ECE4574 – Large-Scale SW Development for Engineering Systems
Lecture 11 – Design Patterns

Creed Jones, PhD

BRADLEY DEPARTMENT
OF ELECTRICAL & COMPUTER ENGINEERING

VIRGINIA TECH.

Course Updates

- Project
  - You should really be starting to get some code in place
- Quiz 4 is next Monday, October 2

Next Week – Oct 2 and 4

Monday, Oct 2 – Project Day

- Come to class and sit in your project groups
  - Online team members should join by zoom or similar
- Use the day to catch up on your project work
  - Stand-up meeting
  - Start on the sprint retrospective
- I will meet briefly with each group

Wednesday, Oct 4 – NO CLASS

No office hours on Tuesday, Wednesday or Thursday (Oct 3, 4 and 5)

Topics for Today

Design by Pattern

- Categories of Design Patterns

A Few Useful Design Patterns

- Proxy
- Strategy
- Observer
- Decorator
- Singleton
- Adapter

DESIGN BY PATTERN

Design Patterns are well-proven solutions to classes of problems

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

Alexander et al, *A Pattern Language*, quoted in our text

Benefits of using patterns

- Patterns give a design **common vocabulary** for software design:
  - Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - A culture; domain-specific patterns increase design speed.
- **Capture expertise** and allow it to be communicated:
  - Promotes design reuse and avoid mistakes.
  - Makes it easier for other developers to understand a system.
- **Improve documentation** (less is needed):
  - Improve understandability (patterns are described well, once).
  - By convention/best-practices

What is NOT a Design pattern

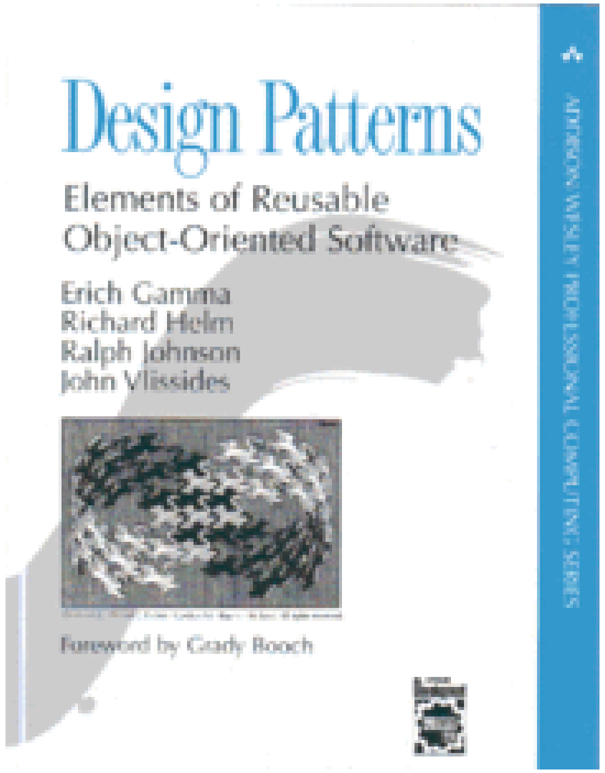
- NOT about specific programming language
- NOT a toolkit
- NOT a framework

- Design patterns are proven solutions to common problems; they are ways to arrange classes and objects to satisfy needs that come up again and again. They are NOT chunks of source code (though there are examples of pattern implementations available)
- Frameworks are partial software solutions that allow for a relatively small amount of coding to customize and complete the overall project

Software Design Patterns

- A group called the *Gang of Four* or "GoF‟ (Gamma, Helm, Johnson, Vlissides) compiled a catalog of design patterns
  - *Their book is a classic in the Software field*
- *I will primarily discuss the patterns that they identified*
  - *Others have been identified and are in use*

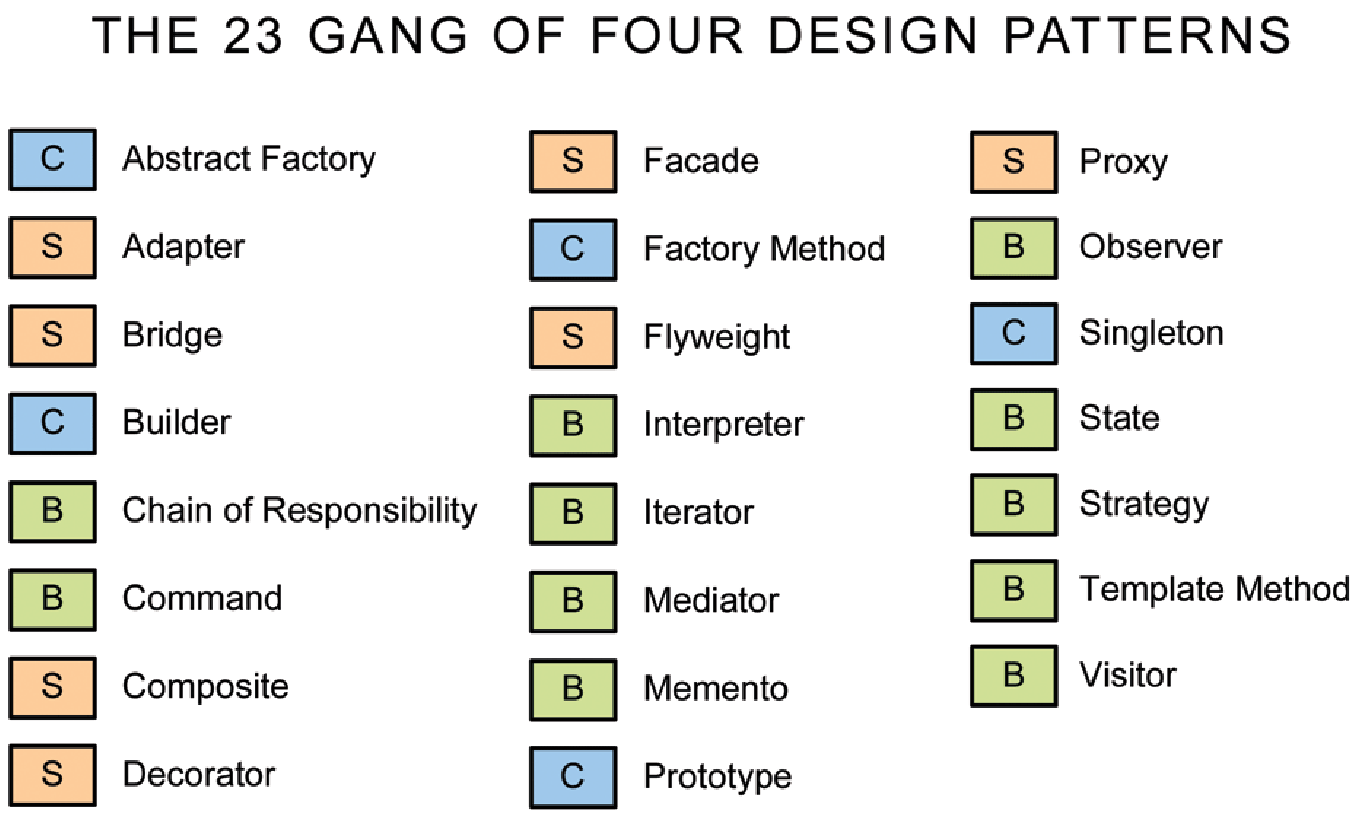
(L-R) Ralph, Erich, Richard and John


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

A Design Pattern has four elements that define it – and determine the level of abstraction that we think and design at

1. Pattern Name – short and descriptive
2. Problem – when the pattern should be applied, and preconditions to its use
3. Solution – the elements of the design: objects, responsibilities, relationships and collaborations
4. Consequences – results and trade-offs

There are 23 basic patterns covered in the "Design Patterns" book, of three types: Creational, Structural and Behavioral

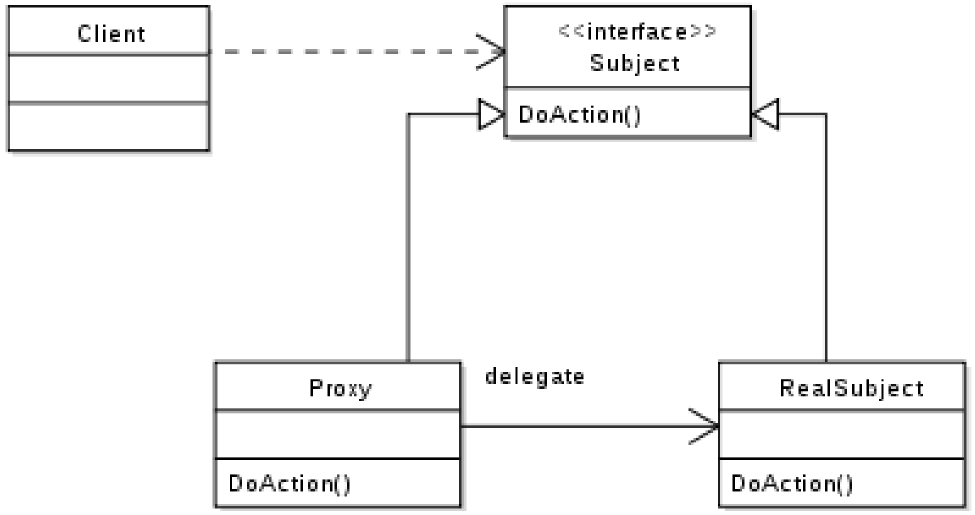


THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory
S	Adapter
S	Bridge
C	Builder
B	Chain of Responsibility
B	Command
S	Composite
S	Decorator
S	Facade
C	Factory Method
S	Flyweight
B	Interpreter
B	Iterator
B	Mediator
B	Memento
C	Prototype
S	Proxy
B	Observer
C	Singleton
B	State
B	Strategy
B	Template Method
B	Visitor

