Resistance is Futile – How to Make Your Java Objects Conform with the Adapter Pattern

by Barry A. Burd and Michael P. Redlich

This article, the fourth in a series on design patterns, introduces the Adapter pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. (The book's nickname is the *Gang of Four* (*GoF*) book, because of its four authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.)

Design Patterns

The GoF book's 23 design patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Adapter design pattern fits into the structural category. Like the Decorator design pattern, the Adapter pattern is also known as a "wrapper." The Adapter and Decorator design patterns are similar, but they perform slightly different roles.

- A decorator adds responsibilities to an object.
- An adapter connects existing interfaces.

The Adapter Pattern

Here's a quotation from the GoF book:

"[An adapter] converts the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."

The Adapter design pattern follows two important design principles (both quoted directly from the GoF book):

- "Favor object composition over class inheritance."
- "Program to interfaces, not implementations."

Motivation

Consider the following situation. You have a nice, useful piece of code, but you can't easily use the code in an existing application. The "useful" code has an interface, but the existing application expects a different interface. Using the Adapter design pattern, you can use the useful code in the existing application.

Implementation

Figure 1 shows the official UML diagram for the Adapter pattern.

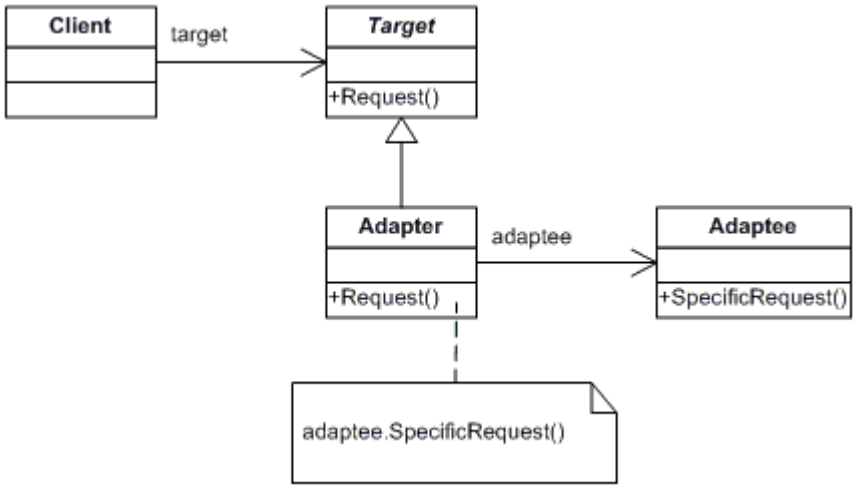

Figure 1: The official UML diagram for the Adapter design pattern

Here's a walk-through of the parts of the UML diagram.

- A **`Client`** class expects a certain interface (called the **`Target`** interface).
- An available interface doesn't match the **`Target`** interface.
- An **`Adapter`** class bridges the gap between the **`Target`** interface and the available interface.
- The available interface is called the **`Adaptee`**.
- The **`Adapter`** class stores a copy of **`adaptee`**, an instance of the **`Adaptee`** class.

What this UML diagram needs is a concrete example. Consider some electronic devices -- Game Boys, cell phones, and other devices. You plug the devices into your car's cigarette lighter socket or the wall outlet in your house. Both car charging and home charging require adapter cables. The UML diagram for this concrete example is in Figure 2.

