Clean Code: Args – A Command-line Argument Parser

Robert C. Martin
26 November 2005

Most of us have had to parse command line arguments from time to time. If we don't have a convenient utility, then we simply walk the array of strings that is passed into the main function. There are several good utilities available from various sources; but none of them do exactly what I want. So, of course, I decided to write my own. I call it: `Args`.

`Args` is very simple to use. You simply construct the `Args` class with the input arguments, and a format string, and then query the `Args` instance for the values of the arguments. Consider the following simple example:

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argument error: %s\n", e.errorMessage());
  }
}
```

You can see how simple this is. We simply create an instance of the `Args` class with two parameters. The first parameter is the format, or schema, string: `"l,p#,d*"`. It defines three command-line arguments. The first: `-l`, is a Boolean argument. The second: `-p`, is an integer argument. The third: `-d`, is a string argument. The second parameter to the `Args` constructor is simply the array of command-line argument passed into main.

If the constructor returns without throwing an `ArgsException`, then the incoming command-line was parsed, and the `Args` instance is ready to be queried. Methods like `getBoolean`, `getInteger`, and `getString` allow us to access the values of the arguments by their names.

If there is a problem, either in the format string, or in the command-line arguments themselves, an `ArgsException` will be thrown. A convenient description of what went wrong can be retrieved from the `errorMessage` method of the exception.

Args Implementation

Here is the implementation of the `Args` class. Please read it very carefully. I consider it to be an example of very clean code. The style of this code is something I think is worth emulating.

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;
  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();

    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
  }

  private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
      if (element.length() > 0)
        parseSchemaElement(element.trim());
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
      marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
      throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
      throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
  }

  private void parseArgumentStrings(List<String> argsList) throws ArgsException
  {
    for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
      String argString = currentArgument.next();
      if (argString.startsWith("-")) {
        parseArgumentCharacters(argString.substring(1));
      } else {
        currentArgument.previous();
        break;
      }
    }
  }

  private void parseArgumentCharacters(String argChars) throws ArgsException {
    for (int i = 0; i < argChars.length(); i++)
```

```
      parseArgumentCharacter(argChars.charAt(i));
  }

  private void parseArgumentCharacter(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
      throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
    } else {
      argsFound.add(argChar);
      try {
        m.set(currentArgument);
      } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
      }
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public int nextArgument() {
    return currentArgument.nextIndex();
  }

  public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
  }
}
```

Notice that you can read this code from the top to the bottom without a lot of jumping around or looking ahead. The one thing you may have had to look ahead for is the definition of `ArgumentMarshaler`; which I left out intentionally. Having read this code carefully, you should understand what the `ArgumentMarshaler` interface is, and what its derivatives do. I'll show a few of them to you now.

```
public interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
}
```

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
  private boolean booleanValue = false;
```

```
  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

The other `ArgumentMarshaler` derivatives simply replicate this pattern for doubles, and string arrays; and would serve to clutter this paper. I'll leave them to you as an exercise.

One other bit of information might be troubling you: the definition of the error code constants. They

are in the `ArgsException` class:

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode,
                       char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() {
    switch (errorCode) {
      case OK:
        return "TILT: Should not get here.";
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
```

```
                            errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                            errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                            errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                            errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                            errorArgumentId);
      case INVALID_ARGUMENT_NAME:
        return String.format("'%c' is not a valid argument name.",
                            errorArgumentId);
      case INVALID_ARGUMENT_FORMAT:
        return String.format("'%s' is not a valid argument format.",
                            errorParameter);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

It's remarkable how much code is required to flesh out the details of this simple concept. One of the reasons for this is that we are using a particularly *wordy* language. Java, being a statically typed language, requires a lot of words in order to satisfy the type system. In a language like Ruby, Python, or Smalltalk, this program would be *much* smaller.

Please read the code over one more time. Pay special attention to the way things are named, the size of the functions, and the formatting of the code. If you are an experienced programmer, you may have some quibbles here and there with various parts of the style or structure. Overall, however, I would expect you to conclude that this program is very nicely written, with a very clean structure.

For example, it should be *obvious* how you would add a new argument type, such as a date argument, or a complex number argument; and that such an addition would require a trivial amount of effort. In short, it would simply require a new derivative of `ArgumentMarshaler`, a new `getXXX` function, and a new case statement in the `parseSchemaElement` function. There would also probably be a new `Args-Exception.ErrorCode`, and a new error message.

How did I do this?

Let me set your mind at rest. I did not simply write this program from beginning to end in its current form. More importantly, I am not expecting you to be able to write clean and elegant programs in one pass. If we have learned anything over the last couple of decades, it is that programming is a craft, more than it is a science. To write clean code, you must first write dirty code; and then clean it.

