Structural Patterns

Proxy

Proxy

- Intent
  - Provide a **placeholder** for another object to **control access** to it
  - A.k.a. Surrogate
  - Put a placeholder when direct access to an object is not desirable or practical due to
    - Security
    - Performance optimization
    - …

Proxy

- Intent
  - Provide a placeholder for another object to control access to it
- A Proxy object controls access to another object, which may be
  - remote
  - expensive to create
  - in need of protection, require permission
  - Or need a smart reference doing more than a simple pointer

Proxy patterns

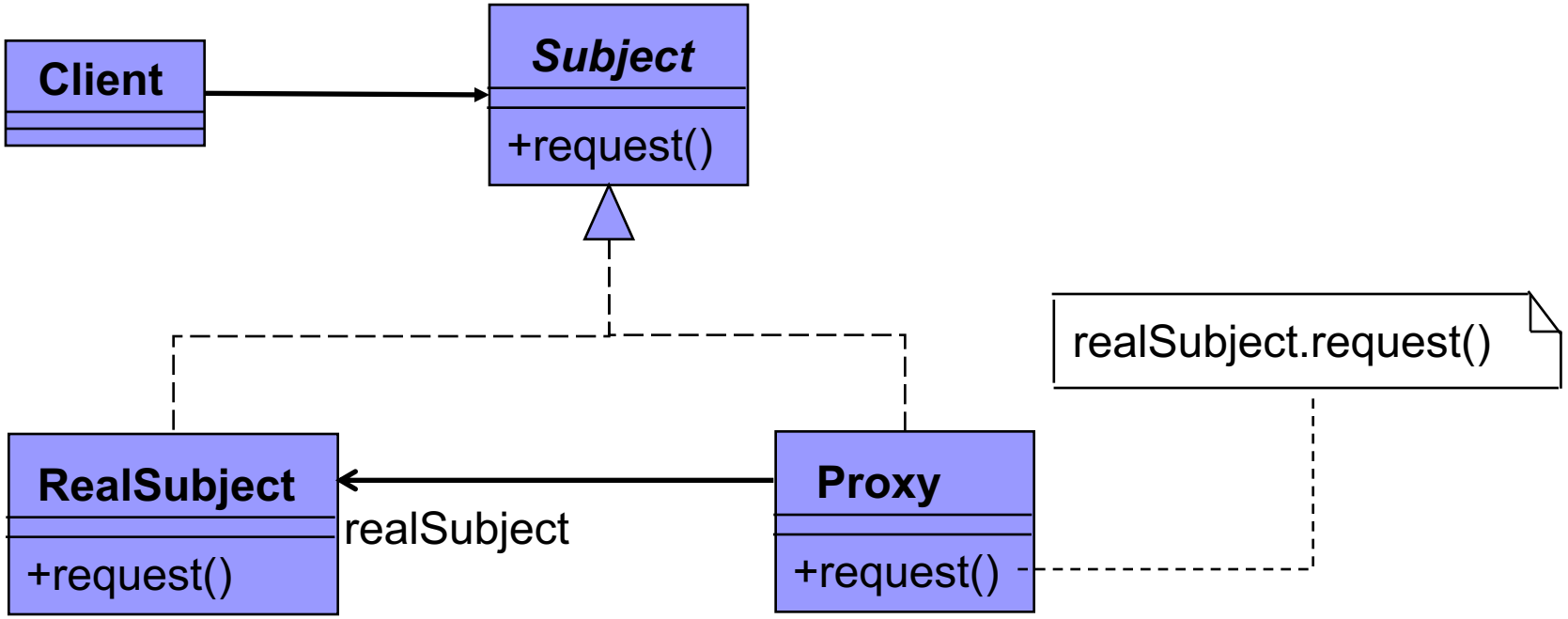
- Real object is remote
  - Remote proxy: illusion that an object on a different machine is a local
- Real object is expensive to create
  - Virtual proxy: creates object lazily
- in need of protection/security
  - Protection proxy: checks permissions and policies
- Resource optimization
  - Caching proxy: delivers cached result
- Pointer with extra responsibility
  - Smart reference

Applicability

Whenever there is a need for a more versatile reference to an object than a simple pointer.

