Decompiling Classes

"When all else fails, read the manual."

Murphy's Technology Laws

Determining When to Decompile

In an ideal world, decompilation would probably be unnecessary, except when learning how other people who don't like to write good documentation implemented a certain feature. In the real world, however, there are often situations where a direct reference to the source code can be the best, if not the only, solution. Here are some of the reasons to decompile:

- Recovering the source code that was accidentally lost
- Learning the implementation of a feature or trick
- Troubleshooting an application or library that does not have good documentation
- Fixing urgent bugs in third-party code for which no source code exists
- Learning to protect your code from hacking

Decompiling produces the source code from Java bytecode. It is a reverse process to compiling that is possible due to the standard and well-documented structure of bytecode. Just like running a compiler to produce bytecode from the source code, you can run a decompiler to obtain the source code for given bytecode. Decompiling is a powerful method of learning about implementation logic in the absence of documentation and the source code, which is why many product vendors explicitly prohibit decompiling and reverse engineering in the license agreement. Be sure to check the license agreement or get an explicit permission from the vendor if you are uncertain about the legality of your actions.

2

IN THIS CHAPTER

Some people might argue that you shouldn't have to resort to extreme measures such as decompiling and that you should rely on vendors of the bytecode for support and bug fixing. In a professional environment, if you are a developer of an application, you are responsible for the functionality being flawless. The users and management do not care whether a bug is in your code or in third-party code. They care about the problem being fixed, and they will hold you accountable for it. Contacting the vendor of the third-party code should be a preferred way. However, in urgent cases when you must provide a solution in a matter of hours, being able to work with the bytecode will give you that extra edge over your peers, and maybe a bonus as well.

Knowing the Best Decompilers

To embark on the task of decompiling, you need the right tools. A good decompiler can produce the source code that will be almost as good as the original source code that was compiled into bytecode. Some decompilers are free, and some are commercially available. Although I support the principles behind commercial software, it needs to offer a useful premium over its free counterparts for me to use it. In the case of decompilers, I have not found the free ones lacking any features, so my personal recommendation is to use a free tool such as JAD or JODE. Table 2.1 lists some of the commonly used decompilers and includes a

STORIES FROM THE TRENCHES

At Riggs Bank we were preparing to go live with a very large and important J2EE application that was deployed into a cluster of application servers from a leading J2EE vendor. Several teams were waiting for the production environment to be ready, but for some strange reason the application server would not start on some of the hosts. The exact same installation would run on some machines but fail on others with an error message about an invalid configuration URL. To make matters worse, the URL in the error message could not be found in any of the configuration files, shell scripts, or environment variables.

Several days were spent trying to fix the problem in vain, and the situation was ready to explode because several teams were about to miss a critical deadline. After copying and reinstalling the application server failed, we finally resorted to finding the class in the application server libraries that was producing the error message. Decompiling it, and a few other classes that were using it, revealed that the URL was programmatically generated based on the server installation directory. The installation directory was determined by executing the `pwd` Unix command. It turned out that on the failing hosts there were no permissions to execute `pwd`, but the misleading error message did not make that obvious. Fixing the permissions took a matter of minutes, and the whole process from the time we found and decompiled the class took less than an hour. Thus, a looming disaster was turned into a big win for the IT team.

short description highlighting the quality of each one. The URLs presented might become outdated, so doing a Google search is typically the best way of finding the decompiler's home page and the latest version to download.

A very important criterion is how well the decompiler supports more advanced language constructs such as inner classes and anonymous implementations. Even though the bytecode format has been very stable since JDK 1.1, it is important to use a decompiler that is frequently updated by its authors. The new language features in JDK 1.5 will require an update in decompilers, so be sure to check the release date of the version you are using.

TABLE 2.1

Decompilers

TOOL/RATING	LICENSE	DESCRIPTION
JAD/Excellent	Free for noncommercial use	JAD is a very fast, reliable, and sophisticated decompiler. It has full support for inner classes, anonymous implementations, and other advanced language features. The generated code is clean, and imports are well organized. Several other decompiling environments use command-line JAD as the decompiling engine.
JODE/Excellent	GNU public license	JODE is a very good decompiler written in Java and available with the full source code on SourceForge.net. It might not be as fast and widespread as JAD, but it produces excellent results, at times even cleaner than JAD. Having the source code for the decompiler itself cannot be underestimated for educational purposes.
Mocha/Fair	Free	Mocha is the first well-known decompiler that has generated a lot of legal controversy but also a wave of enthusiasm. Mocha made it obvious that Java source code can be reconstructed almost to its original form, which was cheered by the development community but feared by the legal departments. The public code has not been updated since 1996, although Borland has presumably updated and integrated it into JBuilder.