PROXY

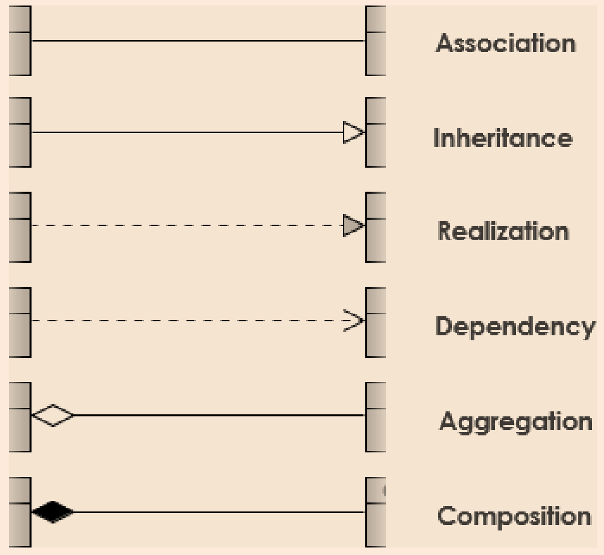
As an example, the Proxy pattern uses an intermediary to allow controlled access to a given object



- *remote proxy:* local representative of an object on another machine
- *virtual proxy:* defer object creation until needed
- *protection proxy:* checks access control rights, etc.

Structural

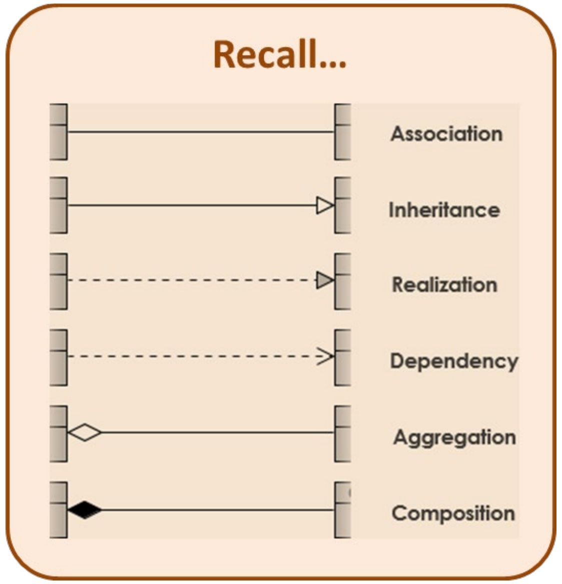
Recall…



Standard Form for Describing Design Patterns

- Pattern Name and Classification
- Intent
- AKA
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Proxy



Structural

Intent

Provide a surrogate or placeholder for another object to control access to it.

Also Known As

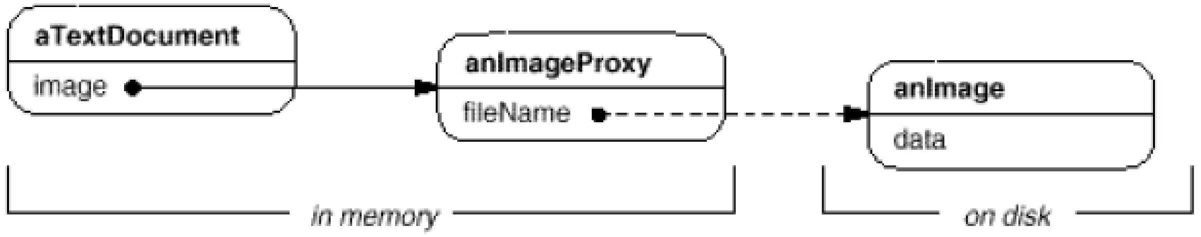
Surrogate

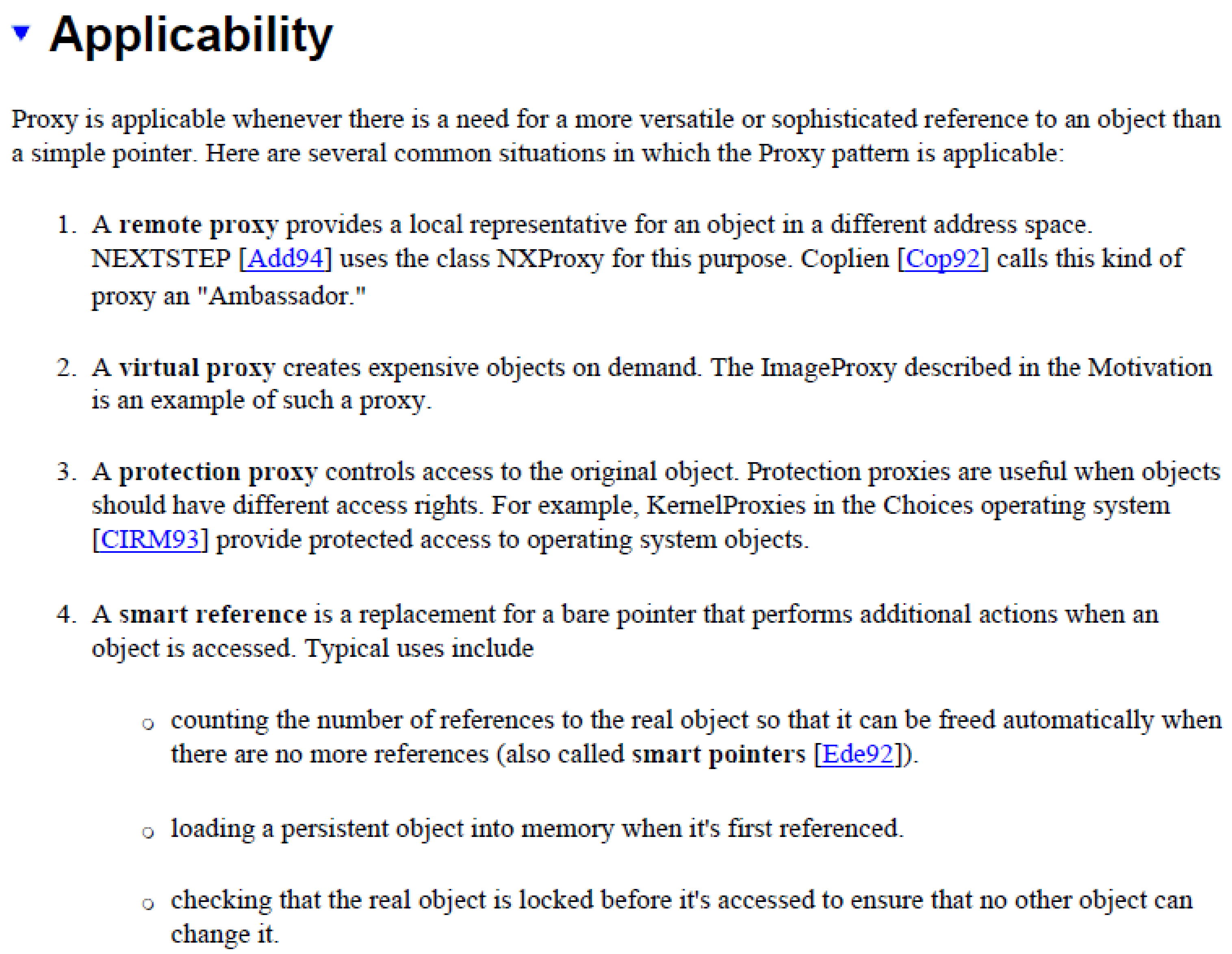
Motivation

One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.

These constraints would suggest creating each expensive object *on demand*, which in this case occurs when an image becomes visible. But what do we put in the document in place of the image? And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation? This optimization shouldn't impact the rendering and formatting code, for example.

The solution is to use another object, an image **proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.





▾ Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."

2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.

3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.

4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include

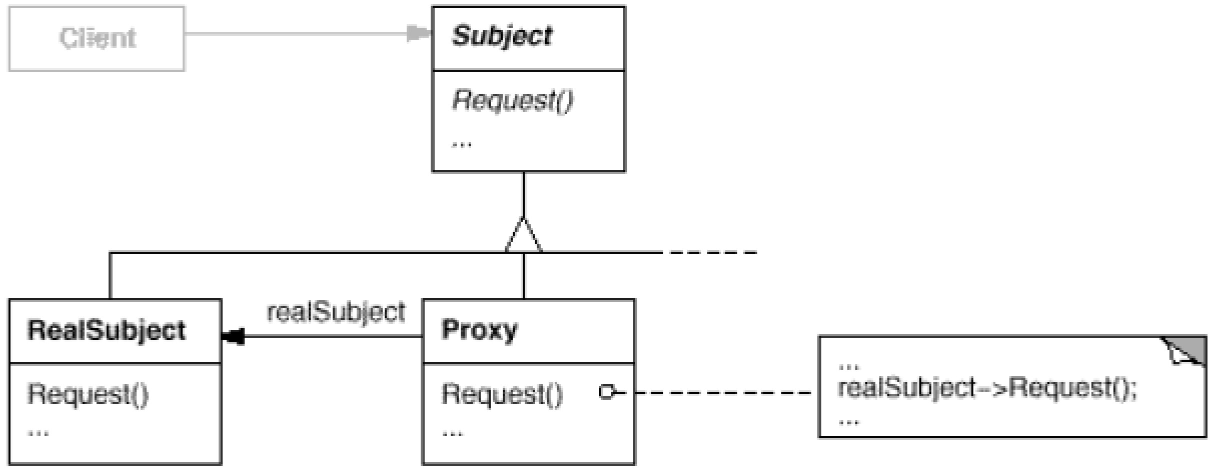
   - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called **smart pointers** [Ede92]).