- Remote proxy: when real object is in a different address space
- Virtual proxy: when creation of an object is costly
- Protection proxy: when objects should have different access rights
- Smart reference: when you need for a more versatile reference to an object than a simple pointer
  - E.g. Count number of references and then delete

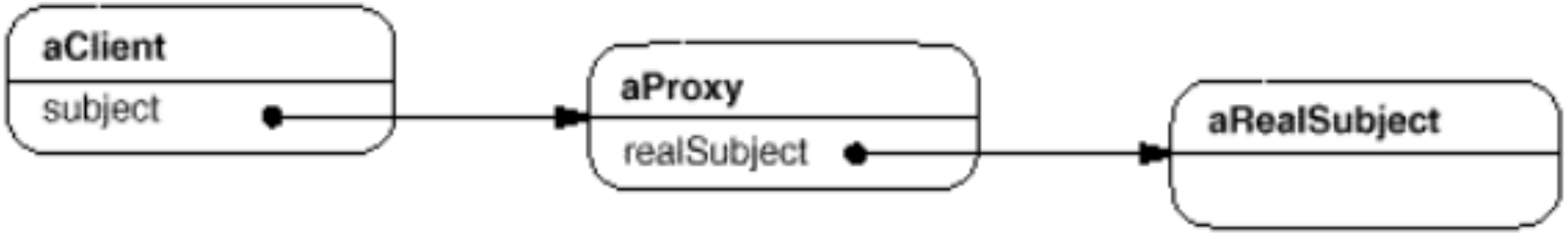
Proxy –Structure



Participants: Subject, RealSubject, Proxy. Client do not care the difference

Proxy structure

- A possible Object diagram of a proxy at runtime



Question: Can I have multiple proxies on top of each other?

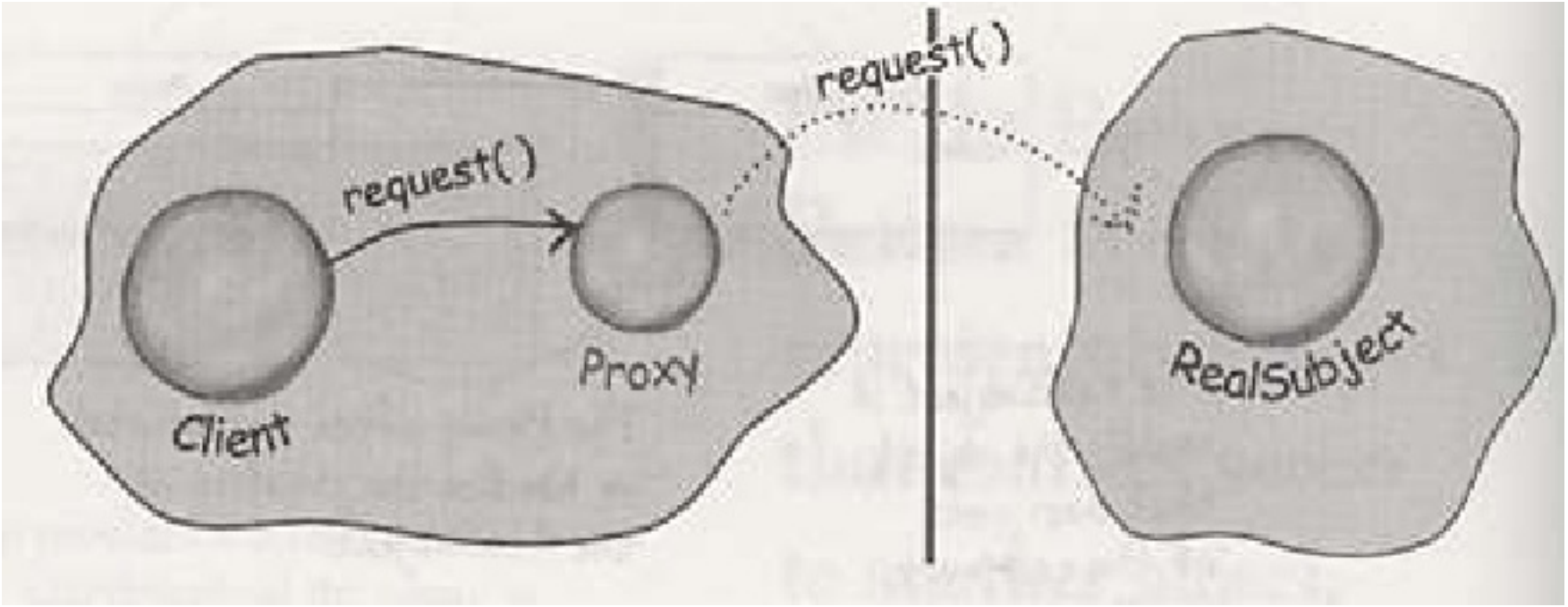
Remote proxy

- Example Scenario: A machine at the College of OOAD has several utility services running as daemons on well-known ports. We want to be able to access these services from various client machines as if they were local objects.