This should not be a surprise to you. We learned this truth in grade school when our teachers tried (usually in vain) to get us to write rough drafts of our compositions. The process, they told us, was that we should write a rough draft, then a second draft, then N subsequent drafts until we had our final version. Writing clean compositions, they tried to tell us, is a matter of *successive refinement*.

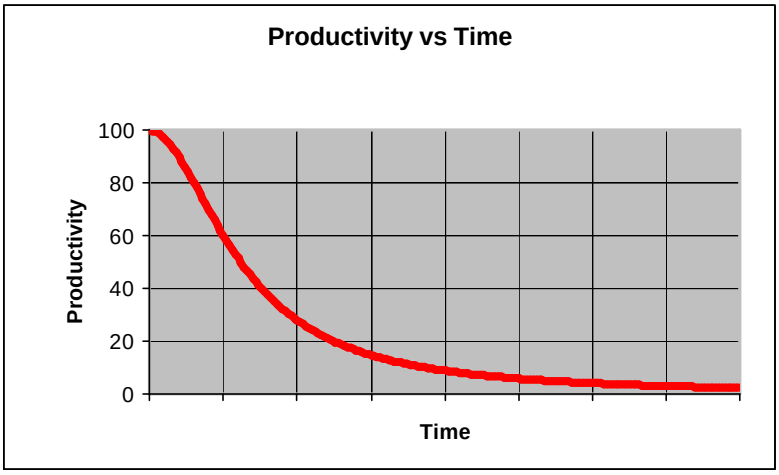
Most freshman programmers (like most gradeshoolers) don't follow this advice particularly well. They believe that the primary goal is to get the program working. Once it's "working" they move on to the

next task, leaving the "working" program in whatever state they finally got it to "work". Most seasoned programmers know that this is professional suicide.

Is Successive Refinement Cost Effective?

If you have been a programmer for more than two or three years, you have probably been significantly slowed down by someone else's messy code. If you have been a programmer for longer than two or three years, you have probably been slowed down by *your own* messy code. The degree of the slow-down can be significant. Over the span of a year or two, teams that were moving very fast at the beginning of a project can find themselves moving at a snail's pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial. Every addition or modification to the system requires that the tangles, twists, and knots be "understood" so that more tangles, twists, and knots can be added. Over time the mess becomes so big and so deep and so tall, they can not clean it up. There is no way at all.

As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero. As productivity decreases, management does the only thing they can; they add more staff to the project in hopes of increasing productivity. But that new staff is not versed in the design of the system. They don't know the difference between a change that matches the design intent, and a change that thwarts the design intent. Furthermore, they, and everyone else on the team are under horrific pressure to increase productivity. So they all make more and more messes, driving the productivity ever further towards zero.



Eventually the team rebels; and they inform management that they cannot continue to develop in this horrific code base. They demand a redesign. Management does not want to expend the resources on a whole new redesign of the project, but they cannot deny that productivity is terrible. Eventually they bend to the demands of the developers, and authorize *the grand redesign in the sky*.

A new tiger team is selected. Everyone wants to be on this team because it's a green-field project. They get to start over and create something truly beautiful. But only the best and brightest are chosen for the tiger team. Everyone else must continue to maintain the current system.

Now the two teams are in a race. The tiger team must build a new system that does everything that the old system does. Not only that, they have to keep up with the changes that are continuously being made to the old system. Management will not replace the old system until the new system can do everything that the old system does, on the day of the change.

This race can go on for a very long time. I've seen it take 10 years. And by the time it's done, the original members of the tiger team are long gone, and the current members are demanding that the new system be redesigned because it's such a mess.

If you have experienced even one small part of the story I just told, then you already know that spending time keeping your code clean is not just cost effective; it's a matter of professional survival.

Args: the rough draft.

Below is an early version of the `Args` class. It "works". And it's messy.

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Character, String>();
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
      parseIntegerSchemaElement(elementId);
```

```
  } else {
    throw new ParseException(
      String.format("Argument: %c has invalid format: %s.",
                    elementId, elementTail), 0);
  }
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
  }
}

private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, false);
}

private void parseIntegerSchemaElement(char elementId) {
  intArgs.put(elementId, 0);
}

private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
  return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
  return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
  for (currentArgument = 0; currentArgument < args.length; currentArgument++)
  {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
```

```
    valid = false;
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  if (isBooleanArg(argChar))
    setBooleanArg(argChar, true);
  else if (isStringArg(argChar))
    setStringArg(argChar);
  else if (isIntArg(argChar))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.put(argChar, new Integer(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}

private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private boolean isStringArg(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}
```

