

# Singleton (Одиночка)

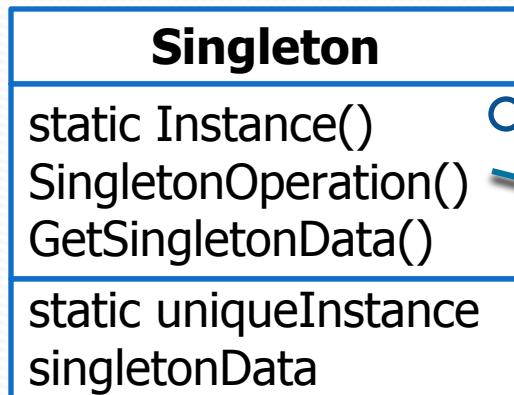
# Назначение паттерна «Одиночка»

- Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа
  - Приложению может потребоваться одна-единственная фабрика компонентов пользовательского интерфейса
  - Приложению может потребоваться одна-единственная база данных

# Применимость

- Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам
  - Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода

# Структура



return uniqueInstance

- Определяет операцию Instance() (статический метод в C++), которая позволяет клиентам получать доступ к единственному экземпляру
- Может нести ответственность за создание собственного уникального экземпляра

# Отношения

- Клиенты получают доступ к экземпляру класса Singleton только через его операцию Instance

# Достоинства

- Контролируемый доступ к единственному экземпляру
- Уменьшение числа имен по сравнению с глобальными переменными
- Допускает уточнение операций и представления
  - От класса Singleton можно порождать подклассы
- Допускает переменное число экземпляров
  - Необходимо лишь изменить операцию Instance
- Большая гибкость, чем у статических функций класса
  - В C++ статические функции не могут быть виртуальными => нельзя использовать полиморфизм

# Простейшая реализация

```
template<class T> class CSingleton : public boost::non_copyable
{
public:
    static T& Instance()
    {
        // у класса T есть конструктор по умолчанию
        static T theSingleInstance;
        return theSingleInstance;
    }
};

class COnlyOne : public CSingleton<COnlyOne>
{
    friend class CSingleton<COnlyOne>
    COnlyOne() {} // закрытый конструктор по умолчанию
    //.. интерфейс класса
};

int main()
{
    COnlyOne & one = COnlyOne::Instance();
    // использование экземпляра one
}
```

# Thread-safe реализация паттерна «Одиночка»

```
#include <boost/utility.hpp>
#include <boost/thread/once.hpp>
#include <boost/scoped_ptr.hpp>

// Warning: If T's constructor throws, instance() will return a null reference.
template<class T> class CSingleton : private boost::noncopyable
{
public:
    static T& instance()
    {
        boost::call_once(init, flag);
        return *t;
    }
    static void init() // never throws
    {
        t.reset(new T());
    }
protected:
    ~CSingleton() {}
    CSingleton() {}
private:
    static boost::scoped_ptr<T> t;
    static boost::once_flag flag;
};

template<class T> boost::scoped_ptr<T> CSingleton<T>::t(0);
template<class T> boost::once_flag CSingleton<T>::flag = BOOST_ONCE_INIT;
```



# Пример использования

```
#include <boost/utility.hpp>
#include <boost/thread/once.hpp>
#include <boost/scoped_ptr.hpp>
// Warning: If T's constructor throws, instance() will return a null
reference.
template<class T> class Singleton : private boost::noncopyable
{
public:
    static T& Instance()
    {
        boost::call_once(Init, flag);
        return *t;
    }
    static void Init() // never throws
    {
        t.reset(new T());
    }
protected:
    ~Singleton() {}
    Singleton() {}
private:
    static boost::scoped_ptr<T> t;
    static boost::once_flag flag;
};

template<class T> boost::scoped_ptr<T> Singleton<T>::t(0);
template<class T> boost::once_flag Singleton<T>::flag = BOOST_ONCE_INIT;
```

```
class CMyClass : public CSingleton<CMyClass>
{
    friend class CSingleton<CMyClass>;
public:
    void DoSomething()
    {
        std::cout << "Something";
    }
private:
    CMyClass();
};

void test()
{
    CMyClass::Instance().DoSomething();
}
```