- Solution: Use a Remote Proxy!
  - Have a local placeholder for an object in a different address space
- This is the essence of modern distributed object technology such as REST, RPC like gRPC, event driven protocols like WebSockets

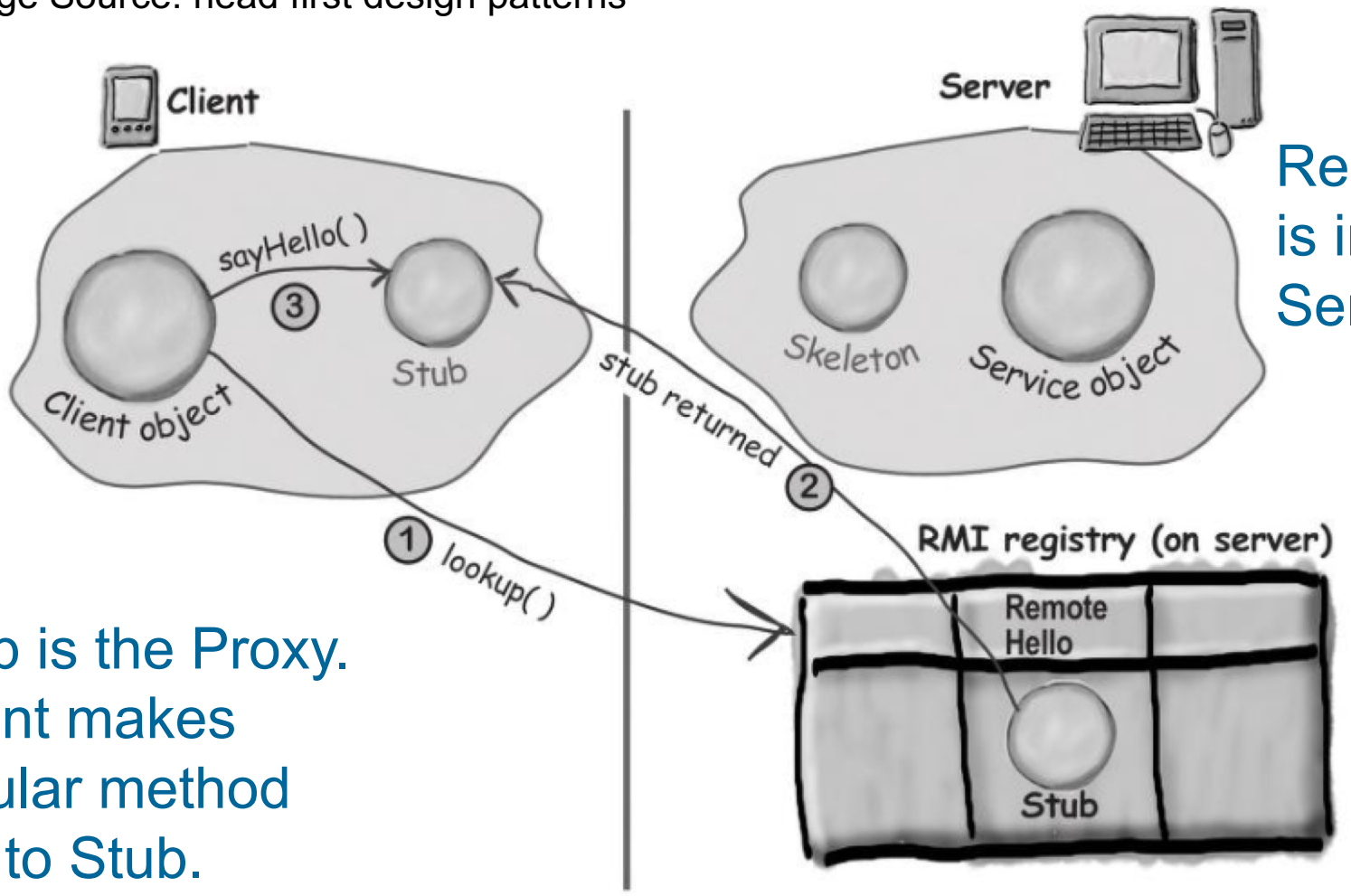
Remote proxy

- Client treats the proxy as local object
- Proxy actually deals with remote communication and access the remote object
  - stubs created in RMI compiler
  - commonly used in client-server applications