```
public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" unexpected.");

  return message.toString();
}

public boolean getBoolean(char arg) {
  return falseIfNull(booleanArgs.get(arg));
}

private boolean falseIfNull(Boolean b) {
  return b != null && b;
}

public String getString(char arg) {
  return blankIfNull(stringArgs.get(arg));
}

public int getInt(char arg) {
  return zeroIfNull(intArgs.get(arg));
}

private int zeroIfNull(Integer i) {
  return i == null ? 0 : i;
}

private String blankIfNull(String s) {
  return s == null ? "" : s;
}
```

```
  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }
}
```

I hope your initial reaction to this mass of code is: "I'm certainly glad he didn't leave it like *that*!" If you feel like this, then remember that's how other people are going to feel about code that you leave in rough-draft form.

Actually "rough-draft" is probably the kindest thing you can say about this code. It's clearly a work-in-progress. The sheer number of instance variables is daunting. The odd strings like "TILT", the `HashSets` and `TreeSets`, and the try-catch-catch blocks, all add up to a festering pile.

I had not wanted to write a festering pile. Indeed, I was trying to keep things reasonably well organized. You can probably tell that from my choice of function and variable names; and the fact that there is a crude structure to the program. But, clearly, I had let the problem get away from me.

The mess built gradually. Earlier versions had not been nearly so nasty. For example, here is an earlier version in which only Boolean arguments were working.

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
  }

  private boolean parseSchema() {
    for (String element : schema.split(",")) {
      parseSchemaElement(element);
    }
    return true;
```

```
}

private void parseSchemaElement(String element) {
  if (element.length() == 1) {
    parseBooleanSchemaElement(element);
  }
}

private void parseBooleanSchemaElement(String element) {
  char c = element.charAt(0);
  if (Character.isLetter(c)) {
    booleanArgs.put(c, false);
  }
}

private boolean parseArguments() {
  for (String arg : args)
    parseArgument(arg);
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (isBoolean(argChar)) {
    numberOfArguments++;
    setBooleanArg(argChar, true);
  } else
    unexpectedArguments.add(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return numberOfArguments;
}

public String usage() {
  if (schema.length() > 0)
     return "-["+schema+"]";
  else
    return "";
}

public String errorMessage() {
  if (unexpectedArguments.size() > 0) {
    return unexpectedArgumentMessage();
  } else
    return "";
```

```
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

Although you can find plenty to complain about in this code, it's really not all that bad. It's compact and simple, and easy to understand. However, within this code it is easy to see the seeds of the later festering pile. It's quite clear how this grew into the latter mess.

Notice that the latter mess has only two more argument types than this: string and integer. The addition of just two more argument types had a massive negative impact on the code. It converted it from something that would have been reasonably maintainable, into something that I would expect to become riddled with bugs and warts.

I added the two argument types incrementally. First I added the string argument, which yielded this:

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs =
    new HashMap<Character, String>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgument = '\0';

  enum ErrorCode {
    OK, MISSING_STRING}

  private ErrorCode errorCode = ErrorCode.OK;

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return valid;
  }
```

```
private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
  else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
  }

}

private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
  return elementTail.length() == 0;
}

private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, false);
}

private boolean parseArguments() {
  for (currentArgument = 0; currentArgument < args.length; currentArgument++)
  {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}
```

```
private void parseElement(char argChar) {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    valid = false;
  }
}

private boolean setArgument(char argChar) {
  boolean set = true;
  if (isBoolean(argChar))
    setBooleanArg(argChar, true);
  else if (isString(argChar))
    setStringArg(argChar, "");
  else
    set = false;

  return set;
}

private void setStringArg(char argChar, String s) {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgument = argChar;
    errorCode = ErrorCode.MISSING_STRING;
  }
}

private boolean isString(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  if (unexpectedArguments.size() > 0) {
    return unexpectedArgumentMessage();
  } else
    switch (errorCode) {
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgument);
```

```
        case OK:
          throw new Exception("TILT: Should not get here.");
      }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
}
```

You can see that this is starting to get out of hand. It's still not horrible; but the mess is certainly starting to grow. It's a pile, but it's not festering quite yet. It took the addition of the integer argument type to get this pile really fermenting and festering.

So I stopped.

I had at least two more argument types to add, and I could tell that they would make things much worse. If I bulldozed my way forward, I could probably get them to work; but I'd leave behind a mess that was too large to fix. If the structure of this code was ever going to be maintainable, now was the time to fix it.

So I stopped adding features, and started refactoring. Having just added the string and integer arguments, I knew that each argument type required new code in three major places. First, each argument type required some way to parse its schema element, in order to select the `HashMap` for that type. Next, each argument type needed to be parsed in the command-line strings and converted to its true type. Finally, each argument type needed a `getXXX` method so that it could be returned to the caller as its true type.

Many different types, all with similar methods, that sounds like a class to me. And so the `ArgumentMarshaler` concept was born.

