

Currency Converter Application Implementation

Glenn Booker; August 12, 2003

Introduction

This document shows how the concepts of object oriented design are implemented for a simple application, and modeled with standard UML diagrams.

This example was developed on the Macintosh OS X operating system using the Objective C programming language and the Cocoa framework, so the terminology is slightly different from other C variants. Nevertheless, the main concepts still apply, and most of the syntax is similar.

The user interface was developed using the Interface Builder tool, and the rest of the application was developed with the Project Builder tool, both [Apple](#) products. The currency converter example was taken from the book *Learning Cocoa* (O'Reilly, (ISBN [0596001606](#))).

Application Objective

This application is a currency converter, which takes the currency conversion (exchange) rate and the amount of US dollars to be converted to determine the amount of the other currency.

The final (and sole) user interface is shown in Figure 1. The first two text fields are entered manually by the User of the application, who then clicks on the Convert button to have the total Amount in Other Currency calculated.

Figure 1. Currency Converter final product



Requirements

Requirements for this application are minimal.

- Presumably, the data entered into the Exchange Rate and Dollars to Convert fields will be either integer or real (floating point) values.
- Clicking on the Convert button fills the answer in the Amount in Other Currency field.

Additional requirements would n't show up directly in the class, such as:

- Want compliance with Macintosh interface design guidance
- Could later add validation of the data inputs (e.g. what happens if the user doesn't enter numbers?)
- Tabbing should alternate between the first two text fields.
- The output field Amount in Other Currency should not be editable.
- After calculating the result, the Dollars to Convert field should be highlighted.

Use Case

This application only has one use case, which we'll call Convert Currency.

- Use Case Name: Convert Currency
Primary Actor: "User", any person who uses the application.
Secondary Actors: None
Preconditions:
1) Application has already been launched.
2) User knows exchange rate and dollar amount to be converted.
Main Success Scenario:
1) User enters conversion rate and dollar amount to be converted.
2) User commands conversion to be calculated.
3) System displays the converted amount.
Postcondition:
1) User knows the converted amount.

Design

In keeping with the BCE (Boundary, Control, Entity) pattern for design, we will assign the names shown in Table 1 to the classes needed for this application.

Table 1. Class Names

Type of Class	Class Name
Boundary (or View or Interface)	MainMenu
Control (Controller)	ConverterController
Entity (or Data or Model)	Converter

The corresponding application class diagram is shown in Figure 2, and the sequence diagram using these classes is shown in Figure 3. The data type “float(idl)” means that the attributes shown are all floating point (real number) valued, using a generic form of C (as opposed to C++, C#, or Visual Basic forms of floating point numbers).

Figure 2. Application Class Diagram for Currency Converter

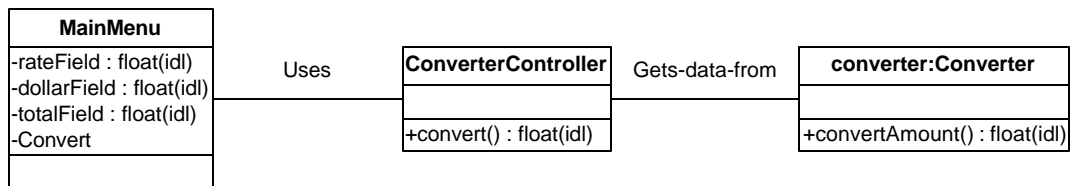
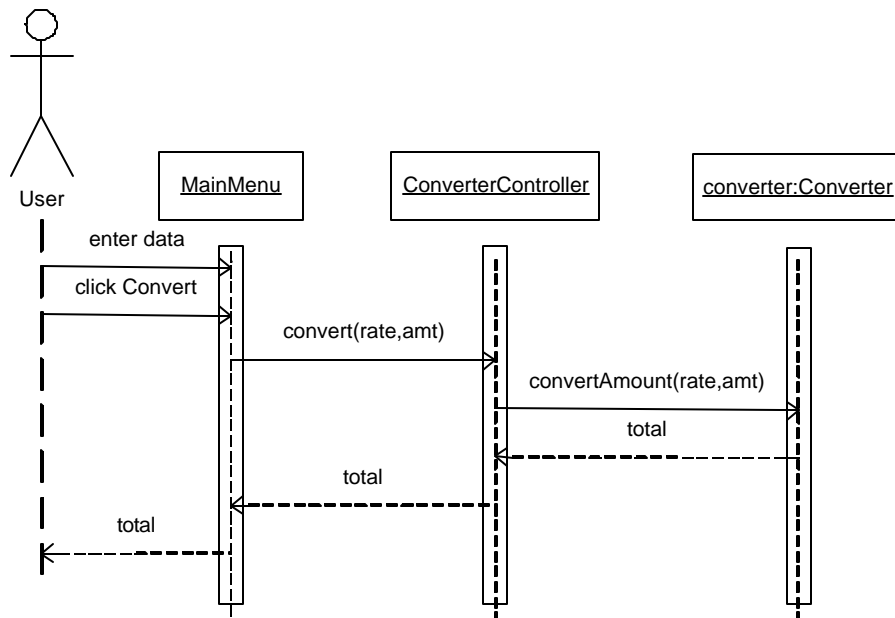