Source: head first design patterns

Image Source: head first design patterns



RealSubject is in the Server

Stub is the Proxy. Client makes regular method call to Stub.

Java RMI is not supported anymore.
But it helps explaining the concept
When using Rest API, create your own remote proxies
SOAP web service frameworks generate proxies

Server Side using WebService

```
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService
@SOAPBinding(style = Style.RPC)
public interface MyComplex {
   @WebMethod  ComplexData getComplexData();
}

@WebService(endpointInterface = "MyComplex")
public class MyComplexImpl implements MyComplex {..}

 public static void main(String[] args) {
       Endpoint.publish("http://localhost:9999/ws/mydata",
                        new MyComplexImpl());}
```

- **JAX-WS** automatically generates a WSDL document for your service.
- Client programmer downloads from http://localhost:9999/ws/mydata?wsdl

Client Side using Web Service

- Create an interface description using the WSDL file –`wsimport` comes with JDK
- `wsimport -keep -p com.example.client http://localhost:9999/ws/mydata?wsdl`
- It will generate necessary client files, which is depends on the provided WSDL file.
- In this example, it will generate one interface and one service implementation file.
  - MyComplex.java
  - MyComplexImplService.java

Source: https://mkyong.com/webservices/jax-ws/jax-ws-hello-world-example/

Client code using Remote proxy

```
import com.example.client.MyComplex;
import com.example.client.MyComplexImplService;
public class Client{
   public static void main(String[] args) {
        MyComplexService service = new MyComplexImplService();
        MyComplex proxy = service.getMyComplexImplPort();

        System.out.println(proxy.getComplexData());
        //regular method call.
    }
}
```

- When return/parameter type is custom object, create proxies from their wsdl files as well.

Another Web Service

- generate stubs. Remote proxy!
  - **wsdl2jws** –wsdl calculatorService.wsdl
- Client code

```
public class Main {
    public static void main(String[] args) {
        // Create a proxy for the calculator service
        CalculatorService calculatorService = new CalculatorService();
        ICalculator calculator = calculatorService.getICalculatorPort();

        // Call the "add" operation
        int result = calculator.add(10, 5);
        System.out.println("The result is: " + result);
    }
}
```

Virtual proxy

- Deferred object creation
- Client treats it as real object, but the real object is not created until it is needed.
  - don’t create until its methods are invoked or fields are accessed
  - after the creation, proxy is just a delegator
- Control access:
  - proxy controls access to the real subject so that before it is loaded proxy provides a default behavior – like “in progress”.
  - Once it is loaded proxy allows full access to it.

Virtual proxy: Example

- Application has an image object loading from a URL, but it takes too much time.
- Do not stall while loading the image, show something.
- Virtual proxy stands in place of the image
  - while loading the image in the background
    - in another thread
  - shows “loading image” message

```
// Subject
interface IImage {
    void modify();
    void save(String fileName);
}

// Real Subject
ImageImpl implements IImage{
   private BufferedImage img;
   public ImageImpl(String imageUrl){
          img = ImageIO.read(new URL(imageUrl));
    }
   public void save (String fileName){
          ImageIO.write(originalImage, "jpg", fileName);
   }
   public void modify(){.....}
}
```

```
ImageProxy implements IImage{
  private IImage realImage=null;
  private String name;    private Image dummy;
  public ImageProxy(String imgUrl){
        name=imgUrl;
        dummy=ImageIO.read(dummy.jpg); /*"loading image" msg*/}
  public void save(){
      if (realImage!=null) realImage.save() else load();}
  public void modify(){
      if (realImage!=null) realImage.modify() else load();}
  private void load(){ //load the real one in another thread
         Thread t=new Runnable(){
            public void run(){
              realImage=new ImageImpl(imageUrl);}};
          t.start();  //need synchronization, todo
   }
```

```
//client
public class ImageDisplayFX extends Application {
   String url=https://cs.someschool.edu/any.jpg

   public void start(Stage stage) throws IOException {
       IImage image = new ImageProxy(url);
       // would be better with dependency injection

       ImageView imageView = new ImageView(image);

       StackPane root = new StackPane(imageView);
       Scene scene = new Scene(root, image.getWidth(), image.getHeight());
       primaryStage.setTitle("Image Display (JavaFX)");
       primaryStage.setScene(scene);
       primaryStage.show();
    }
```