On Incrementalism

One of the best ways to ruin a program is to make massive changes to its structure in the name of improvement. Some programs never recover from such "improvements". The problem is that it's very hard to get the program working the same way it worked before the "improvement".

To avoid this, I use the discipline of Test Driven Development (TDD). One of the central doctrines of this approach is to *keep the system running at all times*. In other words, using TDD, I am not allowed to make a change to the system that breaks that system. Every change I make must keep the system working, as it worked before.

To achieve this, I need a suite of automated tests that I can run on a whim, that verifies that the behavior of the system is unchanged. For the `Args` class, I had created a suite of unit and acceptance tests while I was building the festering pile. The unit tests were written in Java and administered by JUnit. The acceptance tests were written as wiki pages in FitNesse. I could run these tests any time I wanted, and if they passed, I was confident that the system was working as I specified.

So I proceeded to make a large number of very tiny changes. Each change moved the structure of the system towards the `ArgumentMarshaler` concept. And yet, each change kept the system working. The first change I made was to add the skeleton of the `ArgumentMarshaller` to the end of the festering pile.

```
 private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Clearly this wasn't going to break anything. So then I made the simplest modification I could, that would break as little as possible. I changed the `HashMap` for the Boolean arguments to take an `ArgumentMarshaler`.

```
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
```

This broke a few statements, which I quickly fixed.

```
  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
  }
```

```
  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).setBoolean(value);
  }
```

```
  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
  }
```

Notice how these changes are in exactly the areas that I had mentioned before. The parse, set, and get for the argument type. Unfortunately, small as this change was, some of the tests started failing. If you look carefully at `getBoolean` you'll see that if you call it with `'y'`, but there is no `'y'` argument, then `booleanArgs.get('y')` will return `null`; and the function will throw a `NullPointer` exception. The `falseIfNull` function had been used to protect against this, but the change I made caused that function to become irrelevant.

Incrementalism demands that I get this working quickly, before making any other changes. Indeed, the fix is not too difficult. I just have to move the check for `null`. It is no longer the `boolean` that I need to check, it's the `ArgumentMarshaller`.

First I removed the `falseIfNull` call in the `getBoolean` function. It was useless now, so I also eliminated the function itself. The tests still failed in the same way, so I was confident that I hadn't introduced any new errors.

```
public boolean getBoolean(char arg) {
  return booleanArgs.get(arg).getBoolean();
}
```

Next I split the function into two lines and put the `ArgumentMarshaller` into its own variable.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler argumentMarshaler = booleanArgs.get(arg);
  return argumentMarshaler.getBoolean();
}
```

I didn't care for the long variable name; it was badly redundant and cluttered up the function. So I shortened it to 'am'.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am.getBoolean();
}
```

And then I put in the `null` detection logic.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && am.getBoolean();
}
```

At this point you might be wondering if you should keep reading. After all, if I continue to walk you through the changes I made in such fine grained detail, it's going to take a lot of words and space. I suspect that you might not have the time to invest in such a project. So from this point on I'm going to pick up the pace and spare you the minute details.

Keep in mind, however, that even though we're going to start moving faster through the chain of refactorings, each one was made just like the one above. The steps were *tiny*.

String Arguments

Adding `String` arguments was very similar to adding `boolean` arguments. I had to change the `HashMap`, and get the parse, set, and get functions working. There shouldn't be any surprises in what follows except, perhaps, that I seem to be putting all the marshalling implementation in the `ArgumentMarshaller` base class instead of distributing it to the derivatives. All in good time, dearie.

```
  private Map<Character, ArgumentMarshaler> stringArgs =
      new HashMap<Character, ArgumentMarshaler>();
```

```
  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler());
  }
```

```
  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).setString(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

```
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : am.getString();
  }
```

```
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }
  }
```

Again, these changes were made one at a time and in such a way that the tests kept running, if not passing. When a test broke, I made sure to get it passing again before continuing with the next change.

By now you should be able to see my intent. One I get all the current marshalling behavior into the `ArgumentMarshaler` base class, I'm going to start pushing that behavior down into the derivatives. This will allow me to keep everything running while I gradually change the shape of this program.

The obvious next step is to move the integer argument functionality into the `ArgumentMarshaler`. Again, there aren't any surprises.

```
  private Map<Character, ArgumentMarshaler> intArgs =
     new HashMap<Character, ArgumentMarshaler>();
```

```
  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
  }
```

```
  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
```

```
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }
```

```
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
  }
```

```
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }
  }
```

With all the marshalling moved to the `ArgumentMarshaler`, I can start pushing functionality into the derivatives. My first step is to move the `setBoolean` function into the `BooleanArgumentMarshaller`, and make sure it gets called correctly. So I'll delete setBoolean from ArgumentMarshaller, and replace it with an abstract `set` method.

