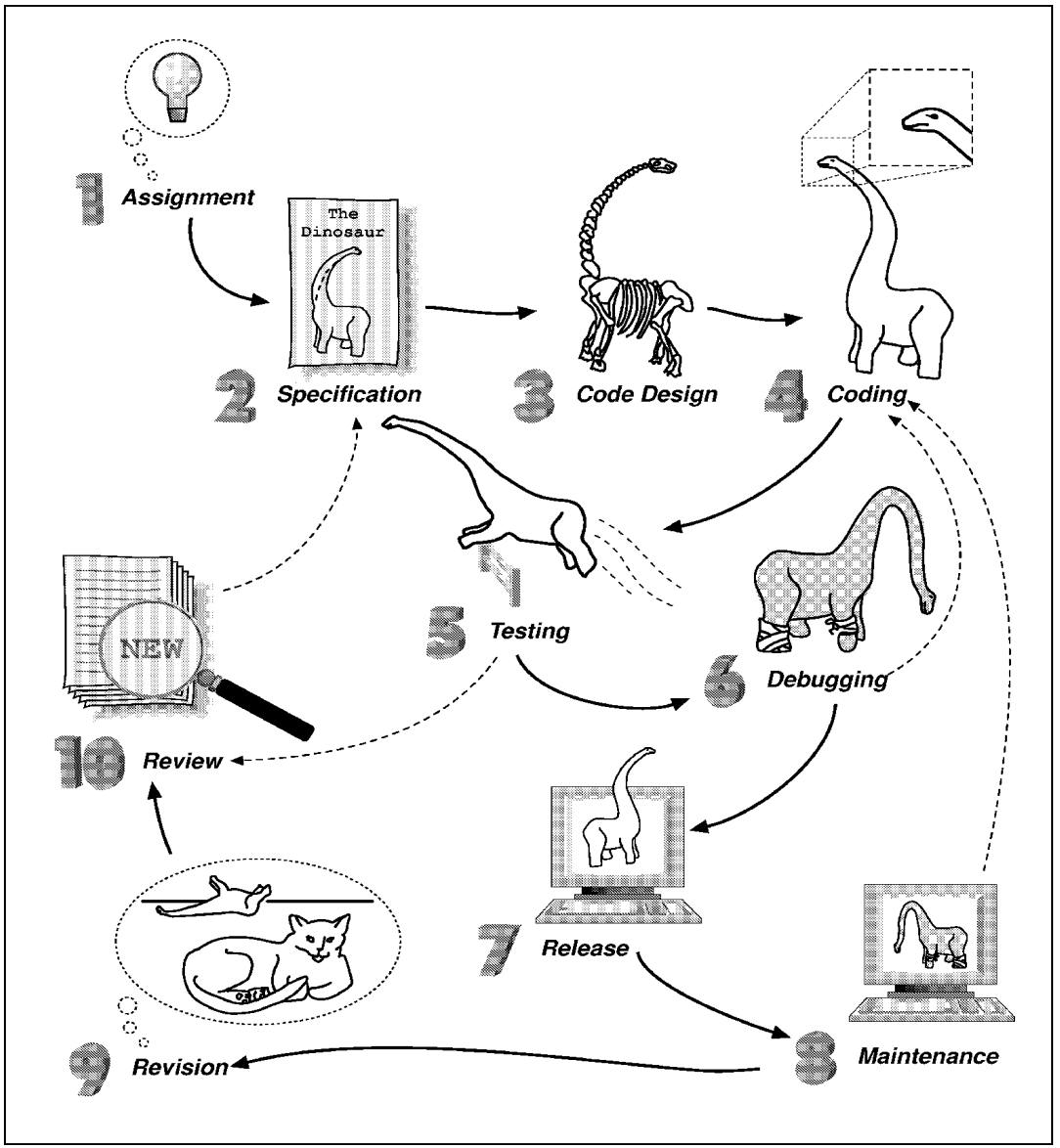
Chapter - 7
The Programming Process

The Programming Process

fig needs fixing ## Actually needs rewriting for this format



Setting Up

In general you want to put each program in a separate directory. To create a directory use the commands:

UNIX:

```
% cd ~
% mkdir calc
% cd calc
```

Microsoft Windows (Command Prompt window):

```
C:> cd \
C:> mkdir calc
C:> cd calc
```

Specification

Calc

A four-function calculator

Preliminary Specification

Dec. 10, 2002 Steve Oualline

Warning: This is a preliminary specification. Any resemblance to any software living or dead is purely coincidental.

Calc is a program that allows the user to turn his $10,000 computer into a $1.98 four-function calculator. The program adds, subtracts, multiplies and divides simple integers.

When the program is run, it zeros the result register and displays its content. The user can then type in an operator and number. The result is updated and displayed. The following operators are valid:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

Sample Use

```
calc
Result: 0
 Enter operator and number: + 123
Result: 123
 Enter operator and number: - 23
Result: 100
 Enter operator and number: / 25
Result: 4
 Enter operator and number: * 4
Result: 16
```

Code Design

Code design is the process of writing down a description of our program in a clear and easy to understand manner. Details may be omitted.