   - loading a persistent object into memory when it's first referenced.

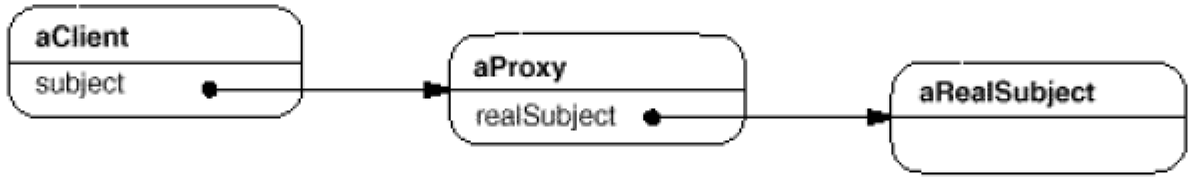
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

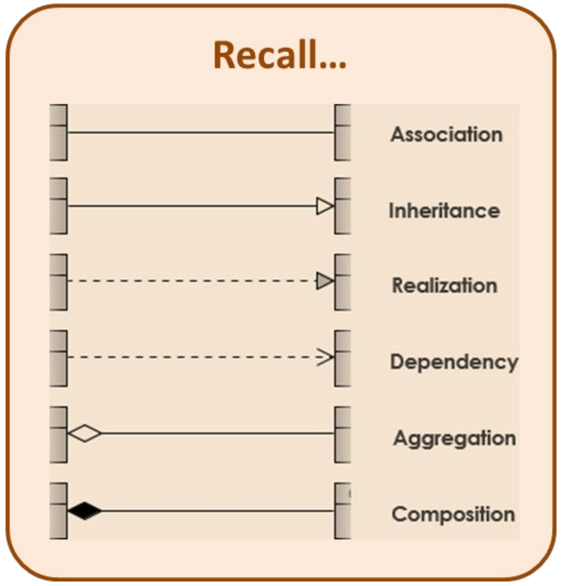
Structural

Structure



Here's a possible object diagram of a proxy structure at run-time:



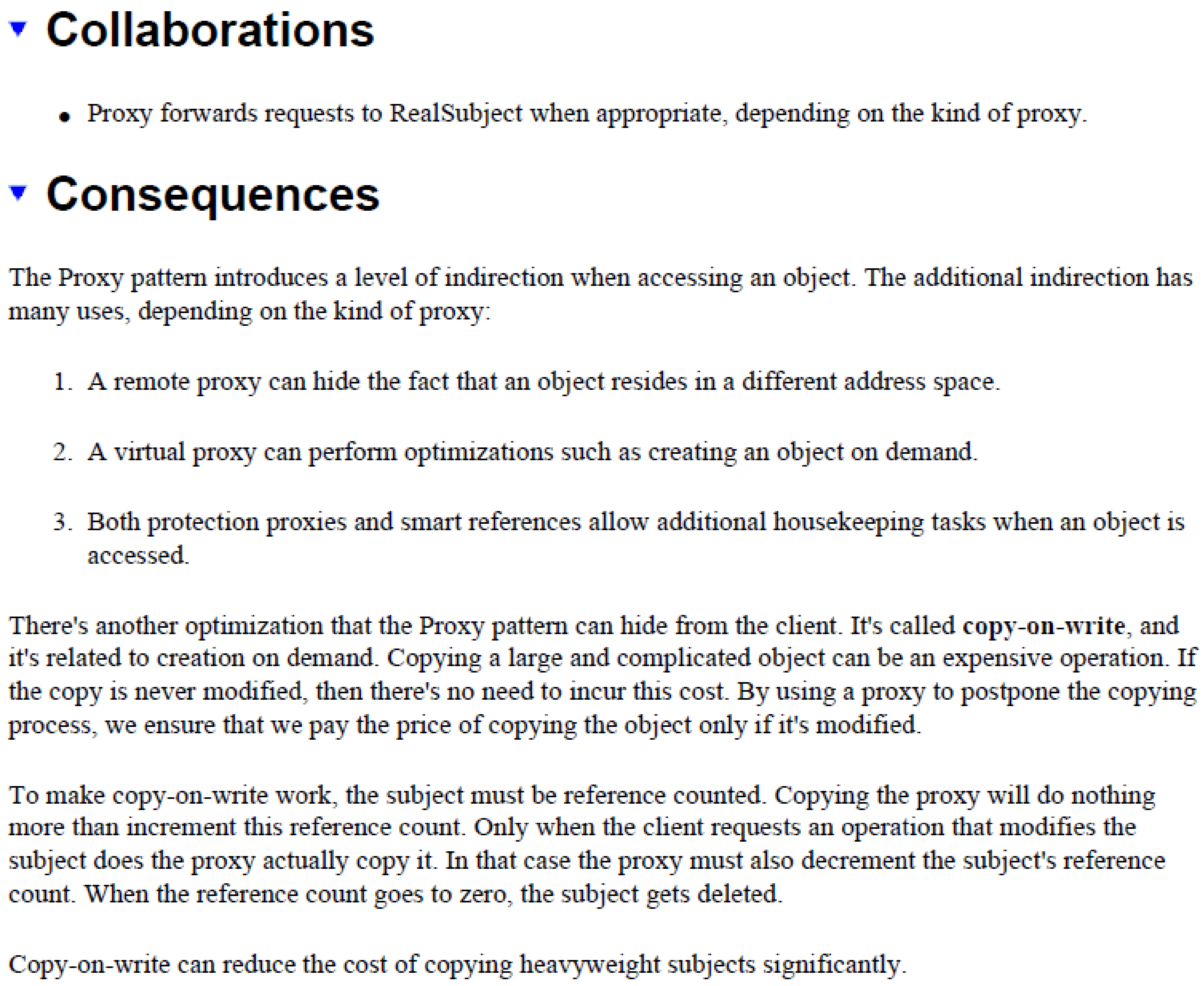


Structural

Participants

- **Proxy** (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
    - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
    - *protection proxies* check that the caller has the access permissions required to perform a request.
- **Subject** (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject** (Image)
  - defines the real object that the proxy represents.

Structural



Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

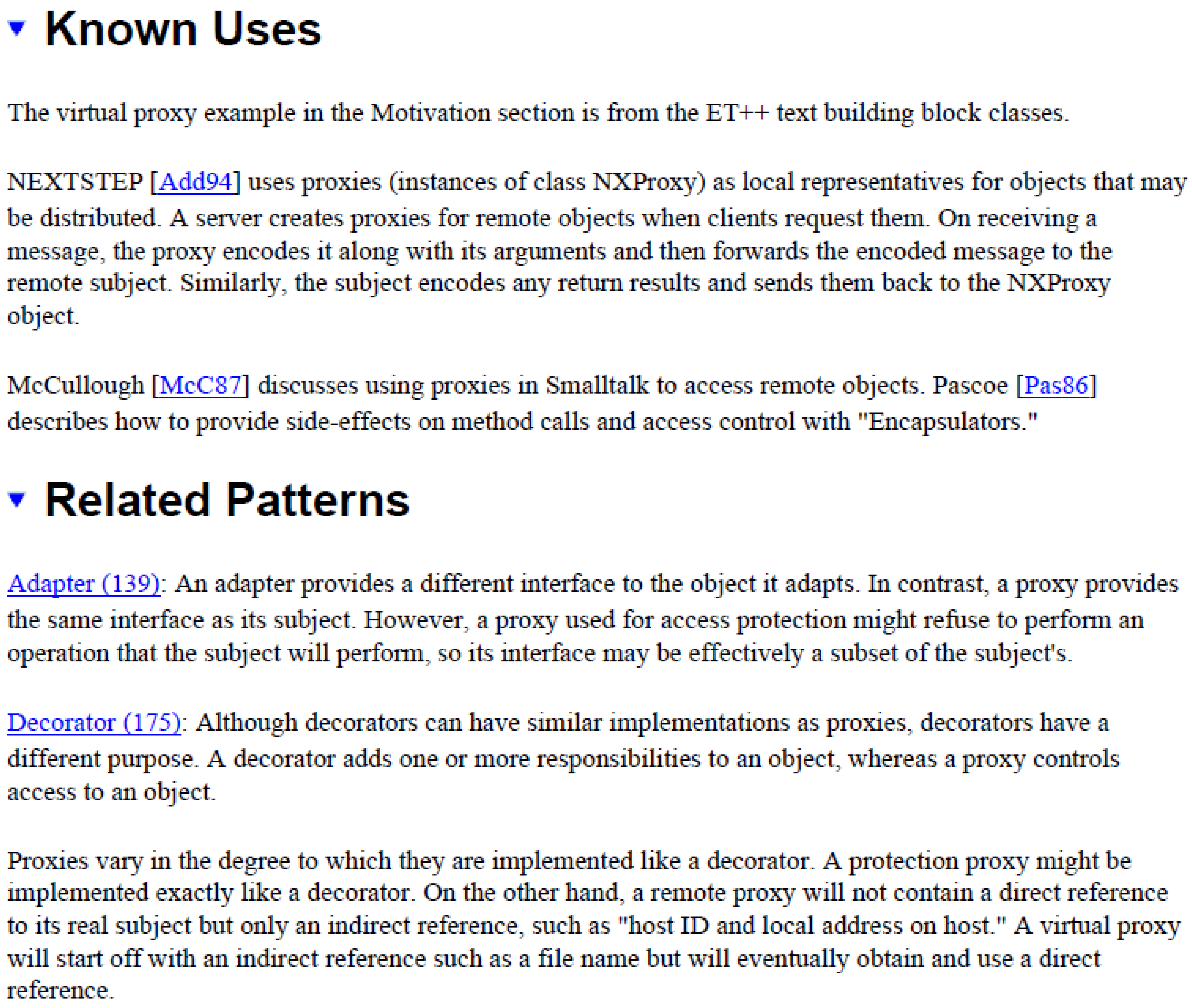
1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write**, and it's related to creation on demand. Copying a large and complicated object can be an expensive operation. If the copy is never modified, then there's no need to incur this cost. By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it's modified.