```
  private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
```

```
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
    return stringValue == null ? "" : stringValue;
  }

  public void setInteger(int i) {
    integerValue = i;
  }

  public int getInteger() {
    return integerValue;
  }

  public abstract void set(String s);
}
```

Then I'll implement the set method in `BooleanArgumentMarshaller`.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  public void set(String s) {
    booleanValue = true;
  }
}
```

And finally I'll replace the call to `setBoolean` with a call to `set`.

```
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).set("true");
}
```

The tests all still pass, but now `set` is being deployed to the `BooleanArgumentMarshaler`. So now I can remove the `setBoolean` method from the `ArgumentMarshaler` base class.

Next, I want to deploy the `get` method into `BooleanArgumentMarshaler`. Deploying get functions is always ugly because the return type always has to be `Object`, and in this case needs to be cast to a `Boolean`.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean)am.get();
}
```

Just to get this to compile, I add the `get` function to the `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  ...

  public Object get() {
    return null;
  }
}
```

This compiles, and obviously fails the tests. Getting the tests working again is simply a matter of making `get` abstract, and implementing it in `BooleanAgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
  ...

  public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
```

```
    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

Once again the tests pass. Now both `get` and `set` have been deployed to the `BooleanArgumentMarshaler`! This means I can remove the old `getBoolean` function from `ArgumentMarshaler`, and move the protected `booleanValue` variable down to `BooleanArgumentMarshaler`, and make it private.

Now I can do the same pattern of changes for strings. I can deploy both set and get, delete the unused functions, and move the variables.

```
  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

```
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : (String) am.get();
  }
```

```
private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
```

```
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {

    public void set(String s) {

    }

    public Object get() {
      return null;
    }
  }
}
```

Finally, we can repeat the process for integers. This is just a little more complicated because integers need to be parsed, and the parse operation can throw an exception. But the result is better because the whole concept of `NumberFormatException` gets buried in the `IntegerArgumentMarshaler`.

```
  private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
```

```
  private void setBooleanArg(char argChar) {
    try {
      booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
```

```
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
  }
```

```
  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
```

```
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
```

```
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

Of course the tests continue to pass. Now we can get rid of the three different maps up at the top of the algorithm, and make the whole system much more generic. However, we can't do that just by deleting them, because that would break the system. Instead, we'll add a new Map for the `ArgumentMarshaler`, and then one-by-one change the methods to use it instead of the three original maps.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

```
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

Of course the tests all still pass. Now we need to change `isBooleanArg` from this:

```
  private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
  }
```

To this:

```
  private boolean isBooleanArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
  }
```

The tests still pass. So now we can make the same change to `isIntArg`, and `isStringArg`.

```
  private boolean isIntArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
  }
```

```
  private boolean isStringArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
```

```
}
```

The tests still pass. Now we can eliminate all the duplicate calls to `marshalers.get` as follows:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (isBooleanArg(m))
    setBooleanArg(argChar);
  else if (isStringArg(m))
    setStringArg(argChar);
  else if (isIntArg(m))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(ArgumentMarshaler m) {
  return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(ArgumentMarshaler m) {
  return m instanceof StringArgumentMarshaler;
}

private boolean isBooleanArg(ArgumentMarshaler m) {
  return m instanceof BooleanArgumentMarshaler;
}
```

But now there's no good reason for the three `isxxxArg` methods. So we can just inline them:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(argChar);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
```

Now we can start using the `marshalers` map in the `set` functions, breaking the use of the other three maps. We start with the `booleans`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(m);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
```

```
private void setBooleanArg(ArgumentMarshaler m) {
  try {
    m.set("true"); // was: booleanArgs.get(argChar).set("true");
  } catch (ArgsException e) {
```

```
    }
  }
```

The tests still pass, so now we can do the same with `Strings` and `Integers`. This allows us to integrate some of the ugly exception management code into the `setArgument` function.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

OK, we're getting close to being able to remove the three old maps. First, we need to change the `getBoolean` function from this:

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
  }
```

To this:

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
```

```
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }
```

This last change might have been a surprise. Why did I suddenly decide to deal with the `ClassCastException`? The reason is that I have a set of unit tests, and a separate set of acceptance tests written in Fitnesse. It turns out that the FitNesse tests made sure that if you called `getBoolean` on a non-boolean argument, you got a false. The unit tests did not. Up to this point, I had only been running the unit tests.

This points out a danger with using acceptance tests and unit tests. Unit test coverage can languish if the programmer knows that the acceptance tests are covering. I fixed this here by adding a unit test that ran all the acceptance tests.

This last change allows us to pull out another use of the boolean map:

```
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

And now we can delete the boolean map.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

Now we can migrate the `String` and `Integer` arguments in the same manner, and do a little cleanup with the booleans.

