AndyPatterns

Teaching 23 patterns in 3 days

I have been running a design patterns course for several years where I teach 23 design pattern patterns in 3 days. I cover all the GOF (Gang of Four) patterns plus more. Here are some tales from the front lines!

History

Having attended Melbourne Patterns Group meetings for quite a few years (what wonderful discussions we have there!), and having written up three design patterns of my own, I decided to teach a design patterns course and offer it commercially.

The course has been running for over four years now (now is 2007) with between six to twelve attendees. Often a company will send a couple of employees to the same course so there are groups of students who already know each other – which is often interesting as they sometimes have discussions in front of the whole group about how a particular pattern could be applied to the software they are both working on in their companies.

How I start the course

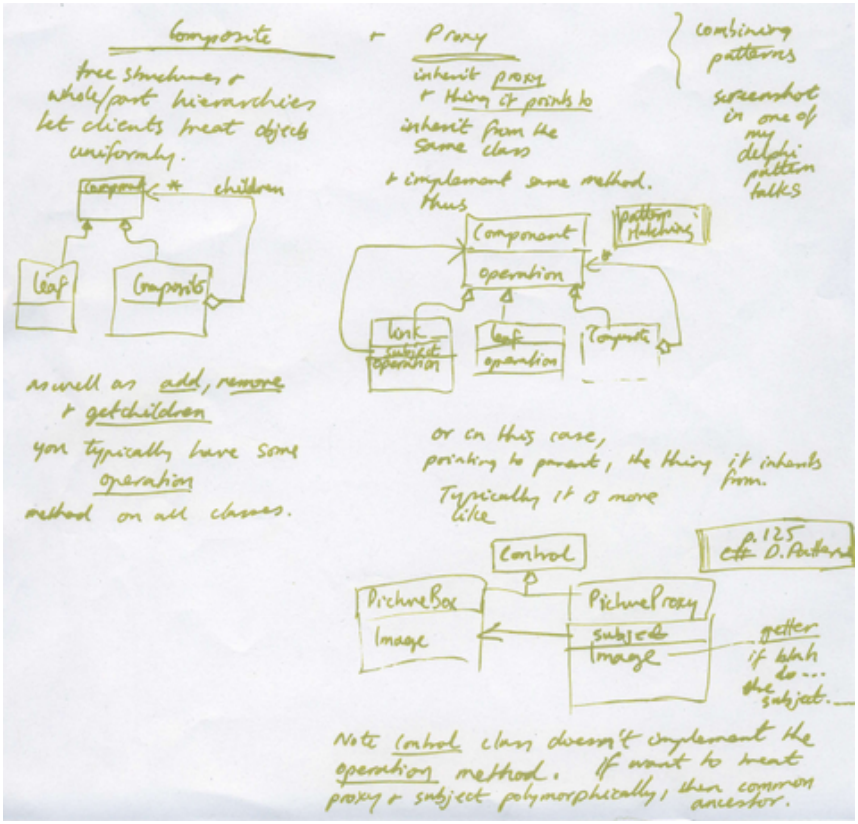
I begin with a PowerPoint presentation where I run through the basics of what patterns are and where they came from. I stress their benefit in communicating and transferring wisdom between programmers. I place a low emphasis on "pattern culture" and refrain from telling tales about Christopher Alexander (the architect who started the patterns movement in architecture) – it is more of a practical course.

I have found that it is critical to revise a bit of OO (Object Oriented Programming) at the beginning of the course, as most of the patterns rely on OO concepts. And it sort of warms everybody up and gets everybody on the same page in terms of concepts like interfaces, inheritance, composition, polymorphism etc. I also do a session on basic UML - as all the patterns I present are expressed in this notation.