To make copy-on-write work, the subject must be reference counted. Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it. In that case the proxy must also decrement the subject's reference count. When the reference count goes to zero, the subject gets deleted.

Copy-on-write can reduce the cost of copying heavyweight subjects significantly.

Structural



▾ Known Uses

The virtual proxy example in the Motivation section is from the ET++ text building block classes.

NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed. A server creates proxies for remote objects when clients request them. On receiving a message, the proxy encodes it along with its arguments and then forwards the encoded message to the remote subject. Similarly, the subject encodes any return results and sends them back to the NXProxy object.

McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls and access control with "Encapsulators."

▾ Related Patterns

Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

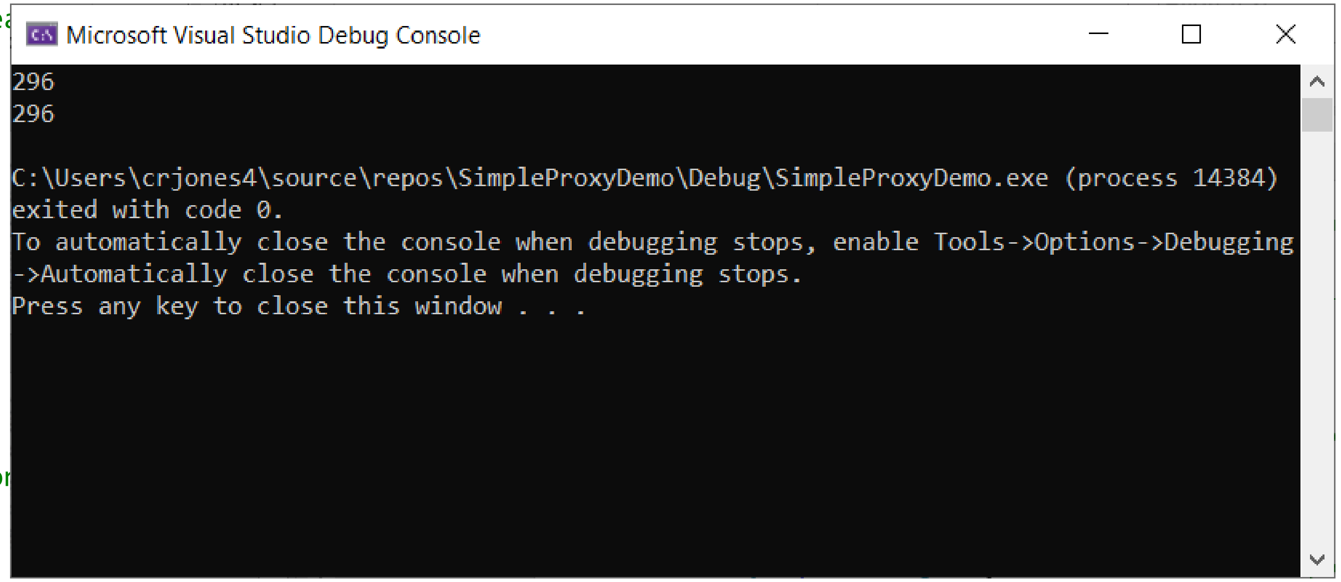
Decorator (175): Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Proxies vary in the degree to which they are implemented like a decorator. A protection proxy might be implemented exactly like a decorator. On the other hand, a remote proxy will not contain a direct reference to its real subject but only an indirect reference, such as "host ID and local address on host." A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

Structural

```
class Subject       //abstract class to define common interface
{
public:
    virtual int getVal() = 0;                   // virtual, so it must be implemented by every subclass
};

class RealSubject : public Subject {
private:
    int myVal = int(this)%1000;                 // real subject has the actual value we want to access
public:
    virtual int getVal() override { return myVal; }
};

class Proxy : public Subject {         // proxy in this case only has the interface - no data element
    RealSubject* pRS;
    void createSubject()     {
        if (pRS == nullptr) { pRS = new RealSubject; }      // create the real subject if it does not exist
    }
public:
    Proxy() { pRS = nullptr; }
    virtual ~Proxy() {}
    virtual int getVal() override {             // first call creates subject
        createSubject();
        return pRS->getVal();
    }
};

void f(Subject& s) {
    std::cout << s.getVal() << std::endl;
}

int main() {
    Proxy proxy;

    f(proxy);                       // call twice - verify that only one real subject is created
    f(proxy);
}
```

```
class Subject       //abstract class to define common interface
{
public:
    virtual int getVal() = 0;                  // virtual, so it must be implemented by every subclass
};

class RealSubject : public Subject {
private:
    int myVal = int(this)%1000;                // real subject has the actual value we want to access
public:
    virtual int getVal() override { return myVal; }
};

class Proxy : public Subject {           // proxy in this case only has the interface - no data element
    RealSubject* pRS;
    void createSubject()     {
        if (pRS == nullptr) { pRS = new RealSubject; }      // create the real subject if it does not exist
    }
public:
    Proxy() { pRS = nullptr; }
    virtual ~Proxy() {}
    virtual int getVal() override {             // first call crea
        createSubject();
        return pRS->getVal();
    }
};

void f(Subject& s) {
    std::cout << s.getVal() << std::endl;
}

int main() {
    Proxy proxy;

    f(proxy);                          // call twice - verify that on
    f(proxy);
}
```

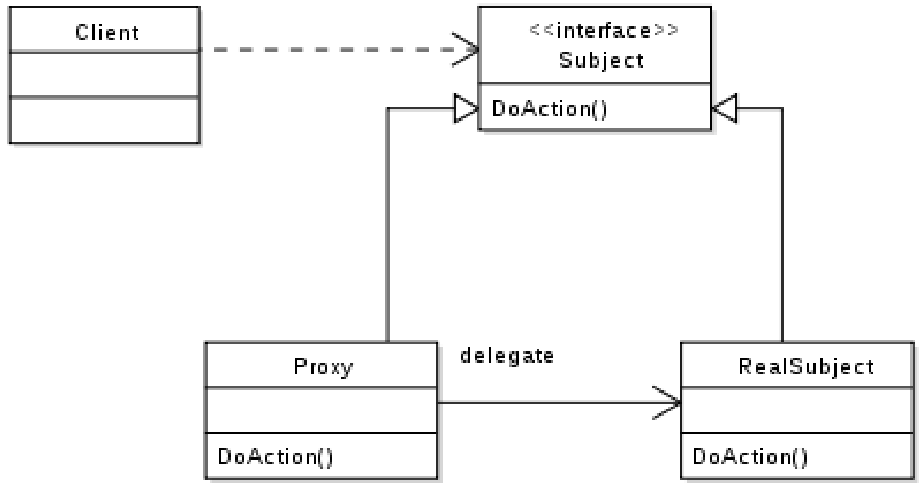

```
296
296

C:\Users\crjones4\source\repos\SimpleProxyDemo\Debug\SimpleProxyDemo.exe (process 14384)
exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging
->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Design Patterns are ideal for solving design issues because good design practice and working techniques are already built-in

- Many of the patterns exist to implement good design behavior (encapsulation, loose coupling, etc)
- They are at a larger scale than classes, but smaller than components and frameworks
- They provide a language for designs:
  "I used a factory in that component"

Using a Design Pattern

1. Read the pattern documentation
2. Study the Structure, Participants and Collaborations sections
3. Look at (and maybe copy) the sample code
4. Choose names for participants in your design
5. Define the classes
6. Define application-specific names for operations (methods)
7. Implement the operations (not part of the design phase)



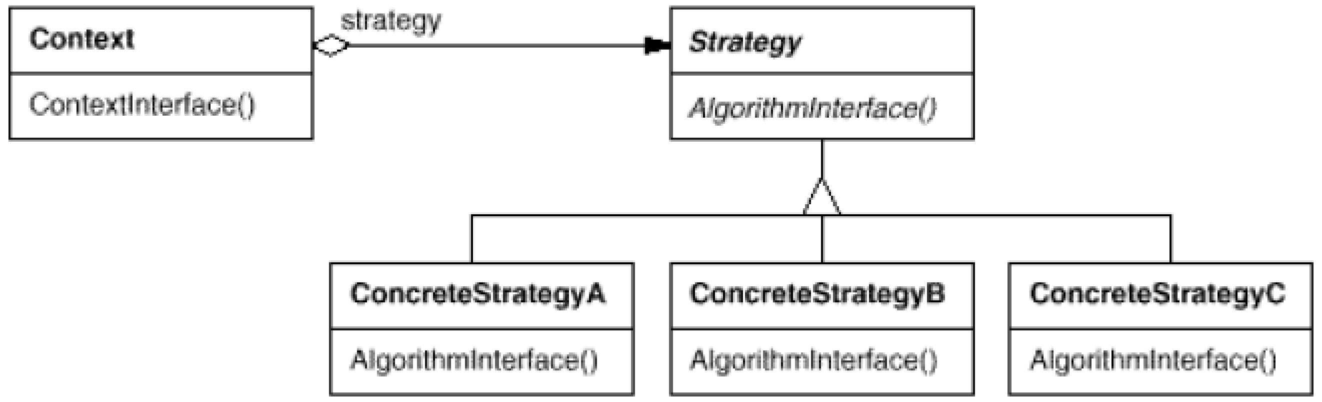
Selecting a Design Pattern can be challenging at first, but their usefulness grows with experience

- Identify the problems your design may face
- Become familiar with common design patterns
- See how others use them
- Study how patterns interrelate
- If you ever need to redesign a project, look into why and what might have prevented the need

- Design Patterns support good Design Principles…

STRATEGY

The Strategy pattern encapsulates a set of algorithms, which can vary independent of the client's interface



The different states are instantiated in different derived classes, each with different behavior – providing a common interface through the Handle() function

Behavioral

Strategy participants

Strategy

- declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy

- implements the algorithm using the Strategy interface.

Context

- is configured with a ConcreteStrategy object.
- maintains a reference to a Strategy object.
- may define an interface that lets Strategy access its data.

Collaborations

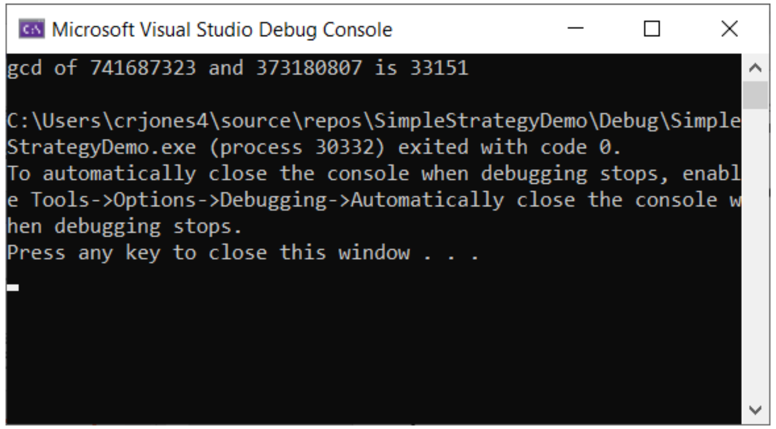
- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively.
- There is often a family of ConcreteStrategy classes for a client to choose from.

Behavioral