```
  private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  }

  private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  }

  private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
  }
```

```
  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
```

```
    }
  }
```

```
public class Args {
...
- private Map<Character, ArgumentMarshaler> stringArgs =
-   new HashMap<Character, ArgumentMarshaler>();
- private Map<Character, ArgumentMarshaler> intArgs =
-   new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

Now we can inline the three parse methods, since they don't do much anymore:

```
  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
  }
```

OK, so now let's look at the whole picture again. Here's the current form of the `Args` class.

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
```

```
  } catch (ArgsException e) {
  }
  return valid;
}

private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (isStringSchemaElement(elementTail))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (isIntegerSchemaElement(elementTail)) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  } else {
    throw new ParseException(String.format(
      "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
  }
}

private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
  return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
  return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
  for (currentArgument=0; currentArgument<args.length; currentArgument++) {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}
```

```
private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  try {
    m.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private void setBooleanArg(ArgumentMarshaler m) {
  try {
    m.set("true");
```

```
  } catch (ArgsException e) {
  }
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" unexpected.");

  return message.toString();
}

public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}

public String getString(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
    return "";
  }
}
```

```
public int getInt(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? 0 : (Integer) am.get();
  } catch (Exception e) {
    return 0;
  }
}

public boolean has(char arg) {
  return argsFound.contains(arg);
}

public boolean isValid() {
  return valid;
}

private class ArgsException extends Exception {
}

private abstract class ArgumentMarshaler {
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
```

```
    }
  }
}
```

After all that work, this is a bit disappointing. The structure might be a bit better, but we still have all those variables up at the top, there's still a horrible type-case in `setArgument`, and all those `set` functions are really ugly. Not to mention all the error processing. We still have a lot of work ahead of us.

How long did it take to get to this point?

To build the festering pile took me about 5 hours of work. That may seem like a lot of time, and it is. However, I'm a busy guy. My phone rings all the time. I travel a lot. I have a dozen people working for me on various jobs around the world. So that five hours was spread over seven sessions that began on the 12th of November, 2005, and ended 11 days later on the 23rd. I suppose that if I had been able to truly concentrate on what I was doing, I might have gotten the whole festering pile built in about three hours.

How much time have I spent refactoring it? The work you see above started at 11:52 on the 23rd, and got to the current point at 12:57. So, just about an hour so far, and there's still a lot to do.

Refactoring takes time. Is it worth it? Let's keep pressing on, and we'll see.

I'd really like to get rid of that type-case up in `setArgument`. What I'd like in `setArgument` is a single call to `ArgumentMarshaler.set`. This means I need to push `setIntArg`, `setStringArg`, and `setBooleanArg` down into the appropriate `ArgumentMarshaler` derivatives. But there is a problem.

If you look closely at `setIntArg`, you'll notice that it uses two instance variables: `args`, and `currentArg`. To move `setIntArg` down into `BooleanArgumentMarshaler`, I'll have to pass both `args` and `currentArgs` as function arguments. That's dirty. I'd rather pass one argument instead of two. Fortunately, there is a simple solution. We can convert the `args` array into a list, and pass an `Iterator` down to the `set` functions. The following took me 10 steps, passing all the tests after each. But I'll just show you the result. You should be able figure out what most of the tiny little steps were.

```
public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
```

```
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }

    return true;
  }
```

```
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      m.set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

These were simple changes that kept all the tests passing. Now we can start moving the set functions down into the appropriate derivatives. First, I need to make the following change in `setArgument`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

This change is important because we want to completely eliminate the if-else chain. Therefore we needed to get the error condition out of it.

Now we can start to move the `set` functions. The `setBooleanArg` function is trivial, so we'll prepare that one first. Our goal is to change the `setBooleanArg` function to simply forward to the `BooleanArgumentMarshaler`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
```

```
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m, currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

```
  private void setBooleanArg(ArgumentMarshaler m,
                             Iterator<String> currentArgument)
                             throws ArgsException {
    try {
      m.set("true");
    catch (ArgsException e) {
    }
  }
```

Didn't we just put that exception processing in? Putting things in so you can take them out again is pretty common in refactoring. The smallness of the steps, and the need to keep the tests running, means that you move things around a lot.

Refactoring is a lot like solving a Rubick's cube. There are lots of little steps required to achieve a large goal. Each step enables the next.

Why did we pass that `iterator`; `setBooleanArg` cetainly doesn't need it. True, but `setIntArg` and `setStringArg` will! And since I want to deploy all three of these functions through an abstract method in `ArgumentMarshaller`, I need to pass it to `setBooleanArg`.