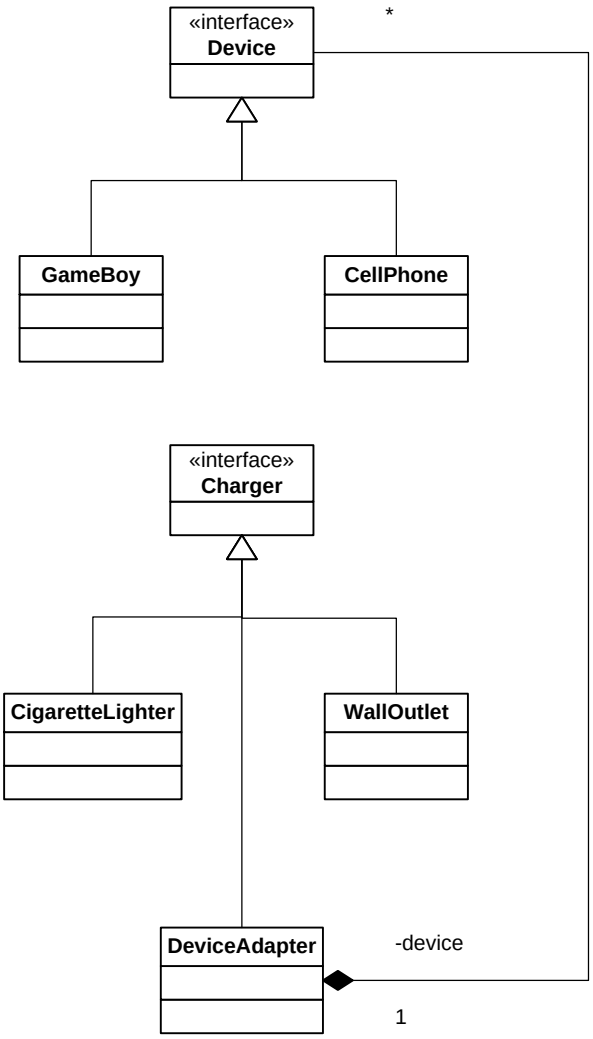

Figure 2: The UML diagram for the device charging application

The elements in Figure 2 have parallels in Figure 1.

- A Game Boy or cell phone device (the **`Client`** in Figure 1) expects a certain interface. This interface may be a mini-plug, a USB port, or some other gizmo. The general name for such an interface is the **`Target`** interface.
- An available **`Charger`** interface (a power source, such as a wall outlet or cigarette lighter socket) doesn't match the **`Target`** interface.
- A **`DeviceAdapter`** (called an **`Adapter`** in Figure 1) bridges the gap between the **`Target`** interface and the **`Charger`** interface.
- The **`Charger`** interface is called the **`Adaptee`**.

Figure 3 shows yet another view of all this terminology.

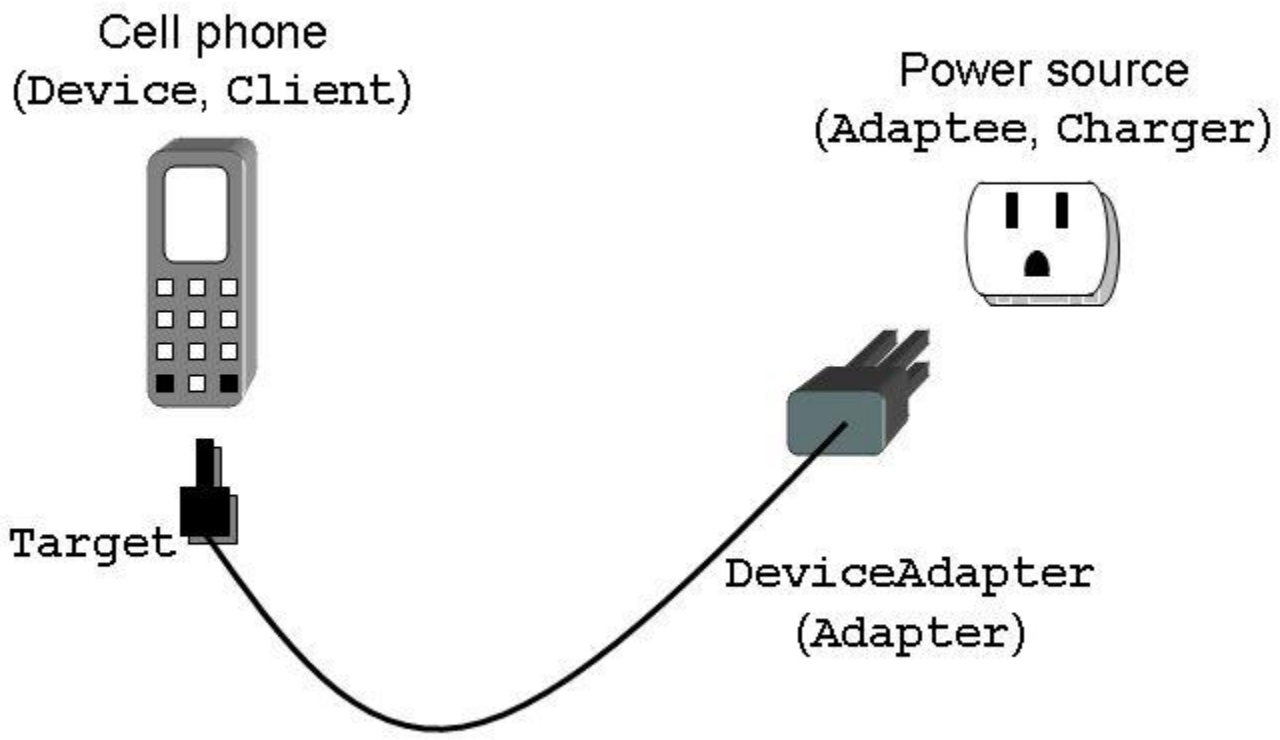

Figure 3: Visualizing the device charging application

The source code for the UML diagram in Figure 2 is in Listings 1 to 7. A main method to test the code is in Listing 8.

```
public interface Device
	{
	public void getCharge(String charger);
	}
```