# Особенности реализации паттерна «Одиночка» в C++

- В C++ не определяется порядок вызова конструкторов для глобальных объектов через границы единиц трансляции
  - Между одиночками не может существовать никаких зависимостей

# Структурные паттерны

# Структурные паттерны

- Определяют различные сложные структуры, изменяющие интерфейс существующих объектов или его реализацию
  - Адаптер (Adapter)
  - Мост (Bridge)
  - Компоновщик (Composite)
  - Декоратор (Decorator)
  - Фасад (Facade)
  - Приспособленец (Flyweight)
  - Заместитель (Proxy)

# Адаптер (Adapter)

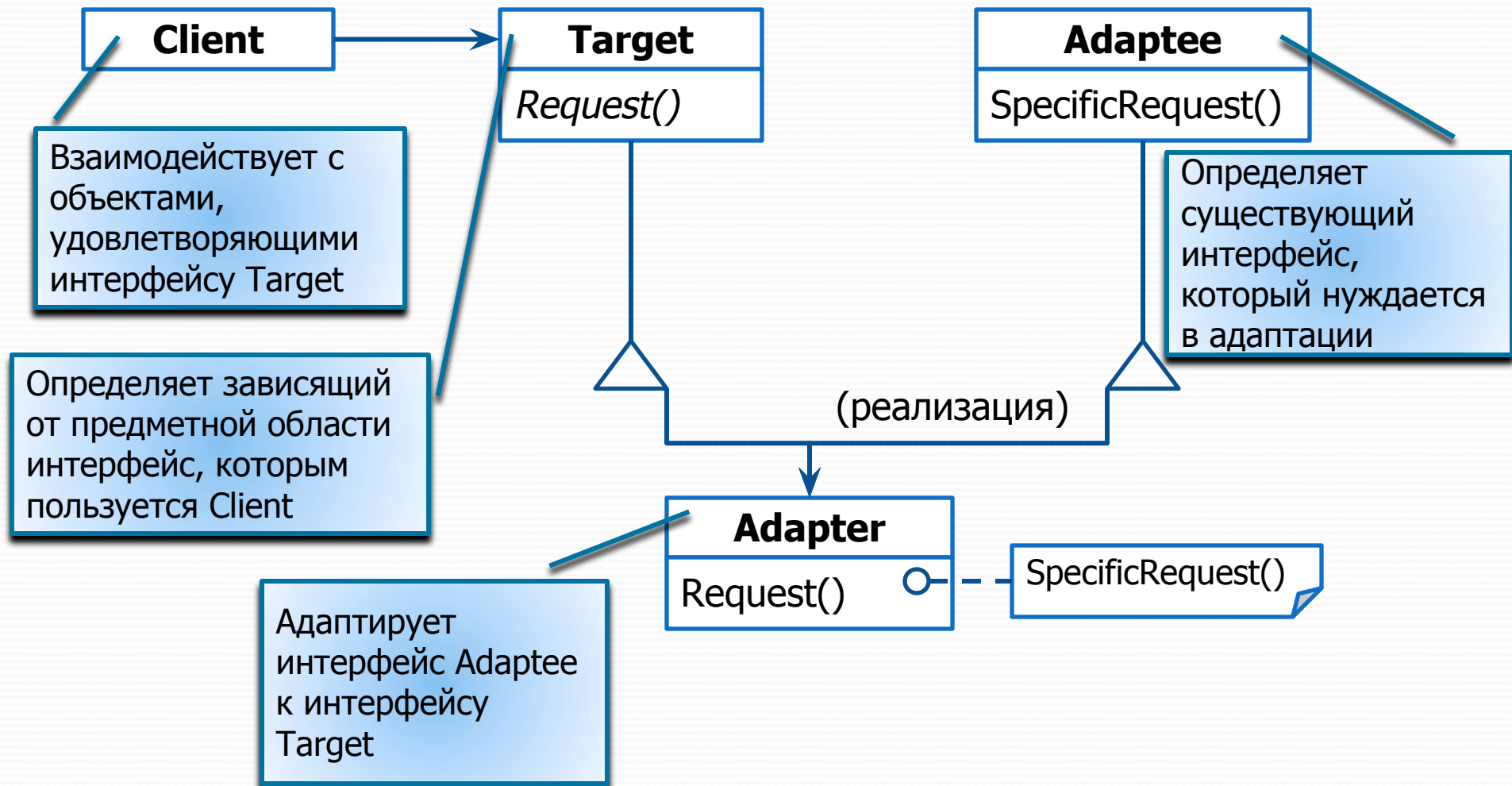
# Паттерн Adapter

- Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты
- Обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна
- Альтернативное название – Wrapper (Обертка)
- Типы:
  - Адаптер класса
  - Адаптер объекта

