
```

```
int main()
{
  Client c;


}
```

Consigliere

```
class Client
{
public:
  class Consigliere
  {
  public:
    static void foo(Client& c)
    {c.foo();}
  };
private:
  void foo() {};
  void bar() {};

  friend class Consigliere;
};
```

```
int main()
{
  Client c;


}
```

Consigliere

```
class Client
{
public:
  class Consigliere
  {
  public:
    static void foo(Client& c)
    {c.foo();}
  };
private:
  void foo() {};
  void bar() {};

  friend class Consigliere;
};
```

```
int main()
{
  Client c;

  Client::Consigliere::foo(c);
}
```

Attorney

One who acts on behalf of another person in some capacity.

Wikiepedia.org

Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};
};
```

```
int main()
{
  Client c;

  c.foo(); // error
}
```

Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};


};
```

```
int main()
{
  Client c;


}
```

Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};

  friend class Attorney;
};

class Attorney
{
public:
  static void foo(Client& c)
  {
     c.foo();
  }
};
```

```
int main()
{
  Client c;

  Attorney::foo(c);
}
```

Counsel

The advocate or advocates engaged in the direction of a cause.

Dictionary.com

Counsel

```
class ClientA
{
private:
  void foo() {};
  void bar() {};
};
```

```
class ClientB
{
private:
  void foo() {};
  void bar() {};
};
```

```
int main()
{
  ClientA a;
  ClientB b;

  a.foo(); // error
  b.foo(); // error
}
```

Counsel

```
class ClientA
{
private:
  void foo() {};
  void bar() {};


};

class ClientB
{
private:
  void foo() {};
  void bar() {};


};
```

```
int main()
{
  ClientA a;
  ClientB b;

  a.foo(); // error
  b.foo(); // error
}
```

Counsel

```
class ClientA
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};

class ClientB
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};
```

```
int main()
{
  ClientA a;
  ClientB b;


}
```

Counsel

```
class ClientA
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};

class ClientB
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};
```

```
class Counsel
{
public:
  static void foo(ClientA& a)
  {a.foo();}

  static void foo(ClientB& b)
  {b.foo();}
};

int main()
{
  ClientA a;
  ClientB b;



}
```

Counsel

```
class ClientA
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};

class ClientB
{
private:
  void foo() {};
  void bar() {};

  friend class Counsel;
};
```

```
class Counsel
{
public:
  static void foo(ClientA& a)
  {a.foo();}

  static void foo(ClientB& b)
  {b.foo();}
};

int main()
{
  ClientA a;
  ClientB b;

  Attorney::foo(a);
  Attorney::foo(b);
}
```

Enforcing the Contract

- Friendship in C++ is an all-or-nothing proposition
- Attorney-client idiom

Good Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};

  friend class GoodAttorney;
};
```

```
class GoodAttorney
{
private:
  static void foo(Client& a){a.foo();}
};
```

```
int main()
{
  Client c;

  GoodAttorney::foo(c);
}
```

Good Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};

  friend class GoodAttorney;
};
```

```
class GoodAttorney
{
private:
  static void foo(Client& a){a.foo();}

};
```

```
int main()
{
  Client c;


}
```

Good Attorney

```
class Client
{
private:
  void foo() {};
  void bar() {};

  friend class GoodAttorney;
};
```

```
class GoodAttorney
{
private:
  static void foo(Client& a){a.foo();}
  friend void foo(Client&);
};

void foo(Client& c)
  {GoodAttorney:foo(c);}

int main()
{
  Client c;

  foo(c);
}
```

Performance

- What is the cost of using this idiom?
- Inline functions that pass through to their client objects
- Main cost is increased complexity in your code

Conslusion

- Using friendship may enhance encapsulation
- Using friendship properly requires understanding of when appropriate but also how to structure
- Enforcing friendship requires the attorney-client idiom

Summary

- Refactoring code
- Keeping encapsulation
- Attorney-client idiom
- Template Pattern
- Virtual Friend Idiom

Where to Get More Information

- January 2006 CUJ, Alan R. Bolton
- C++ FAQs Cline et al
- February 200 CUJ, Scott Meyers
- Refactoring, Martin Fowler
- Design Patterns, Gamma et al
- Efficient C++, Bulka et al
- Effective C++, Scott Meyers
- Design and Evolution of C++, Bjarne Stroustrup
- C++ Programming Language, Bjarne Stroustrup