

## 5.0 Sample Development Chapter

### 5.1 Overview

This is an example of a development chapter for a project thesis. The chapter documents lessons learned, or new things uncovered during development.

This project had a formal requirements and design phase. During development, however, we discovered many things that did not work as expected. Why do we have a development chapter in the thesis? So we can document the knowledge obtained during our work.

This is a sample development chapter. The organization is by technology: J2SE itself, JDBC, JAI, and so on. You may use a similar outline, or your own. The important thing is to communicate what you learned during development. The chapter organization is of little matter.

One more thing: the writing in this sample is rather informal. It is appropriate for a homework or project assignment, but not a thesis. Bear this in mind.

### 5.2 J2SE

These observations deal with the core Java 2 Standard Edition (J2SE) environment, and not a companion sub-technology.

#### 5.2.1 Where Oh Where Does My Little Object Go

One difference between OO languages (like Java) and a structured languages with OO enhancements (like C++ or VBA) is the former do not allow declaring a common block of memory shared by various program units. A pure OO language, in fact, only allows communication by methods. Java at least allows defining object references as `static`, which means they are allocated storage space prior to program load.

In Java, each GUI screen is its own object. Controls such as buttons are attached to some method, perhaps internal to or external from the screen object, that are activated when the control receives mouse focus. In our design we use internal methods: the Login object, for example, displays its own screen and monitors its own buttons.

But what happens when we need to share parameters across objects? For example, the Login object establishes a database connection to check parameter validity. The Database object then needs to either use this connection or a get a copy of the parameters to establish its own connection.

There are at least two (perhaps more) solutions to this problem:

1. Build an external control object that launches screens, hosts control interactions, and shares common data, or
2. Build a start-up object holding all sub objects as `static` members and grant `public` access to sub-objects for populating common data.

In our case, we chose the latter approach: we built an object called Logger whose sole purpose is to store `static` object references and launch the initial GUI screen. Each screen then manages its own controls and populates common data stored by Logger.

#### 5.2.1 A URL Is Not A File Name

Java supports operating system specific approaches to interacting with a file system: a `String` containing a file name, for example, is parsed by the underlying operating system to open a file. A multi-platform interface to the file system is also supported, and that uses Uniform Resource Locator (URL) syntax.

Even though both methods can reference the same file, they are not interchangeable. A URL is a file system abstraction, but a `String` is a direct interface. In this project we use the former for convenience. But integrating this project with some web servers might require the use of both methods.

### [5.2.2 Finish Your Dinner Before Dessert](#)

One problem we had with the original proof of concept code was advancing the video clip to another position before the file save was committed to disk. We used a JAI convenience object (more on this later) to write stills to disk. This convenience object interfaced with the file system directly. But sometimes the frame being copied got updated before the copy finished.

We worked around this problem by using a different convenience function, one that supported a `FileOutputStream`. Java provides methods for closing and flushing this object. Consequently we could call those methods, thus guaranteeing the write operation completed, before leaving the save method. This allowed us to postpone advancing the clip position until after I/O completed.

It seems this asynchronous behavior is found throughout Java. We documented another case at proof of concept. These are examples of code changes implemented due to actual object behavior.

## 5.3 JMF

These observations deal with the Java Media Framework (JMF) API.

### [5.3.1 You Can't Get There From Here](#)

Saving stills is somewhat problematic because we have to convert from one Java API layer to another. The JMF deals with time-ordered media, such as audio and video. It does not handle time-independent media, such as stills. That is the province of the JAI.

Going between these two technologies require using a common base, the Abstract Window Toolkit (AWT). So converting a frame to a still requires first converting the frame to an AWT buffer, then the AWT buffer into something that can be saved on disk. We find interesting Java's approach towards technology integration: morphing objects to and from the lowest common denominator.

### [5.3.2 It Works When It Doesn't](#)

One rule of OO programming is objects are responsible for their own states. If, for example, one attempts to set a parameter to an out of range value, the object should simply ignore the request: garbage in does not necessarily warrant garbage out. When this happens, however, the method should indicate the attempted operation failed.

The JMF does not honor this protocol for setting the current media time:

```
public void setMediaTime(Time now)
```

What happens if, say, the clip is 175 seconds long and we try to set the clip position to 180 seconds? Can we know this failed? Not with a void return signature. So what can the object do? Throw an exception. Does it? No.

Consequently code calling the services of `setMediaTime` must police its own behavior. We find this to be against the spirit of object oriented programming.

## 5.4 JAI

The following observations deal with Java Advanced Imaging (JAI):

### [5.4.1 Convenience Objects A'int](#)

The JAI has a "convenience" object called, strangely enough, JAI that integrates several operations in one static method. This object works like `printf` as the first parameter is a command string and the remaining number and type of parameters vary depending on the value of that string.

The JAI object allows one to quickly load, convert, and store images without understanding the underlying details of how Java interacts with either images or the file system. Therein lies the problem: the associated methods offer no control on when operations are completed. Many of these methods are asynchronous.

Earlier we documented how the use of one all-inclusive JAI method yielded unexpected results in our application. We were forced to use a lower-level method that gave us more control. This seems to be a fundamental issue with Java: often the easy way is the wrong way.

## 5.5 JDBC

The following observations deal with the Java Data Base Component (JDBC):

### [5.5.1 Yesterday All My Troubles Seem So Far Away](#)

The JDBC offers two base methods for executing an SQL statement against a database: `execute` and `executeQuery`. The former is generic for all SQL statements and the latter is specialized for statements that return a result set.

What does this mean? All SELECT statements return result sets, so `executeQuery` is appropriate. But what about other statements? Does INSERT return a result set?

This is up to the interpretation of the database driver author. We know some SELECT statements return result sets without any rows. And MySQL, for example, reports zero rows returned for all SQL statements executed from the command line. So it's possible to use `executeQuery` for all SQL statements by returning a row count of zero and an empty result set when warranted.

Version 2 of the MySQL JDBC driver did this, but Version 3 does not. We had to change our INSERT statements to use the `execute` call instead. Not a big deal, but Java strives to be a "write once, run anywhere" language. That's somewhat difficult to pull off when interpretations of what an API means is left to multiple driver developers.

### [5.5.2 Auto Increment Isn't](#)

Our database tables have primary keys defined as:

```
id INTEGER PRIMARY KEY AUTO_INCREMENT
```

which inserting rows without generating keys to make rows unique. But in our Still table, we have a reference to the Clip table:

```
FOREIGN KEY (clipId) REFERENCES Clips(id)
```

Which is fine. Except don't we have to know the value of clipId before inserting a row? And if the value of Clips.id is supplied by the database, how do we get its value?

Fortunately we first add exactly one row to Clips and then add multiple rows to Stills. In exactly that order. Good thing a property of automatic keys is the last row added will have the highest value. So we can use an INSERT...SELECT statement to query Clips for its maximum key value before populating Stills.

We got lucky. Our process allows us to use automatic keys without storing intermediate results. But this sequence may not work for all database applications.