# Применимость

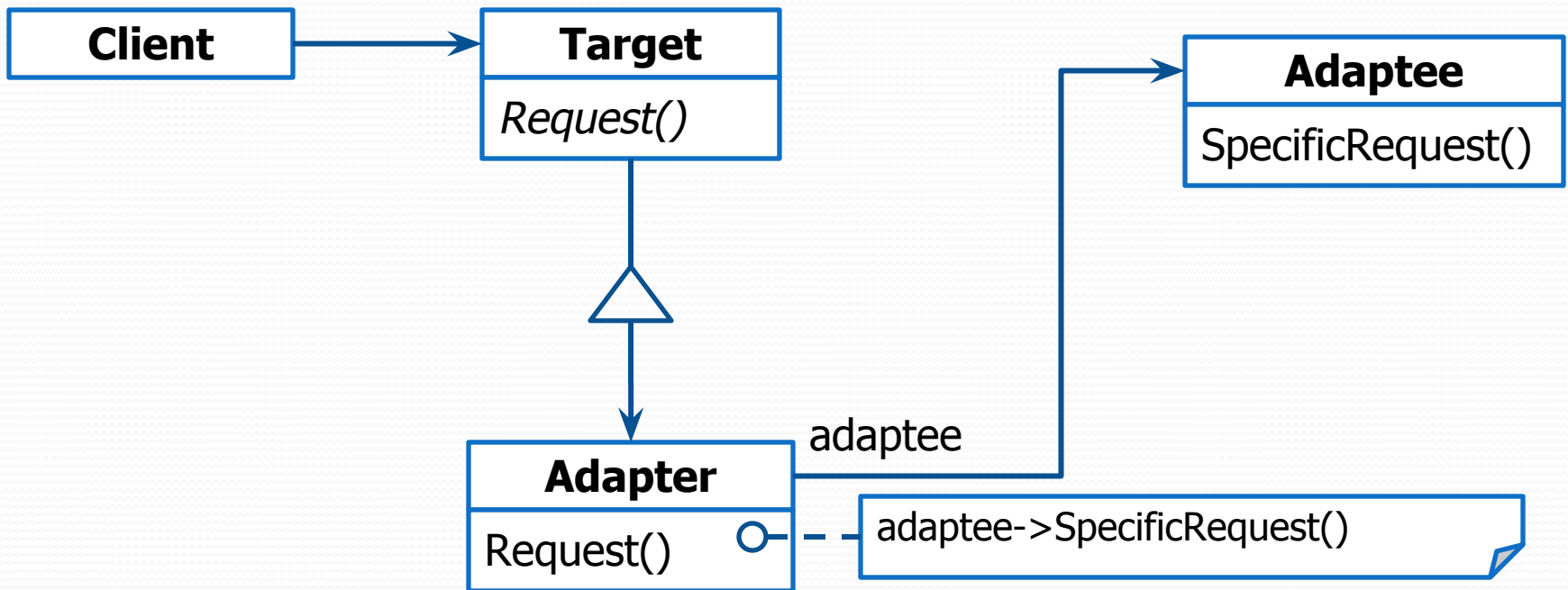
- Необходимо использовать существующий класс, но его интерфейс не соответствует заданным требованиям
- Создание повторно используемого класса, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы
- Использование нескольких существующих подклассов, приспособивая интерфейс их общего родительского класса (только для адаптера объектов)

# Структура адаптера класса





# Структура адаптера объектов



# Отношения (адаптер класса)

- Адаптер класса адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee
  - Этот паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы
- Позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee
- Вводит только один новый объект
  - Для того, чтобы добраться до адаптируемого класса, не нужно дополнительного обращения по указателю