Frequently pseudo code is used for this purpose:

```
Loop
   Read an operator and number
   Do the calculation
   Display the result
End-Loop
```

The Prototype

The prototype code contains a small sub-set of the full program. It is the smallest sub-set that does anything. This allows us to test it before we write the full program.

Prototype

```
                }
            }

    }
```

The *Makefile*

The program *make* acts as the programmers assistant. When you type the command *make* the program looks for the file *Makefile*, reads a description of how to create the program and executes the necessary commands.

Makefile for UNIX generic CC compiler

```
#


#
CC=CC
CFLAGS=-g




clean:
```

Makefile for the GNU g++ compiler

```
#


#
CC=g++




clean:
```

Makefile for Borland-C++

```
#
# Makefile for Borland's Borland-C++ compiler
#
CC=bcc32
#
# Flags
#    -N  -- Check for stack overflow
#    -v  -- Enable debugging
#    -w  -- Turn on all warnings
#    -tWC -- Console application
#
CFLAGS=-N -v -w -tWC
all: calc.exe

calc.exe: calc.cpp
     $(CC) $(CFLAGS) -ecalc calc.cpp

clean:
     erase calc.exe
```

Makefile for Visual-C++ .NET

```
#
# Makefile for Microsoft Visual C++
#
CC=cl
# Flags
#     GZ - Enable stack checking
#     RTCsuc -- Enable all runtime checks
#     Zi -- Enable debugging
#     Wall -- Turn on warnings (Omitted)
#     EHsc -- Turn exceptions on
CFLAGS=/GZ /RTCsuc /Zi /EHsc
all: calc.exe

calc.exe: calc.cpp
     $(CC) $(CFLAGS)  calc.cpp

clean:
     erase calc.exe
```

Warning: The Visual C++ *make* utility is named *nmake*.

Testing

Once the program is compiled without errors, we can move on to the testing phase. Now is the time to start writing a test plan. This document is simply a list of the steps we perform to make sure the program works. It is written for two reasons.

- If a bug is found, we want to be able to reproduce it.
- If we ever change the program, we will want to re-test it to make sure new code did not break any of the sections of the program that were previously working.

Test Plan

Test plan:

```
Try the following operations

+ 123    Result should be 123
+ 52     Result should be 175
x 37     Error message should be output
```

Running the program we get:

```
Result: 0
 Enter operator and number: + 123
Result: 123
 Enter operator and number: + 52
Result: 175
 Enter operator and number: x 37
Result: 212
```

Debugging

One of the simplest ways of debugging is to put print statements in your program. We'll put one before the data goes bad (just to make sure it's good) and one after, to see what went wrong.

```
std::cout << "Enter operator and number: ";
std::cin >> value >> operator;

std::cout << "## after cin " <<operator << '\n';

if (operator = '+') {
    std::cout << "## after if " << operator << '\n';
    result += value;
```

Note: The ## is used to indicate that this is a debug line. It also makes it easier to remove all debugging statements when we're done.

Debug Output

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
## after cin +
## after if +
Result: 175
Enter operator and number: x 37
## after cin x
## after if +
Result: 212
```

You should now be able to spot the problem.

You were warned!

Remember when we were discussing = vs. = =.

I told you then that this is a very common error and you *will* make it. The reason we go on and on about it here is so that you will be aware of it and able to fix it when it does occur.

Finished Program

```
main()
{
```

```
            break;
```

Finished Program (cont.)

```
            }
        }

    }
```

Finished Test Plan

We expand our test plan to include the new operators and try it again.

```
+ 123     Result should be 123
+ 52      Result should be 175
x 37      Error message should be output
- 175     Result should be zero
+ 10      Result should be 10
/ 5       Result should be 2
/ 0       Divide by zero error
* 8       Result should be 16
q         Program should exit
```

Maintenance and Revisions

No matter how much testing is done on a program the user can always find at least one more bug. During the maintenance phase, these bugs are found and removed.

Revisions

No matter how complete a program, the user will want one more feature. So you revise the specifications, add the change to the program, update the test plan, test the program and release it again.

Electronic Archeology

The art of going through someone else's code to discover amazing things
(like how and why the code works).

Contrary to popular belief, most C++ programs are not written by dyslexic orangutans using Zen programming techniques, and poorly commented in Swahili. They just look that way.

Ode to a maintenance programmer

Once more I travel that lone dark road
into someone else's impossible code
Through "if" and "switch" and "do" and "while"
that twist and turn for mile and mile
Clever code full of traps and tricks
and you must discover how it ticks
And then I emerge to ask a new,
"What the heck does this program do?"

Archaeological tools

- Editor (browser)
- Cross referencer
- grep
- indention tools
- pretty printers
- call graphs
- debuggers

Techniques

- Mark up the program (several colored pens are useful)
- Go through and comment the code
- Change the short variables to long ones
- Add comments

```
int state;// Controls some sort of state machine
int rmxy; // Something to do with color correction?
int idn;  / ???
```

A far Too Typical Program

```
int main(){

                ++c;

            }

        }

    }
```

A Better Version

```
/**************************************************************************
```

```
**************************************************************************/
```

A Better Version (cont.)

e

A Better Version (cont.)

```
int main()
{

        /*

         */
```

```
            ++guess_count;
```

A Better Version (cont.)

```
                    break;


                else

            }

        }

    }
```