History of the Course Materials

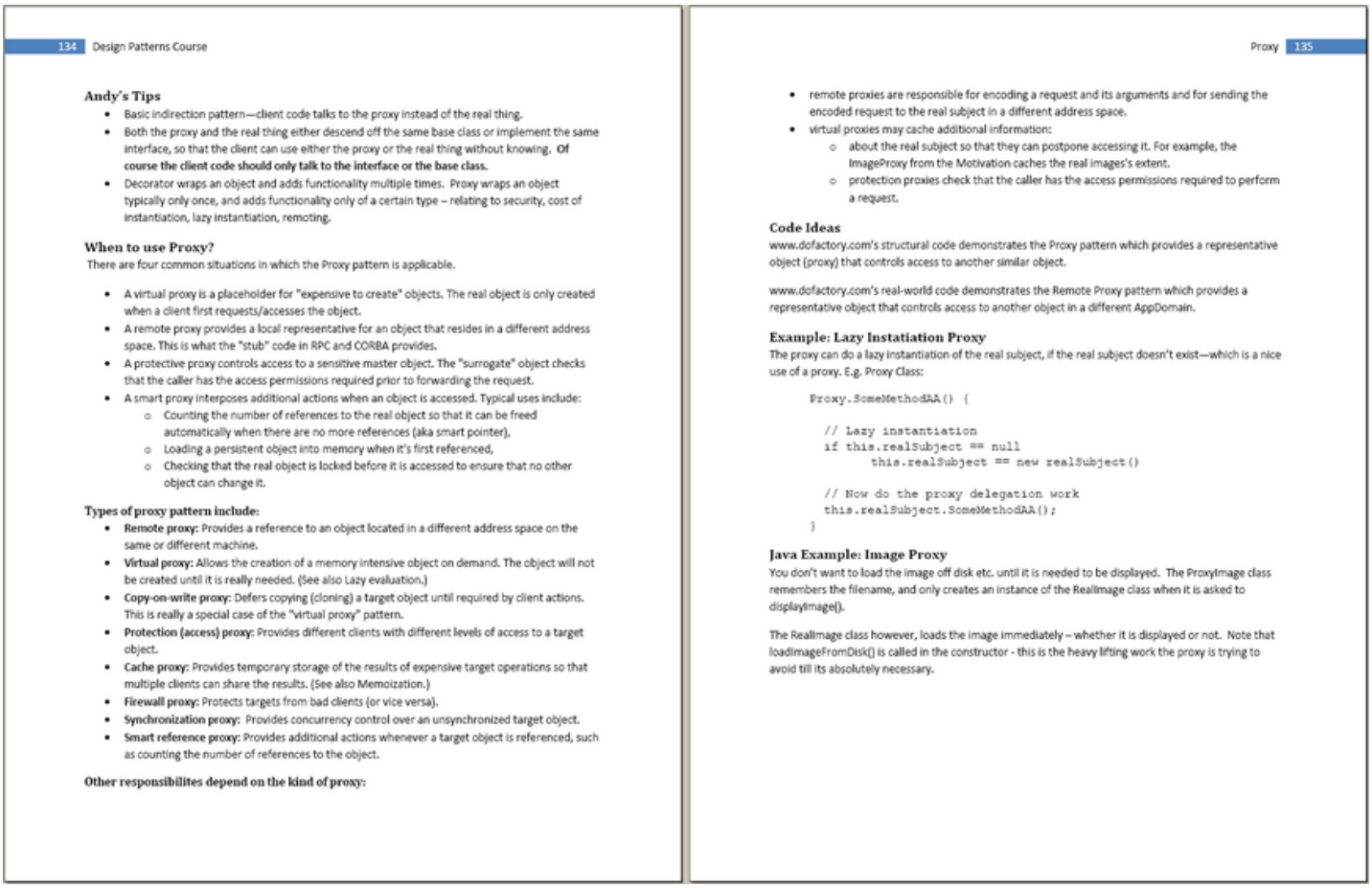
I provide a 300+ page course manual which I edit and improve before each course. During the course I make notes to myself relating to anything that is unclear, typos or any new ideas that come up during the course that I can incorporate back into the course manual.

Historically, writing a training course manual is a major part of getting a course going - it took me many months to write. I began with a handwritten brainstorming outline of all the GOF design patterns – here is an ancient extract from my original starting notes:



I then massaged everything into a proper Microsoft Publisher document, adding content, tips and code examples. I later had to shift to using Microsoft Word 2007 because I needed a dynamic table of contents facility which for some reason Publisher doesn't support.

Here is an example of what the course manual looks like now:



134 Design Patterns Course

Andy's Tips

- Basic indirection pattern—client code talks to the proxy instead of the real thing.
- Both the proxy and the real thing either descend off the same base class or implement the same interface, so that the client can use either the proxy or the real thing without knowing. **Of course the client code should only talk to the interface or the base class.**
- Decorator wraps an object and adds functionality multiple times. Proxy wraps an object typically only once, and adds functionality only of a certain type – relating to security, cost of instantiation, lazy instantiation, remoting.