# Отношения (адаптер объектов)

- Позволяет одному адаптеру Adapter работать со многими адаптируемыми объектами Adaptee
  - С самим Adaptee и его подклассами при их наличии
  - Адаптер может добавить новую функциональность сразу всем адаптируемым объектам
- Затрудняет замещение операций класса Adaptee
  - Для этого необходимо породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласса, а не на сам Adaptee

# Вопросы, которые необходимо иметь в виду

- Объем работы
  - Зависит от того, насколько сильно различаются интерфейсы целевого и адаптируемого классов
- Сменные адаптеры
  - Решается путем адаптации интерфейсов
- Использование двусторонних адаптеров для обеспечения прозрачности
  - Полезны в тех случаях, когда необходимо клиенту необходимо видеть как адаптируемый, так и целевой объект

# Реализация адаптеров классов

- В C++ Adapter должен открыто наследоваться от Target и закрыто – от Adaptee
  - Adapter – подтип Target, но не Adaptee

```
class CTarget
{
public:
    virtual void DoSomething() = 0;
};
class CAdaptee
{
public:
    void DoSomethingGood();
};
class CAdapter : public CTarget, private CAdaptee
{
public:
    virtual void DoSometing()
    {
        DoSomethingGood();
    }
};
```

# Пример – иерархия графических объектов

```
class CPoint
{
public:
    int x, int y
};
class CShape
{
public:
    virtual void GetBoundingBox(CPoint & bottomLeft, CPoint & topRight) const
};

class CTextView
{
public:
    int GetLeft() const;
    int GetTop() const;
    int GetWidth() const;
    int GetHeight() const;
};
```

Задача – добавить в иерархию класс CTextShape (наследник CShape), используя функциональность класса CTextView

# Решение

```
class CPoint
{
public:
    ...
    int x, int y
};
class CShape // целевой объект
{
public:
    virtual void GetBoundingBox(CPoint & bottomLeft, CPoint & topRight) const;
};

class CTextView {...}; // адаптируемый класс

// адаптер
class CTextShape : public CShape, private CTextView
{
public:
    virtual void GetBoundingBox(CPoint & bottomLeft, CPoint & topRight) const
    {
        bottomLeft.x = GetLeft();
        bottomLeft.y = GetTop() + GetHeight();
        topRight.x = GetLeft() + GetWidth();
        topRight.y = GetTop();
    }
};
```

# Реализация сменных адаптеров

- Задача – разработать компонент CTreeDisplay для визуализации древовидных структур
  - Иерархии классов
  - Дерево папок
  - Иерархии живых организмов
- Для разных типов структур нужны разные операции доступа к потомкам:
  - GetSubclasses для классов, GetSubdirectories для файловой системы, и т.п.
- Компонент CTreeDisplay должен уметь отображать иерархии обоих видов даже если у них разные интерфейсы