Although you might find other decompilers on the market, JAD and JODE are certainly good enough and therefore widely used. Many products provide graphical user interfaces (GUIs) but rely on a bundled decompiler to do the actual work. For instance, Decafe, DJ, and Cavaj are GUI tools bundled with JAD and therefore were not included in the review. For the rest of this book, we will use command-line JAD to produce the source code. Most of the time, the command-line decompiler is all you need, but if you prefer to use a GUI, just be sure that it uses a solid decompiler such as JAD or JODE.

Decompiling a Class

In case you haven't used one before, let's see how good a job a decompiler can do. We will work with a slightly enhanced version of the `MessageInfo` class, which is used by Chat to send the message text and the attributes to a remote host. `MessageInfoComplex.java`, shown in Listing 2.1, has an anonymous inner class (`MessageInfoPK`) and a `main()` method to illustrate some of the more complex cases of decompiling.

LISTING 2.1 `MessageInfoComplex` Source Code

```
package covertjava.decompile;

/**
 * MessageInfo is used to send additional information with each message across
 * the network. Currently it contains the name of the host that the message
 * originated from and the name of the user who sent it.
 */
public class MessageInfoComplex implements java.io.Serializable {

    String hostName;
    String userName;

    public MessageInfoComplex(String hostName, String userName) {
        this.hostName = hostName;
        this.userName = userName;
    }

    /**
     * @return name of the host that sent the message
     */
    public String getHostName() {
        return hostName;
    }

    /**
     * @return name of the user that sent the message
     */
    public String getUserName() {
        return userName;
    }
```

```
/**
 * Convenience method to obtain a string that best identifies the user.
 * @return name that should be used to identify a user that sent this message
 */
public String getDisplayName() {
    return getUserName() + " (" + getHostName() + ")";
}


/**
 * Generate message id that can be used to identify this message in a database
 * The format is: <ID><UserName><HostName>. Names are limited to 8 characters
 * Example: 443651_Kalinovs_JAMAICA would be generated for Kalinovsky/JAMAICA
 */
public String generateMessageId() {
    StringBuffer id = new StringBuffer(22);

    String systemTime = "" + System.currentTimeMillis();
    id.append(systemTime.substring(0, 6));

    if (getUserName() != null && getUserName().length() > 0) {
        // Add user name if specified
        id.append('_');
        int maxChars = Math.min(getUserName().length(), 8);
        id.append(getUserName().substring(0, maxChars));
    }

    if (getHostName() != null && getHostName().length() > 0) {
        // Add host name if specified
        id.append('_');
        int maxChars = Math.min(getHostName().length(), 7) ;
        id.append(getHostName().substring(0, maxChars));
    }
    return id.toString();
}


/**
 * Include an example of anonymous inner class
 */
public static void main(String[] args) {
    new Thread(new Runnable() {
```

LISTING 2.1 Continued

```
            public void run() {
                System.out.println("Running test");
                MessageInfoComplex info = new MessageInfoComplex("JAMAICA", "Kalinovsky");
                System.out.println("Message id = " + info.generateMessageId());
                info = new MessageInfoComplex(null, "JAMAICA");
                System.out.println("Message id = " + info.generateMessageId());
            }
        }).start();
    }


    /**
     * Inner class that can be used as a primary key for MessageInfoComplex
     */
    public static class MessageInfoPK implements java.io.Serializable {
        public String id;
    }
}
```

After compiling `MessageInfoComplex.java` using `javac` with default options, we get three class files: `MessageInfoComplex.class`, `MessageInfoComplex$MessageInfoPK.class`, and `MessageInfoComplex$1.class`. As you might know, inner classes and anonymous classes have been added to Java in JDK 1.1, but the design goal was to preserve bytecode format compatibility with earlier versions of Java. That is why these language constructs result in somewhat independent classes, although they do retain the association with the parent class. The final step of our test is to run the decompiler on the class file and then compare the generated source code with the original. Assuming that you have downloaded and installed JAD and added it to the path, you can run it using the following command:

```
jad MessageInfoComplex.class
```

Upon completion, JAD generates the `MessageInfoComplex.jad` file. This is renamed to `MessageInfoComplex_FullDebug.jad`, as shown in Listing 2.2.