When to use Proxy?

There are four common situations in which the Proxy pattern is applicable.

- A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
- A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
- A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
- A smart proxy interposes 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 (aka smart pointer),
  - Loading a persistent object into memory when it's first referenced,
  - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Types of proxy pattern include:

- **Remote proxy:** Provides a reference to an object located in a different address space on the same or different machine.
- **Virtual proxy:** Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed. (See also Lazy evaluation.)
- **Copy-on-write proxy:** Defers copying (cloning) a target object until required by client actions. This is really a special case of the "virtual proxy" pattern.
- **Protection (access) proxy:** Provides different clients with different levels of access to a target object.
- **Cache proxy:** Provides temporary storage of the results of expensive target operations so that multiple clients can share the results. (See also Memoization.)
- **Firewall proxy:** Protects targets from bad clients (or vice versa).
- **Synchronization proxy:** Provides concurrency control over an unsynchronized target object.
- **Smart reference proxy:** Provides additional actions whenever a target object is referenced, such as counting the number of references to the object.

Other responsibilites depend on the kind of proxy:

Proxy 135

- 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 images's extent.
  - protection proxies check that the caller has the access permissions required to perform a request.

Code Ideas

www.dofactory.com's structural code demonstrates the Proxy pattern which provides a representative object (proxy) that controls access to another similar object.

www.dofactory.com's real-world code demonstrates the Remote Proxy pattern which provides a representative object that controls access to another object in a different AppDomain.

Example: Lazy Instatiation Proxy

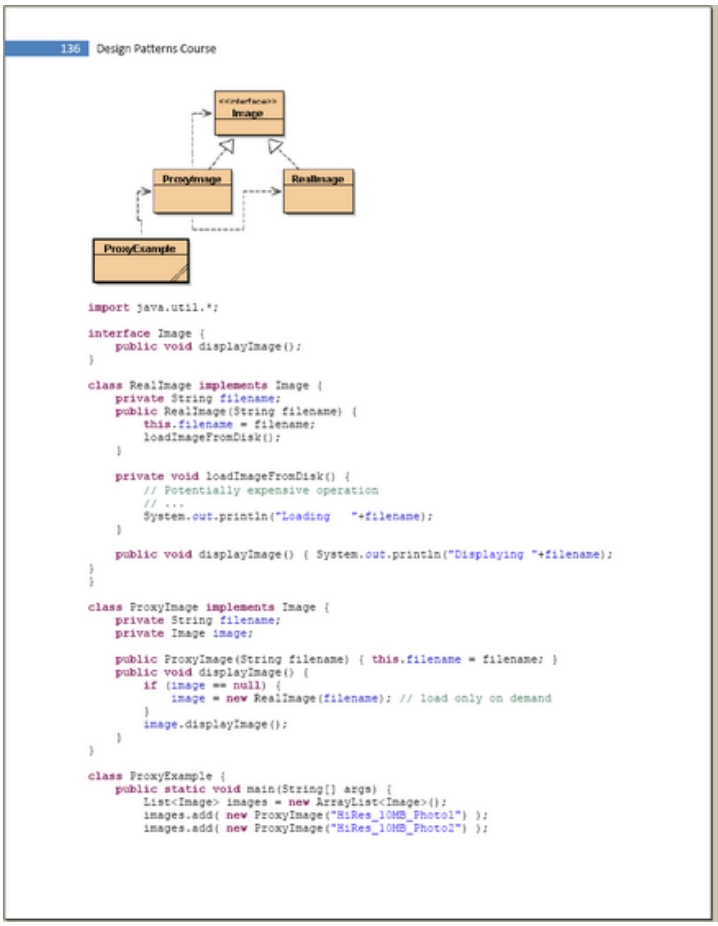
The proxy can do a lazy instantiation of the real subject, if the real subject doesn't exist—which is a nice use of a proxy. E.g. Proxy Class:

```
Proxy.SomeMethodAA() {

  // Lazy instantiation
  if this.realSubject == null
        this.realSubject == new realSubject()

  // Now do the proxy delegation work
  this.realSubject.SomeMethodAA();
}
```