Figure 3. Sequence Diagram for Currency Converter



Note in Figure 2 that the object instantiated from class Converter is called “converter”. Unfortunately, the objects for the MainMenu and ConverterController classes are also

called MainMenu and ConverterController, respectively, so they aren't shown in the class diagram.

In the MainMenu class, the attribute called Convert represents the button labeled Convert; it could be considered a non-primitive data type.

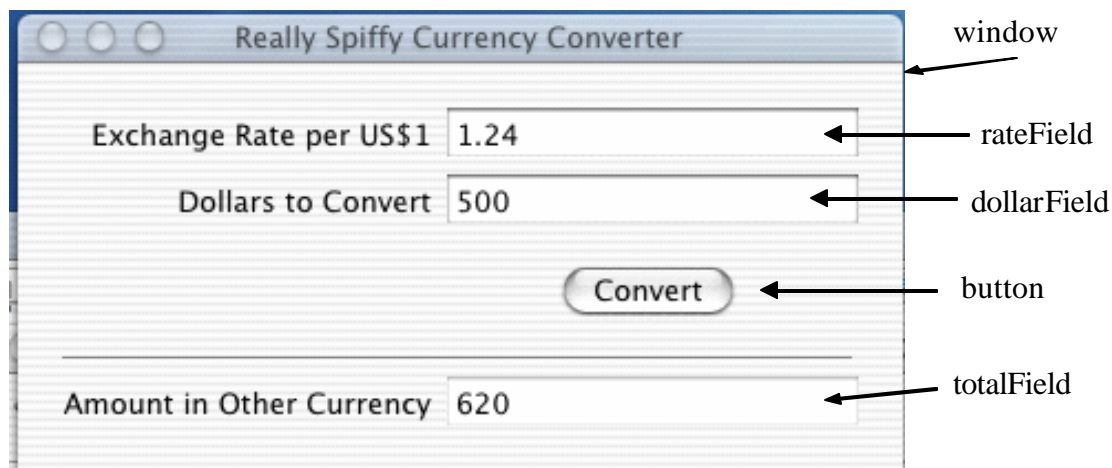
In this case, we could safely leave all objects in place for the life of the application, so we don't have to deal with creating objects except at application launch and exit (which are outside of the use case being modeled, and are handled automatically).

Interface Implementation

The MainMenu class is implemented as the graphic set of objects seen in Figure 4. Within the overall window (and its buttons in the upper left corner, and the title bar reading "Really Spiffy Currency Converter"), there are four objects of interest to the application functionality:

- 1) The first text field is called rateField
- 2) The second text field is called dollarField
- 3) The button is labeled Convert
- 4) And below the line is the output, totalField.

Figure 4. MainMenu Implementation



In addition, the three labels (e.g. "Exchange Rate per US\$1") are each text objects, and the thin line below the button is also an object. Since these objects are static parts of the window, we don't care what they are called.

The MainMenu interface was created using visual guidelines to ensure that the interface uses correct spacing between objects, correct spacing from the edges of the window, standard font sizes, etc., as defined by the [Aqua Human Interface Guidelines](#).

Many characteristics are defined in the interface, including the following:

- When the application is launched, the cursor is put in the rateField automatically (here, using the application window's event called initialFirstResponder).
- The tabs were set to alternate between rateField and dollarField (through an object characteristic called nextKeyView).
- If the user presses Enter or Return, the Convert button is assumed to have been clicked (done with the Equivalence attribute of 'Return' for the button).
- A target of the Convert button is defined to be the ConverterController class.

Source Code

Each controller or entity class is implemented as two files, *classname.h* and *classname.m*. (Note that in other C variants, the latter file is often suffixed *filename.c* or *filename.cpp*.) The file ending with ".h" is the header file; it declares the class and its methods (if any). The file ending with ".m" is the body of the class; it contains the code which makes the class do something.

Comments in source code appear between "/*" and "*/" like this:

```
/* this is a comment */
```

Table 2 contains the source code for Converter.h.