Listing 1: The Device interface

```
public class GameBoy implements Device
	{
	public void getCharge(String charger)
		{
		System.out.println("The Game Boy is being charged by the " + charger + "...");
		}
	}
```

Listing 2. The GameBoy class (a client)

```
public class CellPhone implements Device
	{
	public void getCharge(String charger)
		{
		System.out.println("The Cell Phone is being charged by the " + charger + "...");
		}
	}
```

Listing 3. The CellPhone class (another client)

```
public interface Charger
	{
	public void chargeDevice(String charger);
	public String getCharger();
	}
```

Listing 4. The Charger interface

```
public class CigaretteLighter implements Charger
	{
	String charger;

	public CigaretteLighter(String charger)
		{
		setCharger(charger);
		}

	public void chargeDevice(String charger)
		{
		System.out.println("The " + charger + " is ready");
		}

	public String getCharger()
		{
		return charger;
		}

	public void setCharger(String charger)
		{
		this.charger = charger;
		}
	}
```

Listing 5. The CigaretteLighter class (an adaptee)

```
public class WallOutlet implements Charger
	{
	String charger;

	public WallOutlet(String charger)
		{
		setCharger(charger);
		}

	public void chargeDevice(String charger)
		{
		System.out.println("The " + charger + " is ready");
		}

	public String getCharger()
		{
		return charger;
		}

	public void setCharger(String charger)
		{
		this.charger = charger;
		}
	}
```

Listing 6: The WallOutlet class (another adaptee)

```
public class DeviceAdapter implements Charger
	{
	Device device;

	public DeviceAdapter(Device device)
		{
		this.device = device;
		}

	public void chargeDevice(String charger)
		{
		device.getCharge(charger);
		}

	public String getCharger()
		{
		return "charger";
		}
	}
```

Listing 7: The DeviceAdapter class (the adapter)

```
public class ChargeDevices
	{
	public static void main(String[] args)
		{
		System.out.println("-- Device Charging Application --");
		System.out.println();

		// create the chargers
		Charger lighter = new CigaretteLighter("cigarette lighter");
		Charger outlet = new WallOutlet("wall outlet");

		// create the devices
		Device gameBoy = new GameBoy();
		Device cellPhone = new CellPhone();

		// create the device adapters
		Charger device01 = new DeviceAdapter(gameBoy);
		Charger device02 = new DeviceAdapter(cellPhone);

		String charger01 = lighter.getCharger();
		String charger02 = outlet.getCharger();

		// charge the devices
		device01.chargeDevice(charger01);
		device02.chargeDevice(charger01);
		device01.chargeDevice(charger02);
		}
	}
```

Listing 8: The ChargingDevices class – a main method for informal testing

The **`Device`** interface declares a method called **`getCharge()`** which is the interface that clients (cell phones and Game Boys) use. In the meantime, the **`Charger`** interface declares a different interface method -- a method named **`chargeDevice()`**.

The main application (Listing 8) declares **`Charger`** instances (**`CigaretteLighter`** and **`WallOutlet`**) and **`Device`** instances (**`GameBoy`** and **`CellPhone`**). An adapter for each device allows a charger to charge the device. That's where the fun begins. The main method creates an instance of **`DeviceAdapter`** for each of the devices:

```
// create the device adapters
Charger device01 = new DeviceAdapter(gameBoy);
Charger device02 = new DeviceAdapter(cellPhone);
```

Notice two things about the code in **DeviceAdapter** (Listing 7):

- The code implements the **Charger** interface, so this code must implement the **Charger** interface's **chargeDevice()** method.
- The code's **device** field stores an instance of a **Device** type.

The **DeviceAdapter** uses its stored **device** object to implement the **chargeDevice()** method.

```
public void chargeDevice(String charger)
      {
      device.getCharge(charger);
      }
```

Each **DeviceAdapter** instance has a **Device** instance. Another name for this "has a" relationship is "composition." And the use of composition underlies many design patterns.

Figure 4 displays the output of the **ChargeDevices** program (Listing 8).

```
-- Device Charging Application --

The Game Boy is being charged by the cigarette lighter...
The Cell Phone is being charged by the cigarette lighter...
The Game Boy is being charged by the wall outlet...
```

Figure 4: The output of the device charging application

Adapting in the Java API

Java has two interfaces for traversing a container -- the older **Enumeration** interface, and the newer **Iterator** interface. In the book *Head First Design Patterns*, written by Eric & Elisabeth Freeman (along with Kathy Sierra and Bert Bates), the authors challenge you to write the following code:

- Adapt an iterator to an enumeration.
- Write a main method that uses this new adapted enumeration to traverse an **ArrayList**. (Remember, the **ArrayList** class doesn't support enumerations.)