Realistic scenario…

- Image is built-in in Java; let's have a more realistic scenario
- "There is an ImageIcon that loads its image from a URL."
- Same solution
  - Load the inner image when the icon is painted with a worker thread

```
public class ImageProxy implements Icon{
  volatile ImageIcon imageIcon; //to protect reads in multithread
   final URL imageURL;Thread worker; boolean retrieving=false;
   public ImageProxy(URL url){imageURL=url;}
   public int getIconHeight(){
    if(imageIcon==null) return 800 //default until icon is loaded
    else return imageIcon.getIconHeight();
   }
```

Source: Head first design patterns

```
public void paintIcon(final Component c , Graphics g, int x, int y){
   if(imageIcon!=null) imageicon.paintIcon(g,g,x,y); //simple delegation
   else {// do not stall. Show something while loading
        g.drawString("Loading image....", x+300, y+200);
        if (!retrieving) {  //not to restart loading again and again
             retrieving=true;
             worker=new Thread(new Runnable(){
                  public void run{    //worker thread loads the image
                    try{
                      setImageIcon(new ImageIcon(imageURL, "MyIcon"));
                      c.paint(); //display when ready
                    }catch(Exception e) {e.printStackTrace();}}};
              worker.start();//worker setup done. Start loading
   }
   synchronized void setImageIcon(ImageIcon i) {imageIcon=I;}
}
```

Source: Head first design patterns

Exercise: Draw a diagram

- We need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely.
- Until that point, we can use some light objects exposing the same interface as the heavy objects.
- What are the participants?

Protection proxy

- like a secretary doesn't forward all the phone calls to a manager
- Example 1: only owner modifies its personal data in a chat application
  - if a non-owner access to your personal info than a proxy is returned
  - proxy allows get methods (query methods) but disallows set methods

Protection proxy

- like a secretary doesn't forward all the phone calls to a manager
- Example 2: Grade information for a student shared by administrators and the student. Teachers have access to the grades of students in their own courses, and only those grade in their courses.
  - Grade information need protection.
- Implementation issue:
  - A protection might refuse to perform an operation that the subject will perform
    - its interface may be a subset of the subject's.

Implementation Issues

- Proxy class can deal with its subject solely through an abstract interface
  - Not in virtual proxy. since it will instantiate RealSubjects, they have to know the concrete class.
- Overloading the member access operator in C++
  - overloading operator-> lets you perform additional work whenever an object is dereferenced.
  - the proxy behaves just like a pointer.

Example: Overloading -> in C++

```
class ImagePtr:Graphics {
    public:
        ImagePtr(const char* imageFile);
        virtual ~ImagePtr();
        virtual Image* operator->();
        virtual Image& operator*();
    private:
        Image* LoadImage();
    private:
        Image* _image;
        const char* _imageFile;
    };
    //impl in next slide
```

```
class Image:Graphics{
public: Image(const char* file);
 virtual void Draw(const Point& at);
 virtual void Load(istream& from);
 ...
}
```

```
//client
ImagePtr image =
ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
```

Example: overloading -> C++

```
ImagePtr::ImagePtr (
     const char* theImageFile) {
        _imageFile = theImageFile;
        _image = 0;
    }
Image* ImagePtr::LoadImage () {
    if (_image == 0) {
       _image =
          LoadAnImageFile(_imageFile);
     }
    return _image;
}
Image* ImagePtr::operator-> () {
        return LoadImage();}
Image& ImagePtr::operator* () {
        return *LoadImage();}
```

```
class Image:Graphics{
public: Image(const char* file);
 virtual void Draw(const Point& at);
 virtual void Load(istream& from);
 ...
}
//client
ImagePtr image =
ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
```

The image proxy acts like a pointer, but it's not declared to be a pointer to an Image.