- The import statement invokes the Cocoa framework, which is the basis for all pre-existing objects in this development environment.
- The @interface statement defines Converter as being a subclass of NSObject. In this environment, the highest level class is NSObject.
- The (float) statement declares that Converter has a method called convertAmount, which uses two float parameters, amt and rate.

Table 2. Converter.h source code

```
/* Converter */  
  
#import <Cocoa/Cocoa.h>  
  
@interface Converter : NSObject  
{  
}  
- (float)convertAmount:(float)amt atRate:(float)rate;  
  
@end
```

Table 3 contains the body of the converter class, Converter.m.

- The import statement tells to look in Converter.h for the header information.

- The @implementation and @end statements bracket the implementation of the code for this class.
- The (float) statement indicates we're going to define the source code for the convertAmount method.
- The return statement creates the return message with the value of the dollar amount times the conversion rate – the desired output from the application.

Table 3. Converter.m source code

```
#import "Converter.h"

@implementation Converter
- (float)convertAmount:(float)amt atRate:(float)rate
{
    return (amt*rate);
}
@end
```

Table 4 has the source code for the ConverterController header.

- The @interface statement defines ConverterController as a subclass of NSObject.
- The four IBOutlet statements connect this class to the 'converter' object, and to the three text fields in MainMenu. Recall that these four things are all objects, so these statements are establishing visibility between this class and the boundary and entity objects.
- The IBAction statement means that this class implements a method called 'convert'.

Table 4. ConverterController.h source code

```
/* ConverterController */

#import <Cocoa/Cocoa.h>

@interface ConverterController : NSObject
{
    IBOutlet id converter;
    IBOutlet id dollarField;
    IBOutlet id rateField;
    IBOutlet id totalField;
}
- (IBAction)convert:(id)sender;
@end
```

Table 5 has the source code for the ConverterController body.

- The first import statement connects this file to its header file.
- The second import statement connects this object to the Converter class.
- The @implementation and @end lines bracket the body of this class.
- The IBAction statement indicates we're defining the method called 'convert'.
- The float statement defines floating point local variables called rate, amt, and total.
- The amt statement reads the value of dollarField on MainMenu, and puts that value in the local variable amt.
- The rate statement does the same thing for the rateField text object.
- The value of the total variable is then set by calling the converter object's convertAmount method, with amt and rate as parameters.
- The totalField statement then fills the totalField text object with the value of total.
- The rateField statement highlights the rateField text object, based on the assumption that this is the most likely field the User would want to change next.

Table 5. ConverterController.m source code

```
#import "ConverterController.h"
#import "Converter.h"

@implementation ConverterController

- (IBAction)convert:(id)sender
{
    float rate, amt, total;

    amt = [dollarField floatValue];
    rate = [rateField floatValue];

    total = [converter convertAmount:amt atRate:rate];

    [totalField setFloatValue: total];
    [rateField selectText:self];
}
@end
```

Table 6 defines the source code for the "main" module for this application. As with most C language variants, the application automatically looks for a function called 'main' at launch.

In this case, the 'main' program declares the main program (with the line starting 'int main'), then runs the event called NSApplicationMain. In the Cocoa environment, NSApplicationMain is a mandatory object which runs and supervises the application.

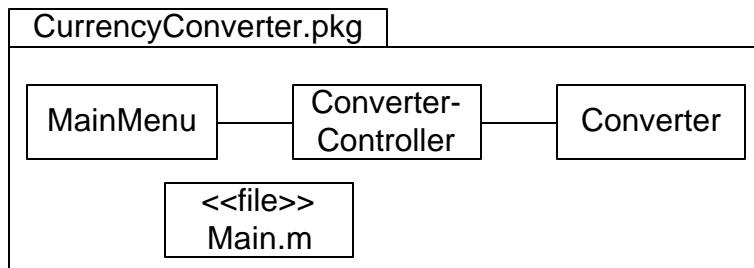
Table 6. Main.m source code

```
//  
// main.m  
// CurrencyConverter  
//  
// Created by Glenn Booker on Sat Jul 19 2003.  
// Copyright (c) 2003 __MyCompanyName__. All rights  
reserved.  
//  
  
#import <Cocoa/Cocoa.h>  
  
int main(int argc, const char *argv[])  
{  
    return NSApplicationMain(argc, argv);  
}
```

Packaging

This application consists of three classes plus the main.m source code, so it would easily and logically all fit in one Package, as shown in Figure 5.

Figure 5. Package Diagram for Currency Converter



Summary

This document has described how a simple currency converter application is designed and implemented using UML notation and object oriented concepts.