```
class gcdStrategy   // abstract class defining the gcdStrategy interface
                    // one public function: gcd(a, b) returns the answer
{
public:
    virtual int gcd(int a, int b) = 0;
protected:
    void orderMinMax(int& a, int& b) {  // swaps so a >= b
        if (b > a) {
            int temp = b; b = a; a = temp;
        }
    }
    bool isEven(int a) {
        return ((a % 2) == 0) ? true : false;
    }
    const int HUGEVAL = 2000000000;
};

class BinaryConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: Binary iteration method
private:
    int gcd(int a, int b) {
        orderMinMax(a, b);
        int stepCount = 0;
        while (stepCount < HUGEVAL) {
            if (a == b)
                return a * pow(2, stepCount);
            else if (isEven(a) && isEven(b)) {
                a /= 2;
                b /= 2;
                stepCount++;
            }
            else if (isEven(a))
                a /= 2;
            else if (isEven(b))
                b /= 2;
            else {
                orderMinMax(a, b);
                a = (a - b) / 2;
            }
        }
        return a;
    }
};
```

```
class EuclidConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: Euclid's method
private:
    int gcd(int a, int b) {
        orderMinMax(a, b);
        while (b > 0) {
            int temp = a % b;
            a = b;
            b = temp;
        }
        return a;
    }
};


class BruteForceConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: EExhaustive search
private:
    int gcd(int a, int b) {
        int answer = 0;
        orderMinMax(a, b);
        for (int incr = 1; incr < a; incr++) {
            if ((a % incr == 0) && (b % incr == 0))
                answer = incr;
        }
        return answer;
    }
};


int main() {
    vector<gcdStrategy*> sVec;      // vector of pointers to the abstract Strategy
class
    sVec.push_back(new EuclidConcStrategy());   // add three different Concrete
algorithms
    sVec.push_back(new BinaryConcStrategy());   // to the vector
    sVec.push_back(new BruteForceConcStrategy());
    int num1 = 262144, num2 = 131072;
    for (int incr = 0; incr < sVec.size(); incr++)  // exercise each algorithm
        cout << "gcd of " << num1 << " and " << num2 << " is " << sVec.at(incr)-
>gcd(num1, num2) << endl;
    return 0;
}
```

```
class gcdStrategy   // abstract class defining the gcdStrategy interface
                    // one public function: gcd(a, b) returns the answer
{
public:
    virtual int gcd(int a, int b) = 0;
protected:
    void orderMinMax(int& a, int& b) { // swaps so a >= b
        if (b > a) {
            int temp = b; b = a; a = temp;
        }
    }
    bool isEven(int a) {
        return ((a % 2) == 0) ? true : false;
    }
    const int HUGEVAL = 2000000000;
};

class BinaryConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: Binary iteration method
private:
    int gcd(int a, int b) {
        orderMinMax(a, b);
        int stepCount = 0;
        while (stepCount < HUGEVAL) {
            if (a == b)
                return a * pow(2, stepCount);
            else if (isEven(a) && isEven(b)) {
                a /= 2;
                b /= 2;
                stepCount++;
            }
            else if (isEven(a))
                a /= 2;
            else if (isEven(b))
                b /= 2;
            else {
                orderMinMax(a, b);
                a = (a - b) / 2;
            }
        }
        return a;
    }
};
```

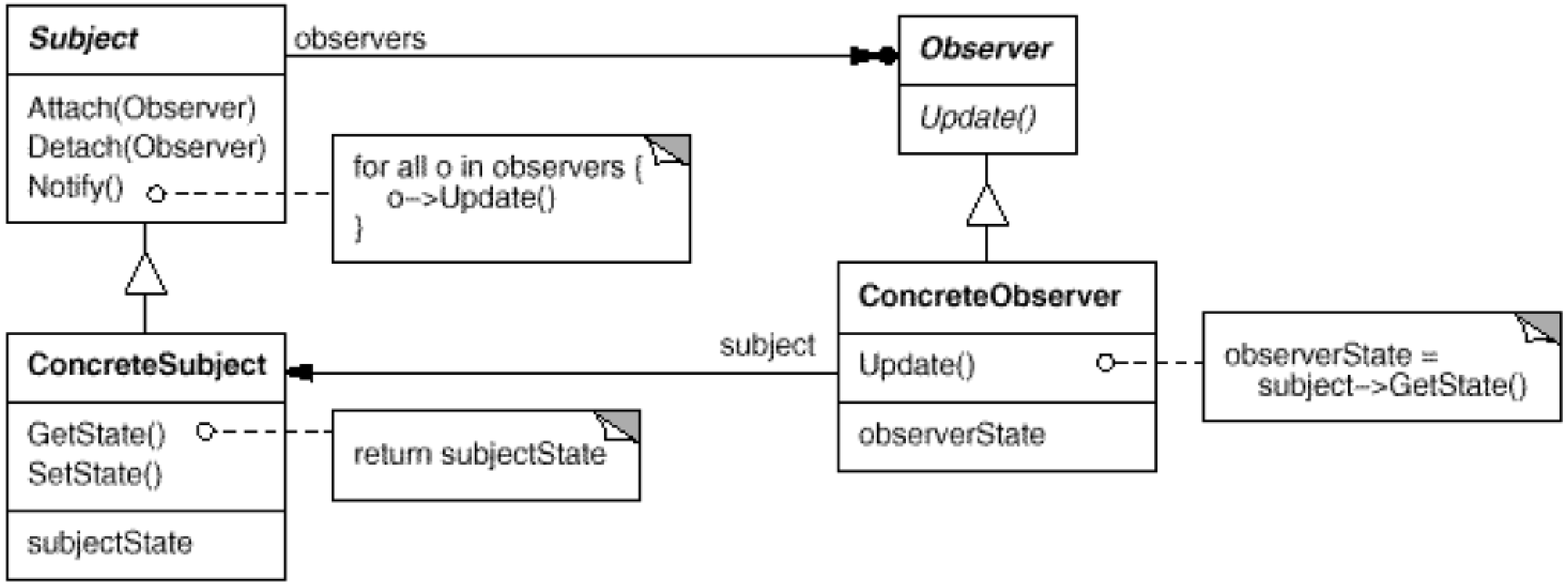


```
gcd of 741687323 and 373180807 is 33151

C:\Users\crjones4\source\repos\SimpleStrategyDemo\Debug\Simple
StrategyDemo.exe (process 30332) exited with code 0.
To automatically close the console when debugging stops, enabl
e Tools->Options->Debugging->Automatically close the console w
hen debugging stops.
Press any key to close this window . . .
```