Listings 9 and 10 contain a version of the iterator adapter and the main method:

```
import java.util.Enumeration;
import java.util.Iterator;

public class IteratorAdapter implements Enumeration
      {
      Iterator iterator;

      public IteratorAdapter(Iterator iterator)
            {
            this.iterator = iterator;
            }
```

```
	public boolean hasMoreElements()
		{
		return iterator.hasNext();
		}

	public Object nextElement()
		{
		return iterator.next();
		}
	}
```

Listing 9. The IteratorAdapter class

```
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;

public class IteratorAdapterTest
	{
	public static void main(String[] args)
		{
		System.out.println("-- Iterator Adapter Test Application --");
		System.out.println();
		System.out.println("Enumerating Through the JavaBoutique Articles");
		System.out.println("Written by Barry Burd and Michael Redlich");
		System.out.println();

		List<String> list = new ArrayList<String>();

		list.add("James: The Java Apache Mail Enterprise Server, " +
			"September 30, 2005");
		list.add("Avoid Excessive Subclassing with the " +
			"Decorator Design Pattern, " +
			"January 27, 2006");
		list.add("Keeping Your Java Objects Informed with the " +
			"Observer Design Pattern, " +
			"June 19, 2006");
		list.add("Manufacturing Java Objects with the " +
			"Factory Design Pattern, " +
			"August 14, 2006");

		Iterator iterator = list.iterator();
		Enumeration enumeration = new IteratorAdapter(iterator);

		int i = 1;
		while(enumeration.hasMoreElements())
			{
			System.out.println(i + ": " + enumeration.nextElement());
			++i;
			}
		}
	}
```

Listing 10: The IteratorAdapterTest class – the client application

Like the **`DeviceAdapter`**, the **`IteratorAdapter`** implements the adaptee's interface (the **`Enumeration`** interface). The **`IteratorAdapter`** stores an instance of **`Iterator`**, and uses this instance to call the **`hasNext()`** method in the implementation of **`hasMoreElements()`**. The main method (Listing 10) creates an **`ArrayList`** to

hold some strings. The **ArrayList** class doesn't support the **Enumeration** interface. Even so, Listing 10 uses an enumeration to traverse an **ArrayList**.

Figure 5 displays the output of the iterator adapter test application.

```
-- Iterator Adapter Test Application --

Enumerating Through the JavaBoutique Articles
Written by Barry Burd and Michael Redlich

1: James: The Java Apache Mail Enterprise Server, September 30, 2005
2: Avoid Excessive Subclassing with the Decorator Design Pattern, January 27, 2006
3: Keeping Your Java Objects Informed with the Observer Design Pattern, June 19, 2006
4: Manufacturing Java Objects with the Factory Design Pattern, August 14, 2006
```

Figure 5: The output of the iterator adapter test

Object vs. Class Adapters

There are actually two ways to implement the Adapter design pattern. The examples in this article demonstrate the use of the Adapter design pattern as an ***object*** adapter. The object adapter uses composition to accomplish its goal.

The ***class*** adapter is similar. But the class adapter uses inheritance (more specifically, multiple inheritance) instead of composition. Java doesn't support multiple inheritance, so you can't implement a class adapter in Java. (To implement a class adapter, you must dirty your hands with C++.)

Resistance is Futile

For the Borg on Star Trek, and for application software, assimilation is crucial. The adapting of one system to another system's interface is an important tool in the development process. If the Borg's computers run Java (and we're certain that they do) then the computers connect with other species using the Adapter design pattern. The pattern provides an elegant, orderly procedure for making one system's interface match another system's expectations.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2
http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1

Head First Design Patterns
Eric & Elisabeth Freeman
ISBN 0-596-00712-4
http://www.oreilly.com/catalog/hfdesignpat/

Object-Oriented Software Construction
Bertrand Meyer
ISBN 0-13-629155-4
http://www.amazon.com/gp/product/0136291554/102-4538903-0207360?v=glance&n=283155

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including “Java 2 For Dummies” and “Eclipse For Dummies,” both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He holds a Bachelor of Science in Computer Science from Rutgers University. In his spare time, Mike facilitates the ACGNJ Java Users Group and serves as ACGNJ Secretary. Mike has co-written several articles for Java Boutique, and his computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.