Smart reference

- Surrogate of an object that performs additional actions when object is accessed
- Typical uses:
  - checking real object's locks before it is accessed
  - counting #of references for garbage collection
  - copy-on-write

Scenario 1: Problem

- Java has garbage collection
- C++ does not.
- When exiting a function, destructors of the objects are invoked.
- That is not enough…

- Memory leak

```
void myfun (){
  int * p=new int();
  *p=20;
  cout << *ptr;}
```

- Upon exiting myfun
  - p is not valid
    - local variable
  - Memory allocated is still there!
  - Need explicit delete

Scenario 1: Solution

- Let's solve this problem with a smart reference
  - SmartPtr

1. Wrap the int* in an object
2. Put delete ptr inside ~SmartPtr
3. myfun will have a SmartPtr instead of int*
4. The object destroys itself when it goes out of its scope
   - Deleting ptr for us

- Memory leak

```
void myfun (){
  int * p=new int();
  *p=20;
  cout << *ptr;}
```

- Upon exiting myfun
  - p is not valid
    - local variable
  - Memory allocated is still there!
  - Need explicit delete

source: https://www.geeksforgeeks.org/smart-pointers-cpp/

Scenario 1: Smart Ref Solution

```
class SmartPtr {
  int* ptr; // Actual pointer
public:
  explicit SmartPtr(int* p = NULL)
 { ptr = p; }

  ~SmartPtr() { delete (ptr); }

   // Overload dereferencing op.
    int& operator*() { return *ptr;
}
};
```

- Memory leak:

```
void myfun (){
   int * p=new int();
   *p=20;
   cout << *ptr;}
```

- No memory leak:

```
void fun(){
    SmartPtr ptr(new int());
    *ptr = 20;
    cout << *ptr;
}
```

source: https://www.geeksforgeeks.org/smart-pointers-cpp/

Scenario 1: Solution

```
class SmartPtr {
  int* ptr; // Actual pointer
public:
  explicit SmartPtr(int* p = NULL)
 { ptr = p; }

  ~SmartPtr() { delete (ptr); }

   // Overload dereferencing op.
    int& operator*() { return *ptr;
}
};
```

- For a more generic solution, use a template

Smart Pointers in C++

```
template <typename T>
class SmartPtr {
  T* ptr; // Actual pointer
public:
  explicit SmartPtr(T* p = NULL)
 { ptr = p; }

  ~SmartPtr() { delete (ptr); }

   // Overload dereferencing op.
    T& operator*() { return *ptr;
}
};
```

STL provides:

std::unique_ptr

std::shared_ptr

std::weak_ptr

std::auto_ptr

"Smart pointers enable automatic, exception-safe, object lifetime management."

https://en.cppreference.com/w/cpp/memory#Smart_pointers

STL support for Smartpointer

```
#include <memory>
class myclass{
private:
    std::unique_ptr<int[]> data;
public:
    myclass(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingMyclass() {
    myclass w(1000000);  // lifetime automatically tied to enclosing scope
                         // constructs w, including the member w.data
    w.do_something();
} // automatic destruction and deallocation for w and w.data
```

https://learn.microsoft.com/en-us/cpp/cpp/object-lifetime-and-resource-management-modern-cpp

- Smart pointers are a *smart reference*–style proxy that focuses on *lifetime*.
- Most structural proxies mediate access control, caching, logging, remoting..

Scenario2:Copy-on-write

- "Copy on write" : everyone has a single shared copy of the same data *until it's written*,
  - Make actual copying only if it is modified
  - When copying a large complicated object is costly
- Postpone the copying and pay the price only if the object is modified
- If the copy is never modified, then there's no need to pay this cost

Scenario2: solution outline

- 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.

Proxy - Consequences

- Introduces a level of indirection when accessing an object
  - Hide extra behavior
- Remote proxy: Hide that the real object resides in a different address space
- Perform optimizations,
  - Create object on demand
  - Caching results
- Smart references and protection proxies:
  - additional housekeeping when objects are accessed

Related patterns

- **Adapter**
  - Adapter provides a different interface to the object it adapts where a Proxy implements the same interface as its subject.
- **Decorator**
  - A decorator implementation can be the same as the proxy however a decorator adds responsibilities to an object while a proxy controls access to it.