```
class EuclidConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: Euclid's method
private:
    int gcd(int a, int b) {
        orderMinMax(a, b);
        while (b > 0) {
            int temp = a % b;
            a = b;
            b = temp;
        }
        return a;
    }
};


class BruteForceConcStrategy : public gcdStrategy
{       // concrete class implementing one algorithm: EExhaustive search
private:
    int gcd(int a, int b) {
        int answer = 0;
        orderMinMax(a, b);
        for (int incr = 1; incr < a; incr++) {
            if ((a % incr == 0) && (b % incr == 0))
                answer = incr;
        }
        return answer;
    }
};


int main() {
    vector<gcdStrategy*> sVec;      // vector of pointers to the abstract Strategy
class
    sVec.push_back(new EuclidConcStrategy());   // add three different Concrete
algorithms
    sVec.push_back(new BinaryConcStrategy());   // to the vector
    sVec.push_back(new BruteForceConcStrategy());
    int num1 = 262144, num2 = 131072;
    for (int incr = 0; incr < sVec.size(); incr++)  // exercise each algorithm
        cout << "gcd of " << num1 << " and " << num2 << " is " << sVec.at(incr)-
>gcd(num1, num2) << endl;
    return 0;
}
```

OBSERVER

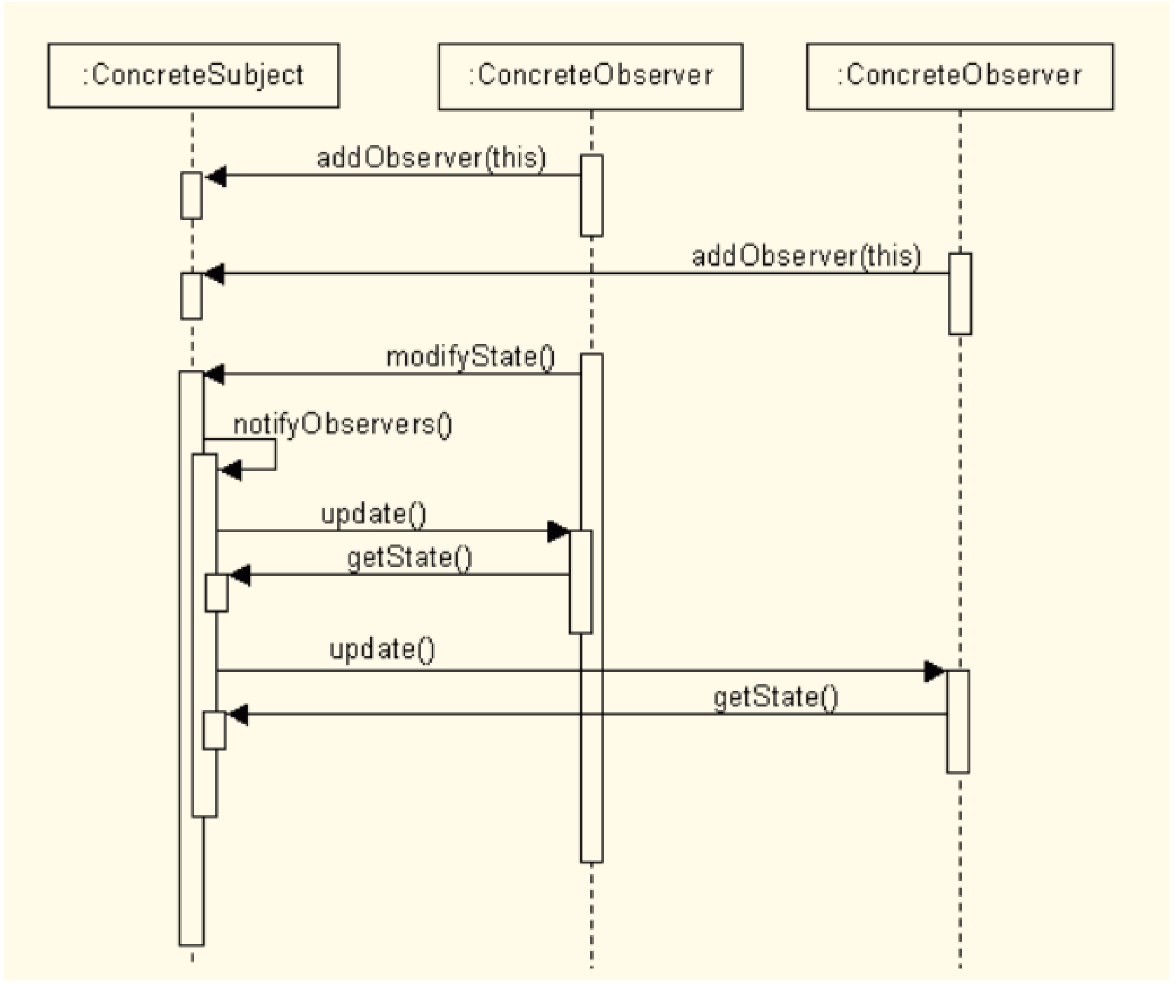
The Observer pattern relates an object that may change to other objects that will be notified of the change



Can also be referred to as a "Publish-Subscribe" arrangement

Behavioral

The UML Sequence diagram for an Observer pattern shows that addObserver() (or *subscribe*) precedes any notification



Behavioral

Observer participants

Subject
- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

Observer
- defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject
- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

ConcreteObserver
- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's

Collaborations
- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Behavioral

Loose Coupling is an important attribute of a good software design; objects don’t have or need detailed information about other objects, and don’t interact with them excessively

Loose coupling

```
class CustomerRepository
{
private:
        bool readonly;
        IDatabase iDatabase;

public:
      CustomerRepository(IDatabase const
& database):
              iDatabase(database)
      {
      }
      void Add(string CustomerName)
      {
            iDatabase.AddRow("Customer",
CustomerName);
      }
};
```

```
class IDataBase
{
public:
        virtual void AddRow(string
Table, string Value) = 0;
}
```

```
class Database : public IDataBase
{
public:
      void AddRow(string Table, string
Value)
        {
        }
};
```

Tight coupling

```
class CustomerRepository
{
private:
        bool readonly;
        Database database;

public:
      CustomerRepository(Database const &
database):
              database(database)
      {
      }
      void Add(string const & CustomerName)
      {
            database.AddRow("Customer",
CustomerName);
      }
};
```