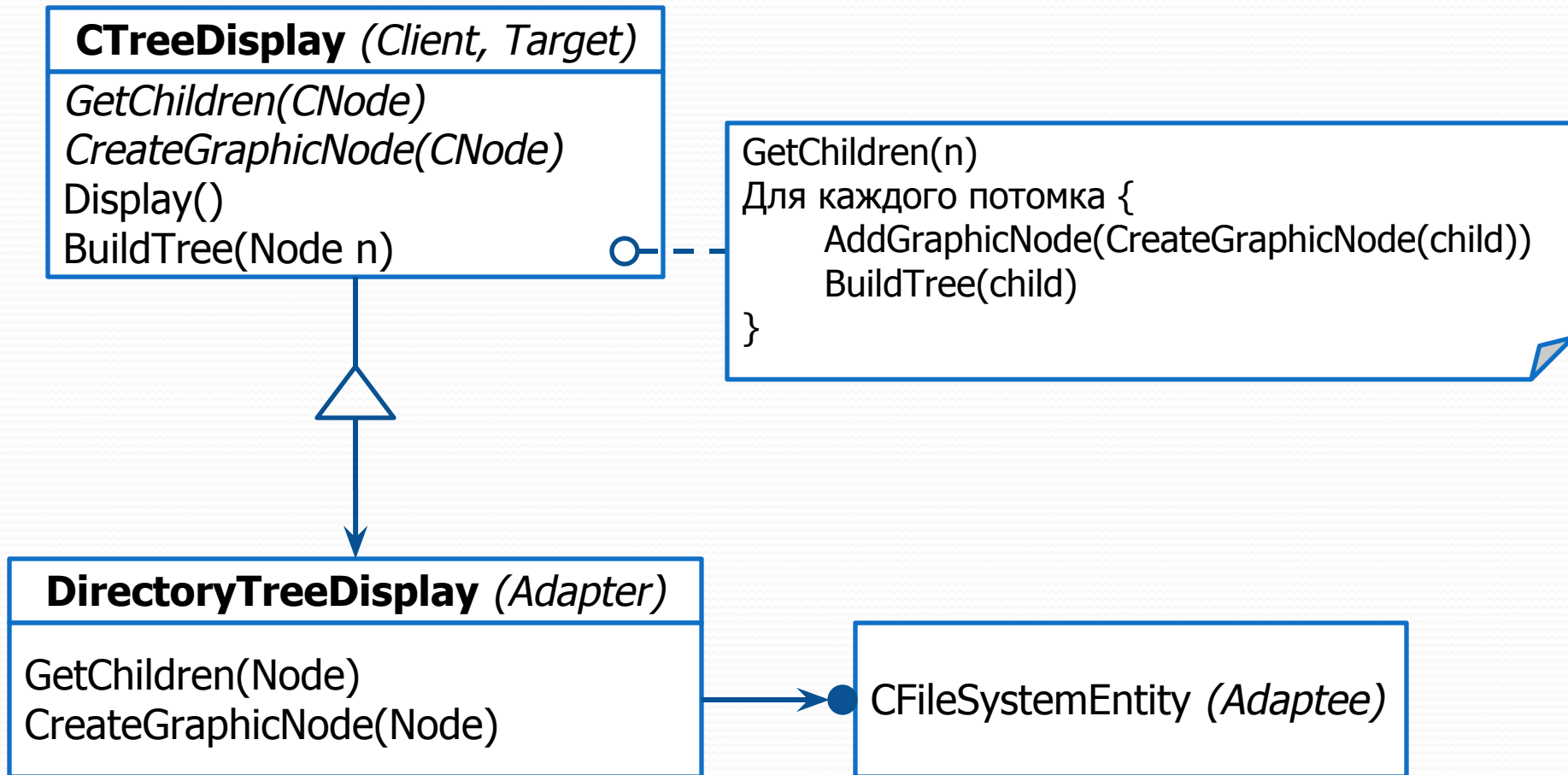
# Реализация сменных адаптеров

- **Шаг 1.** Поиск «узкого» интерфейса для Adaptere
  - Наименьшее подмножество операций, позволяющее выполнить адаптацию
    - Минимальный интерфейс для CTreeDisplay может включать всего две операции
      - Получить графическое представление узла
      - Доступ к потомкам узла
- **Шаг 2.** Выбор одного из следующих подходов к реализации
  - Использование абстрактных операций
  - Использование объектов-уполномоченных

# Подход 1 – «Использование абстрактных операций»

- Определим в классе CTreeDisplay абстрактные операции, соответствующие узкому интерфейсу класса Adaptee
  - Подклассы CTreeDisplay реализовывают данные операции и адаптируют иерархически структурированный объект
    - Класс CDirectoryTreeDisplay будет осуществлять доступ к структуре каталогов файловой системы
    - Класс CDirectoryTreeDisplay специализирует узкий интерфейс таким образом, чтобы он мог отображать структуру каталогов, составленную из объектов CFileSystemEntity

# Структура



# Подход 2 – «Использование объектов-уполномоченных»

- CTreeDisplay выполняет переадресацию запросов к иерархической структуре объекту-уполномоченному
  - Узкий интерфейс объекта-уполномоченного (Целевой объект) помещается в абстрактный класс CTreeAccessorDelegate
  - Класс CDirectoryBrowser (Адаптер) наследуется от CTreeAccessorDelegate, реализуя абстрактные операции уполномоченного объекта

# Структура