So, now `setBooleanArg` is useless. If there were a `set` function in `ArgumentMarshaler`, we could call it directly. So it's time to make that function! The first step is to add the new abstract method to `ArgumentMarshaler`.

```
  private abstract class ArgumentMarshaler {
    public abstract void set(Iterator<String> currentArgument)
                         throws ArgsException;
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
```

Of course this breaks all the derivatives. So let's implement the new method in each.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      booleanValue = true;
    }

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

```
private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

And now we can eliminate `setBooleanArg`!

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);

  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

The tests all pass, and the `set` function is deploying to `BooleanArgumentMarshaler`! Now we can do the same for `Strings` and `Integers`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
```

```
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof IntegerArgumentMarshaler)
      m.set(currentArgument);

  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

```
private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  public void set(String s) {
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
     String parameter = null;
     try {
       parameter = currentArgument.next();
       set(parameter);
     } catch (NoSuchElementException e) {
       errorCode = ErrorCode.MISSING_INTEGER;
       throw new ArgsException();
     } catch (ArgsException e) {
       errorParameter = parameter;
       errorCode = ErrorCode.INVALID_INTEGER;
       throw e;
     }
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

And so the coup-de-grace: The type-case can be removed! Touche!

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
}
```

Now we can get rid of some crufty functions in `IntegerArgumentMarshaler`, and clean it up a bit.

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

We can also turn `ArgumentMarshaler` into an interface.

```
private interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
  Object get();
}
```

So, now let's see how easy it is to add a new argument type to our structure. It should require very few changes, and those changes should be isolated.

First we begin by adding a new test case to check that the double argument works correctly.

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
  assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

First we clean up the schema parsing code and add the ## detection for the double argument type.

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
```

```
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
```

Next we write the `DoubleArgumentMarshaler` class.

```
  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
```

This forces us to add a new `ErrorCode`.

```
private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
MISSING_DOUBLE, INVALID_DOUBLE}
```

And we need a `getDouble` function.

```
  public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }
```

And all the tests pass! That was pretty painless. So now let's make sure all the error processing works correctly. The next test case checks that an error is declared if an unparseable string is fed to a ## argument.

```
  public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x","Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 args.errorMessage());
  }
```

```
  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }
```

And the tests pass.

The next test makes sure we detect a missing double argument properly.

```
  public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[]{"-x"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
                 args.errorMessage());
  }
```

This passes as expected. We wrote it simply for completeness.

The exception code is pretty ugly, and doesn't really belong in the `Args` class. We are also throwing `ParseException`, which doesn't really belong to us. So let's merge all the exceptions into a single `ArgsException` class, and move it into its own module.

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

```
public class Args {
  ...
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
```

```
  this.schema = schema;
  argsList = Arrays.asList(args);
  valid = parse();
}

private boolean parse() throws ArgsException {
  if (schema.length() == 0 && argsList.size() == 0)
    return true;
  parseSchema();
  try {
    parseArguments();
  } catch (ArgsException e) {
  }
  return valid;
}

private boolean parseSchema() throws ArgsException {
  ...
}

private void parseSchemaElement(String element) throws ArgsException {
  ...
  else
    throw new ArgsException(
      String.format("Argument: %c has invalid format: %s.",
                    elementId,elementTail));
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId)) {
    throw new ArgsException(
      "Bad character:" + elementId + "in Args format: " + schema);
  }
}

...

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

...

private class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  public Object get() {
    return stringValue;
```

```
    }
  }

  private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }

  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
}
```

This is nice. Now the only exception thrown by `Args` is `ArgsException`. Moving `ArgsException` into its own module means that we can move a lot of the miscellaneous error support code into that module and out of the `Args` module. It provides a natural and obvious place to put all that code, and will really help us clean up the `Args` module going forward.

It took about half an hour to make the following set of changes that completely separate the exception and error code from the `Args` module. This was achieved through a series of about 30 tiny steps (according to the local history that IntellJ keeps for me), keeping the tests passing between each step.

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
```

```
public void testCreateWithNoSchemaOrArguments() throws Exception {
  Args args = new Args("", new String[0]);
  assertEquals(0, args.cardinality());
}

public void testWithNoSchemaButWithOneArgument() throws Exception {
  try {
    new Args("", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                 e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testWithNoSchemaButWithMultipleArguments() throws Exception {
  try {
    new Args("", new String[]{"-x", "-y"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                 e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }

}

public void testNonLetterSchema() throws Exception {
  try {
    new Args("*", new String[]{});
    fail("Args constructor should have thrown exception");
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                 e.getErrorCode());
    assertEquals('*', e.getErrorArgumentId());
  }
}

public void testInvalidArgumentFormat() throws Exception {
  try {
    new Args("f~", new String[]{});
    fail("Args constructor should have throws exception");
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
    assertEquals('f', e.getErrorArgumentId());
  }
}

public void testSimpleBooleanPresent() throws Exception {
  Args args = new Args("x", new String[]{"-x"});
  assertEquals(1, args.cardinality());
  assertEquals(true, args.getBoolean('x'));
}

public void testSimpleStringPresent() throws Exception {
  Args args = new Args("x*", new String[]{"-x", "param"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals("param", args.getString('x'));
}

public void testMissingStringArgument() throws Exception {
```

```
  try {
    new Args("x*", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testSpacesInFormat() throws Exception {
  Args args = new Args("x, y", new String[]{"-xy"});
  assertEquals(2, args.cardinality());
  assertTrue(args.has('x'));
  assertTrue(args.has('y'));
}

public void testSimpleIntPresent() throws Exception {
  Args args = new Args("x#", new String[]{"-x", "42"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42, args.getInt('x'));
}

public void testInvalidInteger() throws Exception {
  try {
    new Args("x#", new String[]{"-x", "Forty two"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
    assertEquals("Forty two", e.getErrorParameter());
  }

}

public void testMissingInteger() throws Exception {
  try {
    new Args("x#", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[]{"-x", "42.3"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}

public void testInvalidDouble() throws Exception {
  try {
    new Args("x##", new String[]{"-x", "Forty two"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
    assertEquals("Forty two", e.getErrorParameter());
  }
}
```

```
  public void testMissingDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }
}
```

```
public class ArgsExceptionTest extends TestCase {
  public void testUnexpectedMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        'x', null);
    assertEquals("Argument -x unexpected.", e.errorMessage());
  }

  public void testMissingStringMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                        'x', null);
    assertEquals("Could not find string parameter for -x.", e.errorMessage());
  }

  public void testInvalidIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                        'x', "Forty two");
    assertEquals("Argument -x expects an integer but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
    assertEquals("Could not find integer parameter for -x.", e.errorMessage());
  }

  public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
  }
}
```

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }
```

```
public ArgsException(ErrorCode errorCode, String errorParameter) {
  this.errorCode = errorCode;
  this.errorParameter = errorParameter;
}

public ArgsException(ErrorCode errorCode, char errorArgumentId,
                     String errorParameter) {
  this.errorCode = errorCode;
  this.errorParameter = errorParameter;
  this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId() {
  return errorArgumentId;
}

public void setErrorArgumentId(char errorArgumentId) {
  this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
  return errorParameter;
}

public void setErrorParameter(String errorParameter) {
  this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode() {
  return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
  this.errorCode = errorCode;
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return String.format("Argument -%c unexpected.", errorArgumentId);
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
    case INVALID_DOUBLE:
      return String.format("Argument -%c expects a double but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_DOUBLE:
      return String.format("Could not find double parameter for -%c.",
                           errorArgumentId);
  }
  return "";
}

public enum ErrorCode {
```

```
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
  }

  private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                              elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                              elementId, null);
    }
  }

  private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }
  }
```

```
private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                            argChar, null);
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    e.setErrorArgumentId(argChar);
    throw e;
  }
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public boolean getBoolean(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}

public String getString(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
    return "";
  }
}
```

```
  public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }


  public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

The majority of the changes to the `Args` class were deletions. A lot of code just got moved out of `Args` and put into `ArgsException`. Nice. We also moved all the `ArgumentMarshallers` into their own files. Nicer!

Much of good software design is simply about partioning -- creating appropriate places to put different kinds of code. This separation of concerns makes the code much simpler to understand, and maintain.

Of special interest is the `errorMessage` method of `ArgsException`. Clearly it was a violation of the SRP to put the error message formatting into `Args`. `Args` should be about the processing of arguments, not about the format of the error messages. However, does it really make sense to put the error message formatting code into `ArgsException`?

Frankly, it's a compromise. Users who don't like the error messages supplied by `ArgsException` will have to write their own. But the convenience of having canned error messages already prepared for you is not insignificant.

By now it should be clear that we are within striking distance of the final solution that appeared at the start of this chapter. I'll leave the final transformations to you as an exercise.

Conclusion

It is not enough for code to work. Code that works is often badly broken. Programmers who satisfy themselves with merely working code are behaving unprofessionally. They may fear that they don't have time to improve the structure and design of their code; but I disagree. Nothing has a more profound and long-term degrading effect upon a development project than bad code. Bad schedules can be redone, bad requirements can be redefined. Bad team-dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down. Time and time again I have seen teams grind to a crawl because, in their haste, they created a malignant morass of code that forever thereafter dominated their destiny.

Of course bad code can be cleaned up. But it's very expensive. As code rots the modules insinuate themselves into each other, creating lots of hidden and tangled dependencies. Finding and breaking old dependencies is a long an arduous task. On the other hand, *keeping* code clean is relatively easy. If you make a mess in a module in the morning, it is easy to clean it up in the afternoon. Better yet, if you make a made a mess 5 minutes ago, it's *very* easy to clean it up right now.

So the solution is to continuously keep your code as clean and simple as it can be. Never let the rot get started.