Java Example: Image Proxy

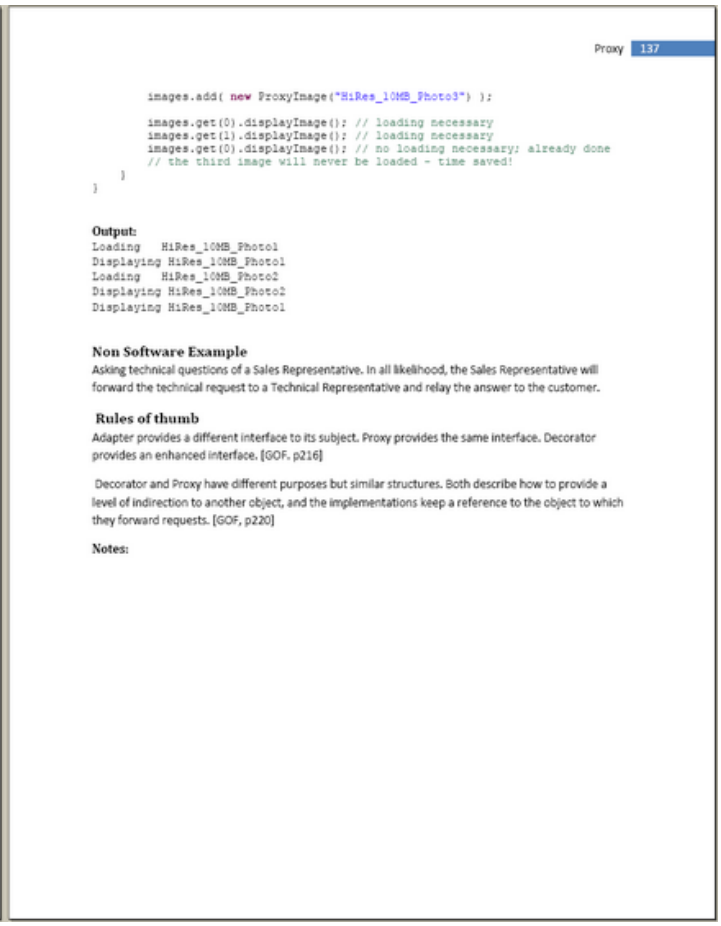
You don't want to load the image off disk etc. until it is needed to be displayed. The ProxyImage class remembers the filename, and only creates an instance of the RealImage class when it is asked to displayImage().

The RealImage class however, loads the image immediately – whether it is displayed or not. Note that loadImageFromDisk() is called in the constructor - this is the heavy lifting work the proxy is trying to avoid till its absolutely necessary.



136 Design Patterns Course


```
import java.util.*;

interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        // Potentially expensive operation
        // ...
        System.out.println("Loading   "+filename);
    }

    public void displayImage() { System.out.println("Displaying "+filename);
}
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        List<Image> images = new ArrayList<Image>();
        images.add( new ProxyImage("HiRes_10MB_Photo1") );
        images.add( new ProxyImage("HiRes_10MB_Photo2") );
```



Proxy 137

```
        images.add( new ProxyImage("HiRes_10MB_Photo3") );

        images.get(0).displayImage(); // loading necessary
        images.get(1).displayImage(); // loading necessary
        images.get(0).displayImage(); // no loading necessary; already done
        // the third image will never be loaded - time saved!
    }
}
```

Output:

```
Loading   HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading   HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo1
```

Non Software Example

Asking technical questions of a Sales Representative. In all likelihood, the Sales Representative will forward the technical request to a Technical Representative and relay the answer to the customer.

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF, p216]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Notes:

How I present

For each design pattern, I get each student to read the first page of the pattern from the course notes. This way each student 'owns' a pattern or two (we usually cycle around the students at least twice to get through the 23 patterns). It also engages the students a bit more keeping them awake (and possible terrified or excited). I follow that by a whiteboard presentation followed by my leaping onto a computer and building a code example from scratch.