LISTING 2.2 `MessageInfoComplex` Decompiled Code

```
// Decompiled by Jad v1.5.7g. Copyright 2000 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MessageInfoComplex.java
```

```
package covertjava.decompile;

import java.io.PrintStream;
import java.io.Serializable;

public class MessageInfoComplex
    implements Serializable
{
    public static class MessageInfoPK
        implements Serializable
    {

        public String id;

        public MessageInfoPK()
        {
        }
    }


    public MessageInfoComplex(String hostName, String userName)
    {
        this.hostName = hostName;
        this.userName = userName;
    }

    public String getHostName()
    {
        return hostName;
    }

    public String getUserName()
    {
        return userName;
    }

    public String getDisplayName()
    {
        return getUserName() + " (" + getHostName() + ")";
    }

    public String generateMessageId()
```

LISTING 2.2 Continued

```
    {
        StringBuffer id = new StringBuffer(22) ;
        String systemTime = "" + System.currentTimeMillis();
        id.append(systemTime.substring(0, 6));
        if(getUserName() != null && getUserName().length() > 0)
        {
            id.append('_');
            int maxChars = Math.min(getUserName().length(), 8);
            id.append(getUserName().substring(0, maxChars));
        }
        if(getHostName() != null && getHostName().length() > 0)
        {
            id.append('_');
            int maxChars = Math.min(getHostName().length(), 7);
            id.append(getHostName().substring(0, maxChars));
        }
        return id.toString();
    }

    public static void main(String args[])
    {
        (new Thread(new Runnable() {

            public void run()
            {
                System.out.println("Running test");
                MessageInfoComplex info = new MessageInfoComplex("JAMAICA", "Kalinovsky");
                System.out.println("Message id = " + info.generateMessageId());
                info = new MessageInfoComplex(null, "JAMAICA");
                System.out.println("Message id = " + info.generateMessageId());
            }

        })).start();
    }

    String hostName;
    String userName;
}
```

Take a few moments to review the generated code. As you can see, the code is almost a 100% match to the original! The order of variables, methods, and inner class declarations is different, and so is the formatting, but the logic is absolutely the same. We have also lost the comments, but well-written Java code such as ours is self-evident, isn't it?

Our case produced good results because full debugging information is included by `javac` when the `-g` option is used. If the source code was compiled without the debug information (the `-g:none` option), the decompiled code would lose some of the clarity, such as the parameter names of methods and names of local variables. The following code shows the constructor and a method that uses local variables for `MessageInfoComplex` with no debugging information included:

```
public MessageInfoComplex(String s, String s1)
{
    hostName = s;
    userName = s1;
}

public String generateMessageId()
{
    StringBuffer stringbuffer = new StringBuffer(22);
    String s = "" + System.currentTimeMillis();
    stringbuffer.append(s.substring(0, 6));
    if(getUserName() != null && getUserName().length() > 0)
    {
        stringbuffer.append('_');
        int i = Math.min(getUserName().length(), 8);
        stringbuffer.append(getUserName().substring(0, i));
    }
    if(getHostName() != null && getHostName().length() > 0)
    {
        stringbuffer.append('_');
        int j = Math.min(getHostName().length(), 7);
        stringbuffer.append(getHostName().substring(0, j));
    }
    return stringbuffer.toString();
}
```

What Makes Decompiling Possible?

Java source is not compiled to binary machine code like C/C++ source is. Compiling Java source produces intermediate bytecode, which is a platform-independent representation of the source code. Bytecode can be interpreted or compiled after loading, which results in a two-step transformation of the high-level programming language into the low-level machine code. It is the intermediate step that makes decompiling Java bytecode nearly flawless. Bytecode carries all the significant information found in a source file. Even though the comments and formatting are lost, all the methods, variables, and programming logic are obviously preserved. Because the bytecode does not represent the lowest-level machine language, the format of the code closely resembles the source code. The JVM specification defines a set of instructions that match Java language operators and keywords, so a fragment of Java code such as

```
public String getDisplayName() {
    return getUserName() + " (" + getHostName() + ")";
}
```

is represented by the following bytecode:

```
 0 new #4 <java/lang/StringBuffer>
 3 dup
 4 aload_0
 5 invokevirtual #5 <covertjava/decompile/MessageInfoComplex.getUserName>
 8 invokestatic #6 <java/lang/String.valueOf>
11 invokestatic #6 <java/lang/String.valueOf>
14 invokespecial #7 <java/lang/StringBuffer.<init>>
17 ldc #8 < (>
19 invokevirtual #9 <java/lang/StringBuffer.append>
22 aload_0
23 invokevirtual #10 <covertjava/decompile/MessageInfoComplex.getHostName>
26 invokevirtual #9 <java/lang/StringBuffer.append>
29 ldc #11 <)>
31 invokevirtual #9 <java/lang/StringBuffer.append>
34 invokestatic #6 <java/lang/String.valueOf>
37 invokestatic #6 <java/lang/String.valueOf>
40 areturn
```