```
class Database
{
public:
      void AddRow(string const &
Table, string const & Value)
        {
        }
};
```

DECORATOR

The Decorator pattern is used to add functionality (responsibilities) to an object at run-time.

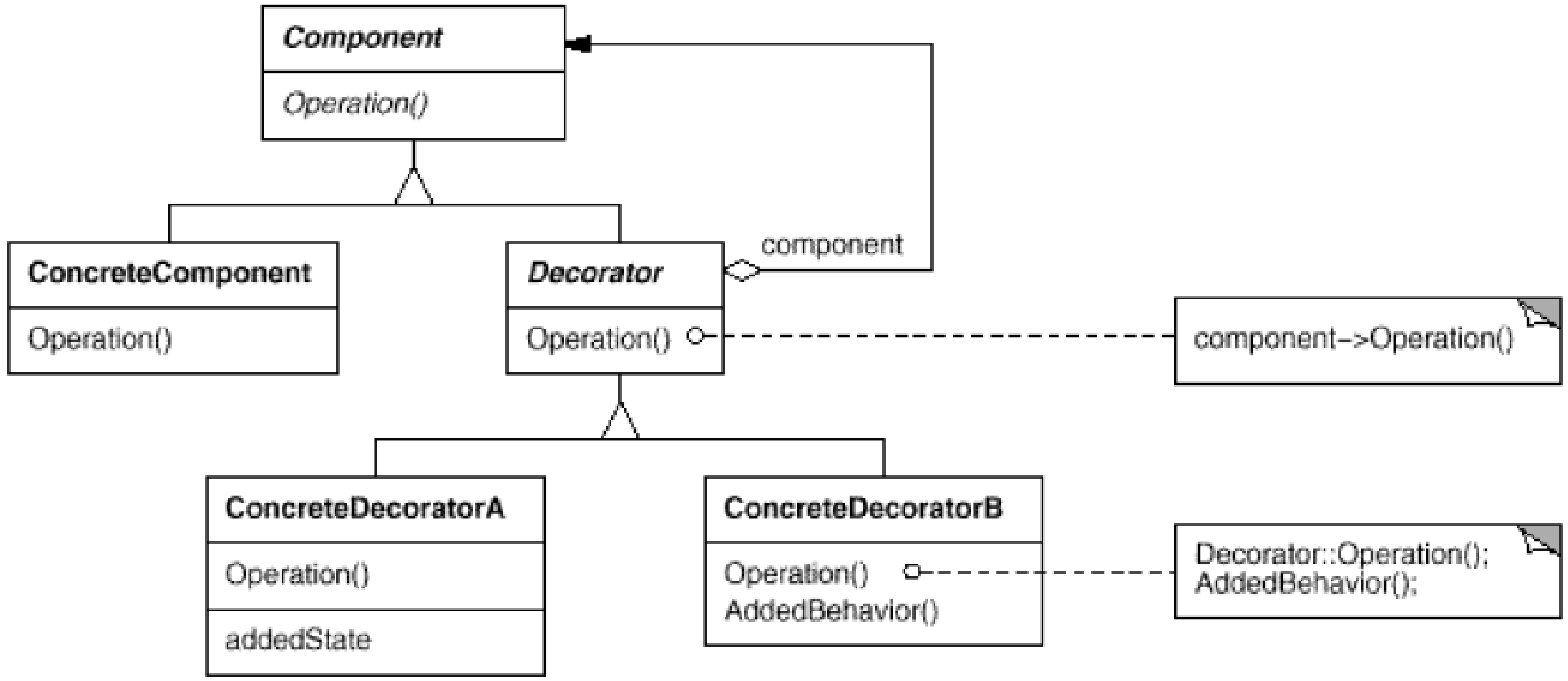
- It's also called a Wrapper

- The essence is that the base class (the Component) defines the interface, but multiple derived classes with added "stuff" may be instantiated and addressed thereby
- Of course, addressing through base class references won't expose any additional functions
  - That would require addressing in terms of the derived class

- Can be more flexible than simple inheritance

Structural

The Decorator pattern



Structural

What is meant by “decorating” in this context?

- What is being decorated?
- What are the “decorations” added?
- What design requirements would imply decoration?

Structural

What is meant by “decorating” in this context?

- What is being decorated?
- An object – at run-time

- What are the “decorations” added?
- Additional responsibilities (behaviors and/or state)

- What design requirements would imply decoration?
- Multiple and overlapping variations of objects – especially when they can change dynamically

Structural

Participants in the Decorator pattern

Component

- defines the interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent

- defines an object to which additional responsibilities can be attached.

Decorator

- maintains a reference to a Component object and defines an interface that conforms to Component's interface.

ConcreteDecorator

- adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Structural

Some points about the decorator pattern (and the concept of decoration of objects)

- KEY DESIGN PRINCIPLE: Classes should be open for extension but closed for modification
  - USUALLY (not always)
  - Careful when overriding superclass behavior!

- Decorators inherit from the same superclass as the objects they decorate.
- Many decorators can be applied to the same object.
- Decorated objects can be passed by reference to the superclass (of course)
- Object decoration can happen at run-time
- New behavior added through decoration can occur either before or after basic (undecorated) behavior

Structural

Proxy, Adapter and Decorator patterns all are used to indirectly access some existing object – but there are differences

- Proxy provides the same interface as the core object.
  - But can offload or delay time-consuming work, or condition or test data.
- Adapter provides a different interface to its subject.
  - Different methods and/or different input/output data.
- Decorator provides an enhanced interface.
  - Supports the same interface as the core object, but also provides new methods and/or new functionality.

Structural

Creational Patterns are used to implement the instantiation process – generally providing greater abstraction

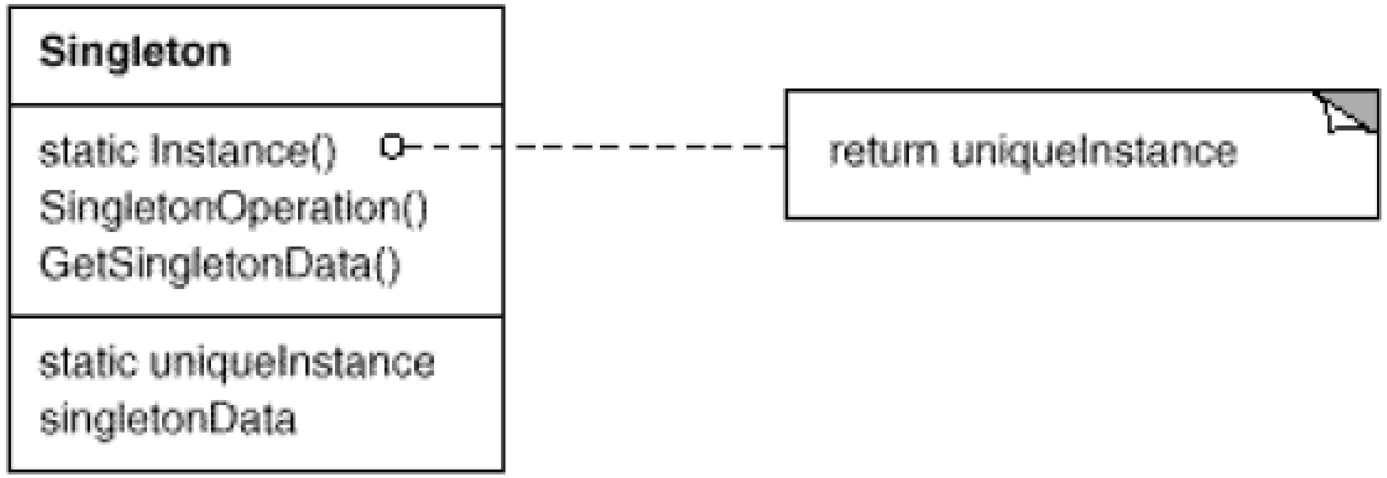
- Keep implementation details as contained as possible
- Make it easy to change platform/components/design

- Two common themes:
- Encapsulate knowledge about which concrete classes are being used.
- Hide the way in which instances of these classes are created and put together.

- Interfaces are specified in one or more abstract classes, which derived classes must implement, but clients can then use.

SINGLETON

The Singleton pattern shows how to create a class for which there can only be one instance



- uniqueInstance is the actual object – created by a call to Instance()
  - <u>the constructor is protected</u>
- A pointer or reference to the uniqueInstance is returned by Instance() and is used to access the object
  - Usually the method is named getInstance()

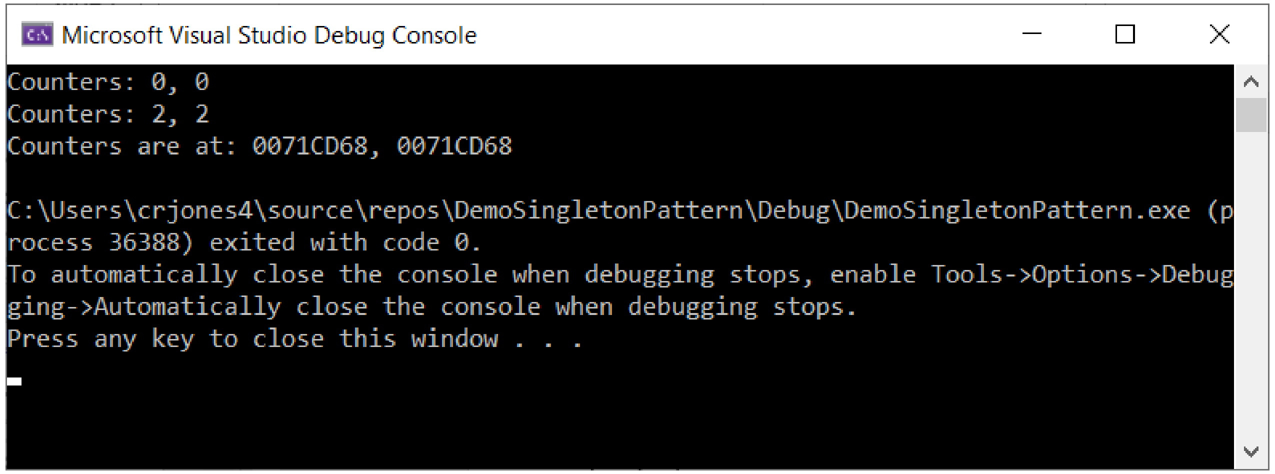
Creational

```
#include <iostream>

class CounterSingleton {
          // define the singleton class
private:
    static int counter;
    static CounterSingleton* counterInstance;
protected:
    CounterSingleton() {
          // constructor protected - we can't call it
        CounterSingleton::counter = 0;
    }
public:
    static CounterSingleton* getInstance() {
          // only create objects thru this static fcn
        if (counterInstance == NULL)
            counterInstance = new CounterSingleton();
          // and only if it doesn’t yet exist
        return counterInstance;
          // otherwise return existing object
    }
    void increment() {
        counter++;
    }
    int getValue() {
        return counter;
    }
};
```

```
int CounterSingleton::counter;
CounterSingleton* CounterSingleton::counterInstance;

int main()
{
    CounterSingleton* ctr1 = CounterSingleton::getInstance();
    CounterSingleton* ctr2 = CounterSingleton::getInstance();
    std::cout << "Counters: " << ctr1->getValue() << ", " <<
ctr2->getValue() << std::endl;
    ctr1->increment();
    ctr2->increment();
    std::cout << "Counters: " << ctr1->getValue() << ", " <<
ctr2->getValue() << std::endl;
    std::cout << "Counters are at: " << ctr1 << ", " << ctr2 <<
std::endl;
}
```

```
#include <iostream>

class CounterSingleton {
          // define the singleton class
private:
    static int counter;
    static CounterSingleton* counterInstance;
protected:
    CounterSingleton() {
          // constructor protected - we can't call it
        CounterSingleton::counter = 0;
    }
public:
    static CounterSingleton* getInstance() {
          // only create objects thru this static fcn
        if (counterInstance == NULL)
            counterInstance = new CounterSingleton();
          // and only if it doesn’t yet exist
        return counterInstance;
          // otherwise return existing object
    }
    void increment() {
        counter++;
    }
    int getValue() {
        return counter;
    }
};
```

```
int CounterSingleton::counter;
CounterSingleton* CounterSingleton::counterInstance;

int main()
{
    CounterSingleton* ctr1 = CounterSingleton::getInstance();
    CounterSingleton* ctr2 = CounterSingleton::getInstance();
    std::cout << "Counters: " << ctr1->getValue() << ", " <<
ctr2->getValue() << std::endl;
    ctr1->increment();
    ctr2->increment();
    std::cout << "Counters: " << ctr1->getValue() << ", " <<
ctr2->getValue() << std::endl;
    std::cout << "Counters are at: " << ctr1 << ", " << ctr2
<< std::endl;
}
```


Microsoft Visual Studio Debug Console

```
Counters: 0, 0
Counters: 2, 2
Counters are at: 0071CD68, 0071CD68

C:\Users\crjones4\source\repos\DemoSingletonPattern\Debug\DemoSingletonPattern.exe (p
rocess 36388) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debug
ging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

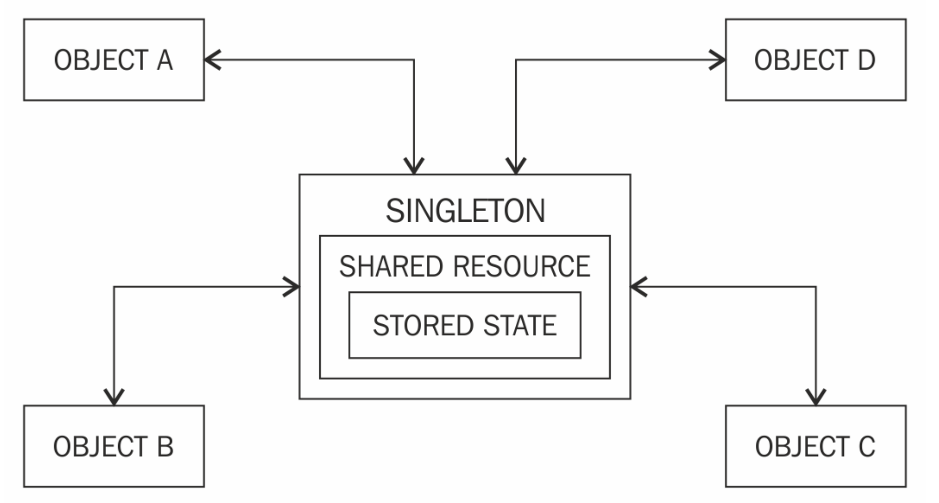
There are three reasons to use a Singleton; access control, distributed use and only one instance

A Singleton may be right if:

- there can be only one object;
- the object controls concurrent access to a shared resource;
- access to the resource will be requested from multiple, disparate parts of the system.