I used to select patterns at random, depending on my mood and on what I thought the students would relate to best next. For example some patterns flow on from other patterns e.g. you need to understand the strategy design pattern early, as a lot of other patterns use this technique. The feedback I got sometimes expressed the preference for a fixed order of presentation so that it matched the order of the printed course materials. So from then on I changed the way I did things. I settled on a teaching order based on the reasoning of Joshua Kerievsky in his A Learning Guide To Design Patterns – which meant rearranging the teaching manual etc. I got no more complaints then and quite possibly the course is more effective because of the way the patterns build upon each other.

Code Examples

I quite often start building the examples from a UML diagramming tool like BlueJ in order to create the classes – then I fill out the code. This way we keep our attention on the high level design pattern usage. My code examples are in Java, C# and occasionally in Python – in my training courses you never know what the preferred development language of the students is – so my examples are evenly mixed between C# and Java with the occasional Python example thrown in to add something new into to the mix.

When building a code example – I like to do it from scratch – as I believe that pre-canned code examples are murky and mysterious – better to have everyone understand and see everything – and that means building examples from the beginning. I used to pride myself on "jamming" (like jazz improvisation) and used to code live in front of the class - but due to the stress and amount of code involved, especially in some of the more complex examples, I now copy and paste code from the student notes using a keyboard macro facility which types the code slowly – as if I am typing it. This alleviates the stress, allows me to commentate and doesn't bore the audience with my typos.

I always build examples that compile and run – either achieving some result or passing some set of unit tests. Sometimes I adapt the examples as needed with variable names and method names invented by the group, so that the code feels more personalized and meaningful, and sometimes also we jam with the code a little to experiment with variations, in order to

learn more. At the end of the training course I provide all the code we develop to anyone who wants it - on USB stick or via email.

Puzzles

Sometimes I preceded the presentation of a pattern with a puzzle and ask student to sketch out a solution to a problem themselves which gives the students some personal time to think and design. And when I present the elegant solution using the relevant design pattern, the student appreciates it better because he or she has struggled with the problem themselves. Note that I don't actually have students using laptops as this is a recipe for students becoming distracted – it's all done on paper. If I ever get to expand (or rather get paid for a presenting) my 3 day course to 4 days, then there will be more time for these useful puzzle sessions.

From a teaching point of view

Teaching these courses requires a lot of energy as we move at a rate of seven patterns a day. You have to engage the room and make it entertaining - filled with insight and technical detail. I'm not sure about the students, but after the course finishes on the 3rd night I usually collapse into a deep coma sleep well into the next morning. It's a lot for students to absorb too, though most good programmers handle the amount of information presented well and come away excited.

As programmers are a highly intelligent bunch, I constantly have to handle probing questions – which I try to answer as honestly as I can. I've tried to become more precise over the years – answering the exact question rather than converting it into yet another convenient question which helps in the learning process. During some earlier courses I felt I was answering questions like a politician, especially if I felt the questions were half-baked and that our time was better spent answering a more profound question. But I didn't like the feeling of being a politician so I try to answer every question as it stands. There is also a delicate balance between being an authority and my not knowing. Whilst there are situations (especially in the wide and wonderful world of design patterns and OO design) where there is no clear answer to a question, too much uncertainty is not good for a training course where certainty is expected.

I encourage a certain amount of group discussion. Its powerful letting students think and debate for themselves. I put clues and questions their way during the discussion in order to facilitate the learning outcomes. You have to not let it go on too long and stay on schedule.
Students have different personalities - there are active students who get right into it and then you get the quiet ones who just listen most of the time (until they have to read their pattern!). And funnily enough the cross-section of the groups always seem to form a similar pattern: usually two or three positive and really bright sparks, one or two slower students, always one slightly grumpy negative student, and the rest are just normal.

Future Directions

Last time I wrote a "future directions" section (in an earlier version of this blog) I promised to present patterns in a known order - exactly matching the training manual – this I now do and it has been a worthwhile change. I also promised that I would add more code samples – and this is now the case – every pattern has at least one code example.

In terms of the future, I would love to actually deliver more of these courses, and would like to try a 4 day course so that we get more time for puzzles etc. and spend more time exploring UML tools that support design pattern automation etc. Most employers don't like to lose their programmer for so long - even three days is pretty long, so a shorter course – a 2 day course – is something I am also offering now. I've done one day course before too, but in the end my favorite course is the classic 3 day course where we get to cover all the basic 23 GOF design patterns.

-Andy

For details on my course click here.