Bytecode format is covered in detail in Chapter 17, “Understanding and Tweaking Bytecode,” but you can see the resemblance by just looking at the bytecode. The decompiler loads the bytecode and tries to reconstruct the source code based on the bytecode instructions. The names of class methods and variables are typically preserved, whereas the names of method parameters and local variables are lost. If the debugging information is available, it provides the decompiler with parameter names and line numbers—and that makes the reconstructed source file even more readable.

Potential Problems with Decompiled Code

Most of the time, decompiling produces a readable file that can be changed and recompiled. However, on some occasions decompiling does not render a file that can be compiled again. This can happen if the bytecode was obfuscated, and the names given by the obfuscator result in ambiguity at the compilation. The bytecode is verified when loaded, but the verifications assume that the compiler has checked for a number of errors. Thus, the bytecode verifiers are not as strict as compilers and obfuscators can take advantage of that to better protect the intellectual property. For example, here is the JAD output on the anonymous inner class from the `MessageInfoComplex main()` method that was obfuscated by the Zelix ClassMaster obfuscator:

```
    static class c
        implements Runnable
    {

        public void run()
        {
            boolean flag = a.b;
            System.out.println(a("*4%p\002\026&kj\016\0135"));
            b b1 = new b(a("2\000\006_\";\0"), a("3 'w\005\02778u\022"));
            System.out.println(a("5$8m\n\037$kw\017X¦k").concat(String.valueOf
            ➥(String.valueOf(b1.d()))));
            b1 = new b(null, a("2\000\006_\";\0"));
            System.out.println(a("5$8m\n\037$kw\017X¦k").concat(String.valueOf
            ➥(String.valueOf(b1.d()))));
            if(flag)
                b.c = !b.c;
        }

        private static String a(String s)
        {
            char ac[];
            int i;
            int j;
            ac = s.toCharArray();
            i = ac.length;
            j = 0;
            if(i > 1) goto _L2; else goto _L1
_L1:
            ac;
            j;
```

```
_L10:
            JVM INSTR dup2 ;
            JVM INSTR caload ;
            j % 5;
            JVM INSTR tableswitch 0 3: default 72
        //                        0 52
        //                        1 57
        //                        2 62
        //                        3 67;
               goto _L3 _L4 _L5 _L6 _L7
_L4:
            0x78;
              goto _L8
_L5:
            65;
              goto _L8
_L6:
            75;
              goto _L8
_L7:
            30;
              goto _L8
_L3:
            107;
_L8:
            JVM INSTR ixor ;
            (char);
            JVM INSTR castore ;
            j++;
            if(i != 0) goto _L2; else goto _L9
_L9:
            ac;
            i;
              goto _L10
_L2:
            if(j >= i)
                return new String(ac);
            if(true) goto _L1; else goto _L11
_L11:
        }

    }
```

As you can see, it is a total fiasco, not even closely resembling Java source. What's more disturbing, JAD produced source code that cannot be compiled. The other two decompilers have reported an error on the class file. Needless to say, the JVM recognizes and loads the bytecode in question with no problems. Obfuscation is covered in detail in Chapter 3, "Obfuscating Classes."

A powerful way of protecting the intellectual property is encoding the class files and using a custom class loader to decode them on loading. This way, the decompilers cannot be used on any of the application classes except for the entry point and the class loader. Although not unbreakable, encoding makes hacking much more difficult. A hacker would first have to decompile the class loader to understand the decoding mechanism and then decode all the class files; only then could he proceed with decompiling. Chapter 19, "Protecting Commercial Applications from Hacking," provides information on how to best protect the intellectual property in Java applications.

Quick Quiz

1. What are the reasons to decompile bytecode?
2. Which compiler options affect the quality of decompilation, and how?
3. Why is decompiled Java bytecode almost identical to the source code?
4. How can you protect the bytecode from decompiling?

In Brief

- Decompiling produces the source code from bytecode, which is almost identical to the original.
- Decompiling is a powerful method of learning about implementation logic in the absence of documentation and source code. However, decompiling and reverse engineering might be explicitly prohibited in the license agreement.
- Decompiling requires downloading and installing a decompiler.
- Decompiling Java classes is effective because the bytecode is an intermediate step between the source code and machine code.
- A good obfuscator can make decompiled code very hard to read and understand.