An Introduction to Design Patterns

William H. Mitchell (whm)
Mitchell Software Engineering (.com)

What is a "pattern"?

The first among a dozen definitions in Webster's Ninth New Collegiate is this:

> *A form or model proposed for imitation*

As currently used in the software industry:

> *A pattern is a description of a common problem and a likely solution, based on experience with similar situations.*
>
> —whm

A little history

Christopher Alexander, a bricks-and-mortar architect, proposed in the 1970s that visually pleasing and practical structures for a certain application and/or setting could be described by a *pattern language*.

For example, Alexander wrote that desirable farmhouses in the Bernese Oberland area of Switzerland used these patterns:

North South Axis
West Facing Entrance Down The Slope
Two Floors
Hay Loft At The Back
Bedrooms In Front
Garden To The South
Pitched Roof
Half-Hipped End
Balcony Toward The Garden
Carved Ornaments

Alexander claimed that following these patterns when designing a farmhouse produces a structure that blends with the environment and the community, and has the "Quality Without A Name".

Alexander further claimed that a pattern language permits a great variety of buildings to be designed that meet a particular need.

A little history, continued

In 1987, Kent Beck and Ward Cunningham presented an OOPSLA paper titled *Using Pattern Languages for Object-Oriented Programs*.

It described a project that applied Alexander's work to the problem of user-interface design in which the eventual users of a system designed the interface guided by these patterns:

> Window Per Task
> Few Panes Per Window
> Standard Panes
> Short Menus
> Nouns and Verbs

In the late 1980s and early 1990s a number of individuals began to look at the problem of identifying and describing patterns used to create software.

In 1995 the now-classic text *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides was published.

Design Patterns basically focuses on one issue:

> Devising a set of objects and orchestrating an interaction between them to perform a computation can be a non-trivial problem.

Design Patterns is essentially a catalog of 23 commonly occurring problems in object-oriented design and a pattern to solve each one.

The authors are often called the Gang of Four (GoF).

An object-oriented design problem

Imagine a system that uses a number of temperature sensors to monitor the condition of a hardware device of some sort.

The first model of the device uses TempTek, Inc. TS7000 sensors.

TempTek supplies a simple Java class to interface with the sensors:

```
class TS7000 {
   native double getTemp();
   ...
   }
```

Here is some monitoring code that simply calculates the mean temperature reported by the sensors.

```
double sum = 0.0;
for (int i = 0; i < sensors.length; i++)
   sum += sensors[i].getTemp();

double meanTemp = sum / sensors.length;
```

Note that sensors is declared as an array of TS7000 objects. (TS7000 sensors[ ] = new TS7000[...])

Any problems here?

An OO design problem, continued

The second model of the device uses more temperature sensors and the design uses a mix of TS7000s and sensors from a new vendor, Thermon.

The Thermon sensors are SuperTemps and a hardware interfacing class is supplied:

```
class SuperTempReader {
   //
   // NOTE: temperature is Celsius tenths of a degree
   //
   native double current_reading();
   ...
   }
```

Here is a terrible way to accommodate both types of sensors:

```
for (int i = 0; i < sensors.length; i++)
{
   if (sensors[i] instanceof TS7000)
      sum += ((TS7000)sensors[i]).getTemp();
   else
      // Must be a SuperTemp!
     sum +=
         ((SuperTempReader)sensors[i]).current_reading() * 10;
}
```

In this case sensors is an array of Objects. The type is tested with instanceof and an appropriate cast and method call is performed.

What's terrible about this code?

A pattern to the rescue!

Problems arose when a component from a second vendor was introduced. More vendors may be involved in the future.

We have no control over the name of the temperature-reporting method in the vendor-supplied classes and additionally, the value produced may need scaling, unit conversion, etc.

All that we can really expect is that a temperature can be determined for each sensor location.

The ADAPTER design pattern provides a solution to this problem.

What is a Design Pattern?

A Design Pattern is essentially a description of a commonly occurring object-oriented design problem and how to solve it.

Part of the idea of Design Patterns is that patterns have a certain literary form. In the GoF book, patterns typically have these (major) elements:

Intent
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Implementation
Sample Code
Known Uses
Related Patterns

Part of the benefit of patterns rises from the discipline of having to verbalize these various aspects.