- Might use a Singleton for a system logging object.

- This pattern is controversial – some say that it's unnecessary in well-written code.



Creational

ADAPTER

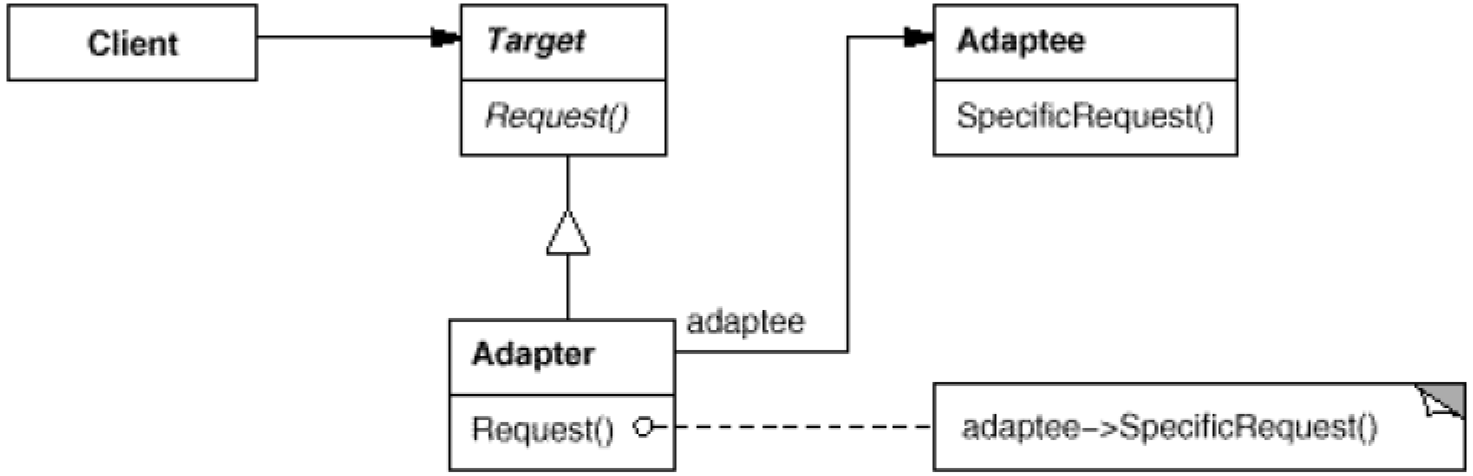
The Adapter pattern allows a class with one interface to be used by clients that expect another

- Often when reusing code, the existing interface needs to be modified to fit with a newer design
- Interfaces may need to be adapted when supporting multiple implementations or algorithms
- Editing a working class directly, to modify the interface, is a bad idea

- The Adapter provides an in-between layer to transform one interface into another
- This adaptation can happen at the Class or Object level…

Structural

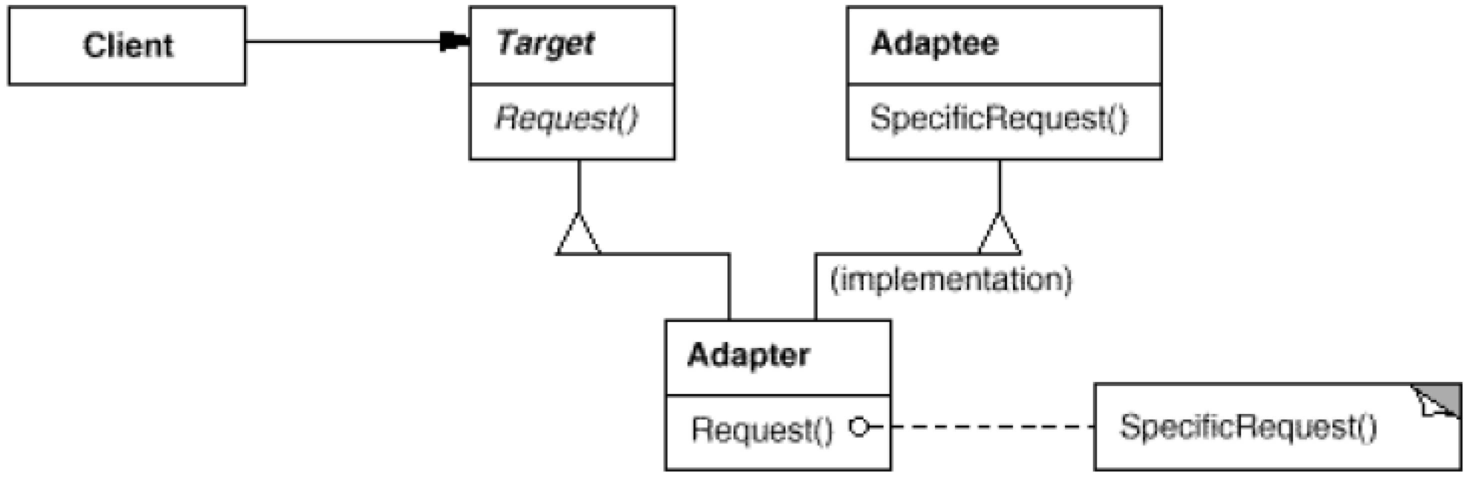
The Object Adapter Pattern



- The Adapter inherits from Target but references an Adaptee object
  - Calls to Request() generate calls to one or more SpecificRequest() functions, and probably some additional code is executed (in Adapter)

Structural

The Class Adapter Pattern



- The Adapter inherits from <u>both</u> the Target and Adaptee classes
  - The Request() interface is what the client sees
  - The SpecificRequest() interface is what we are adapting (what exists in the class to be adapted)

Structural

Participants in the Adapter Pattern

Target

- defines the domain-specific interface that Client uses.

Client

- collaborates with objects conforming to the Target interface.

Adaptee

- defines an existing interface that needs adapting.

Adapter

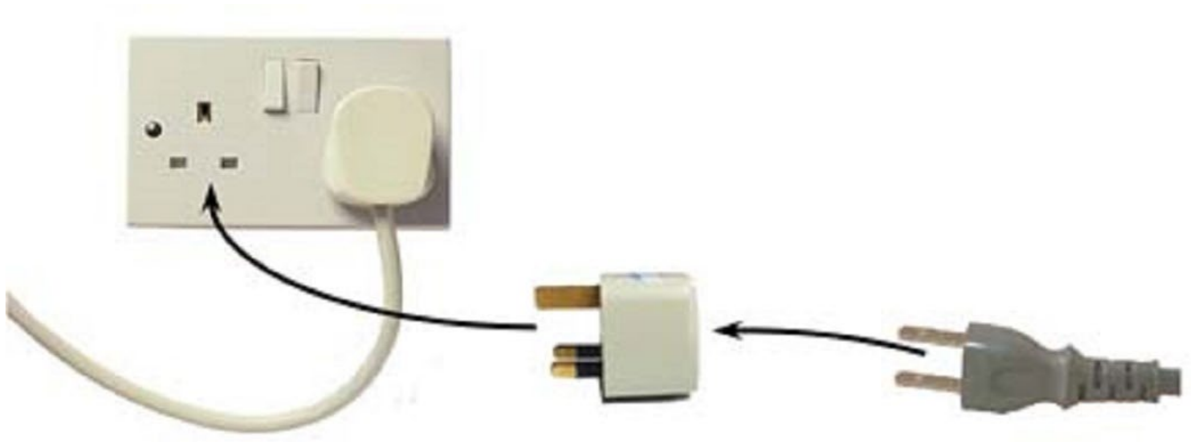
- adapts the interface of Adaptee to the Target interface.

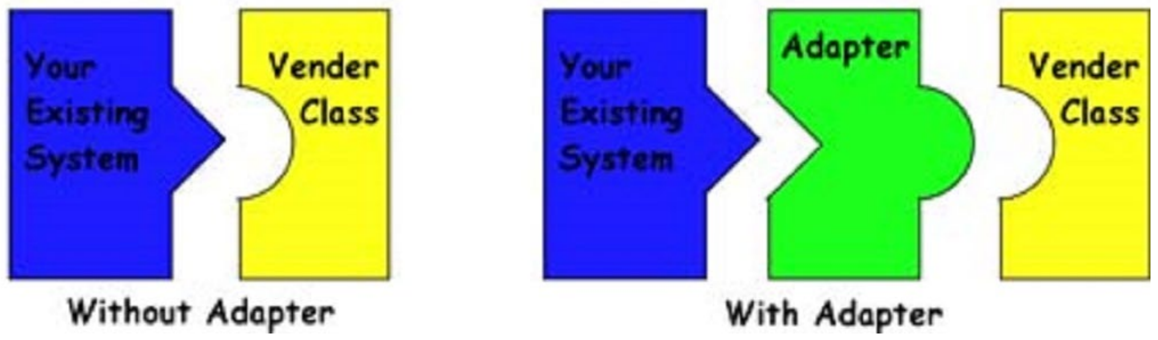
Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Structural

If the object to actually be accessed is called the *Vendor*, an adapter is needed for clients requiring a different interface





Structural

Topics for Today

Design by Pattern

- Categories of Design Patterns

A Few Useful Design Patterns

- Proxy
- Strategy
- Observer
- Decorator
- Singleton
- Adapter