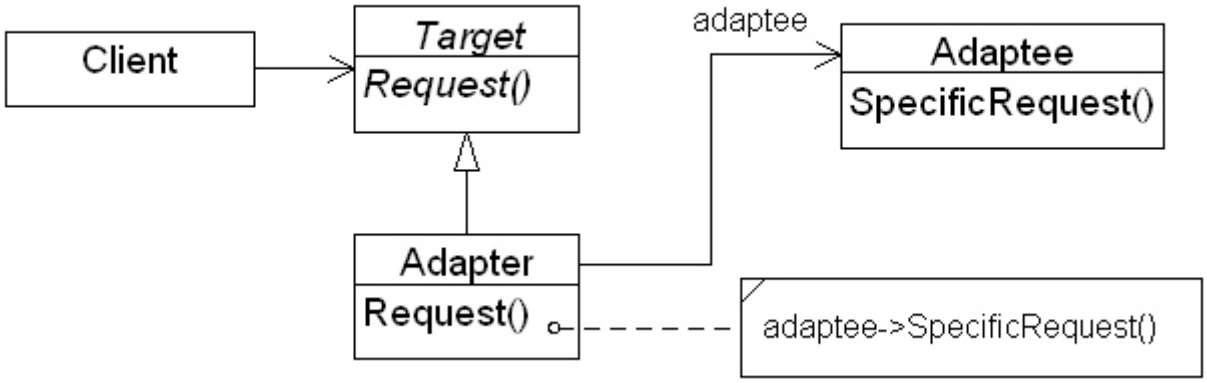
The specification of ADAPTER in the GoF book covers about twelve pages.

The ADAPTER Pattern

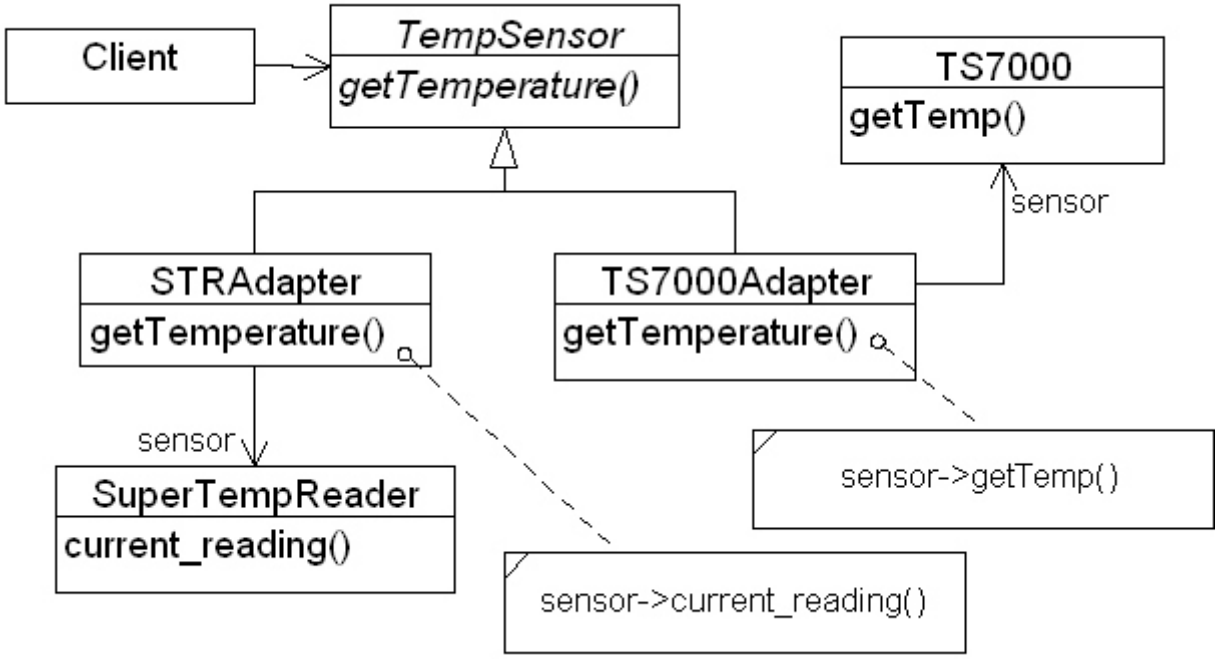
For the *Intent* of ADAPTER, the GoF cites this:

> *Convert the interface of a class into another interface clients expect. ADAPTER lets classes work together that couldn't otherwise because of incompatible interfaces.*

Here is the *Structure* of ADAPTER:



Here is ADAPTER applied to the problem at hand:



ADAPTER, continued

Here's the code for the new classes:

```
abstract class TempSensor
{
   abstract double getTemperature();
}

class STRAdapter extends TempSensor
{
   public double getTemperature()
   {
      return sensor.current_reading() * 10;
   }
   ... constructor, etc. ...
}

class TS7000Adapter extends TempSensor
{
   public double getTemperature()
   {
      return sensor.getTemp();
   }
   ... constructor, etc. ...
}
```

Here's the new version of the mean temperature calculation:

```
double sum = 0.0;
for (int i = 0; i < sensors.length; i++)
   sum += sensors[i].getTemperature();
```

In this case sensors is an array of TempSensors that contains a mix of TS7000Adapters and STRAdapters, which in turn reference instances of TS7000 and SuperTempReader.

ADAPTER—results

For reference, *Intent*:

> *Convert the interface of a class into another interface clients expect. ADAPTER lets classes work together that couldn't otherwise because of incompatible interfaces.*

Results of applying the ADAPTER pattern:

> The mean-calculating code can be written in terms of TempSensor operations; no vendor-specific code remains.
>
> Sensors from additional vendors can be accommodated by simply creating additional subclasses of TempSensor.
>
> In other words, the mean-calculating code satisfies Meyer's Open-Closed Principle.

Design Patterns—What's the benefit?

The problem, and the solution provided by ADAPTER, are not rocket science.

A developer with training in object-oriented design would probably produce a similar solution without much effort.

What's the real contribution of design patterns?

> Simply having a name for a solution provides a real benefit: A developer might say
>
> > *"Let's use an ADAPTER."*
>
> rather than
>
> > "*How about a superclass with a getTemperature method and then subclass that with classes that reference instances of the vendor-supplied classes for....*"
>
> Developers can have some confidence that the solution chosen is not entirely off the wall and has been used with success in similar situations in other systems.
>
> The solution of ADAPTER is not hard to reach independently but it is a relatively simple pattern. For more complex patterns it's useful to have a catalog that can be consulted.

A quick look at some other patterns

Here are some of the other GoF patterns, and their Intent:

COMPOSITE:

> Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

OBSERVER:

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

FLYWEIGHT:

> Use sharing to support large numbers of fine-grained objects efficiently.

MEMENTO:

> Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

ITERATOR:

> Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

FACADE:

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Are patterns really something new?

Betrand Meyer wrote that new ideas are often met with a sequence of three reactions:

1. There's nothing to it.

2. It'll never work.

3. That's how we've always done it.

Therefore...

1. Is the notion of a design pattern a significant idea?

2. Do they really work?

3. What's new about describing solutions to problems?

Lots of "patterns"

For a while "pattern" was essentially synonymous with "object oriented design pattern" but the meaning has broadened.

In the literature you can now find "patterns" for:

High-level application architecture
User interfaces
Software testing
Project organization and management
Web sites
And more...

There are also *AntiPatterns*, which "describe a solution to a problem that generates decidedly negative consequences."

Recommended reading

Books:

Design Patterns by Erich Gamma et al. 1995. Published by Addison-Wesley. ISBN 0-201-63361-2

Agile Software Development—Principles, Patterns, and Practices, by Robert C. Martin. 2003. Published by Pearson Education, Inc. ISBN 0-13-597444-5

Applying UML and Patterns, by Craig Larman. 2002. Published by Prentice Hall PTR. ISBN 0-13-092529-1

Web sites:

www.hillside.net/patterns

www.cmcrossroads.com/bradapp/docs/patterns-intro.html

Deep Background:

The Timeless Way of Building and *A Pattern Language* by Christopher Alexander