OOPSLA 2006 Educator's Symposium: Nifty Assignment

Temperature Calculator

Programming for Change

Dung "Zung" Nguyen, Mathias Ricken

Rice University, Houston, TX USA

Website with Applets and Updated Materials

A version of this document with working applets and updated materials is available at:

http://www.cs.rice.edu/~mgricken/research/tempcalc/website

Description

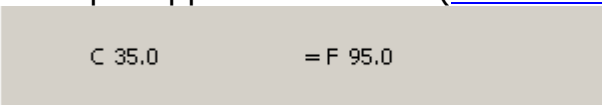
What is programming for change? The development of any piece of software begins with a set of specifications describing in some detail the problems to be solved. Almost invariably, the specifications fluctuate and change during the course of development of the software. Trying to anticipate all possible changes and write a program that does all and covers all is a futile undertaking. However, it is not unreasonable to expect and even demand that programs be written in such a way that an "epsilon" (read "small") change in the specifications will only necessitate a "delta" (read "manageable") change in code. Though "small" and "manageable" are software characteristics with subjective interpretations based on differing software engineering metrics, at some basic level they can be expressed in terms of lines of code and "big Oh" analysis.

Programming for change is a continual process in which a software system is designed and re-designed over many iterations to capture the essence of the problems at hand and express it in code. At the heart of this process is the incessant effort to identify those elements that can vary —the *variants*— and delineate them from those that do not —the *invariants*. A properly designed software system should strive to decouple the variants from the invariants in

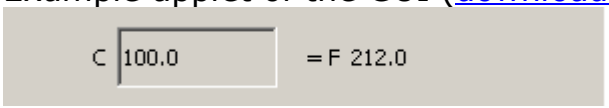
order to facilitate the re-use of the invariants and allow modifications to the variants with minimal perturbation to the existing code.

To illustrate the importance of programing for change to the students, we guide them through the development of a program that converts temperature measurements from one unit to another. The programming assignment consists of a series of small exercises, each of which imposes a small change in the requirements and forces some appropriate modification on the implementation code. To promote code re-use, we apply the Janus Principle ([1]) and require that the program at each phase must be written in such a way that it can support at least two distinct user interfaces: a GUI interface and a command line interface. For certain specification changes, we ask students to identify the variants and the invariants and make appropriate modifications to the code. In many situations, we require the students to modify their code in more than one way and discuss the pros and cons. The programming assignment is summarized in the following.

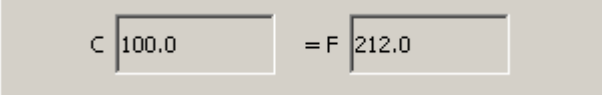
- **Preliminary step**: Write a program to convert 35°C to Fahrenheit.
  - We provide the student the conversion formula and what we call the "constant solution" in GUI form and command line form.
  - Example applet of the GUI (download solution):



- **Exercise 1**: Write a program to convert any temperature measured in Celsius to Fahrenheit.
  - Here we want the students to write a static method as a direct translation of the conversion formula from Celsius to Fahrenheit.
  - Example applet of the GUI (download solution):

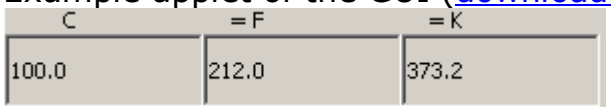


- **Exercise 2**: Write a program to convert temperatures between Celsius and Fahrenheit in both directions.
  - Here we require the students to add another static method for the conversion from Fahrenheit to Celsius. This way of modifying the model to comply with the change in the problem specification will call for numerous changes in the GUI and in the command line interface.
  - Example applet of the GUI (download solution):



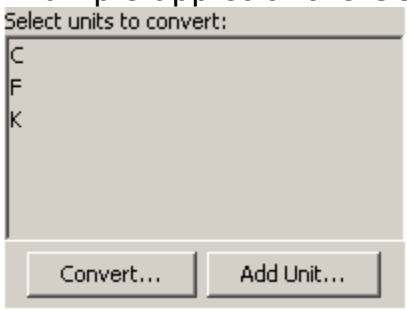
- **Exercise 3**: Write a program to convert temperatures between Celsius, Fahrenheit and Kelvin in all directions.

    - Again we ask the students to add more static methods here as done in the preceding exercise. The students should experience something "bad" here: an epsilon change in the specification is engendering a large change in code with the current design.
    - Example applet of the GUI (download solution):

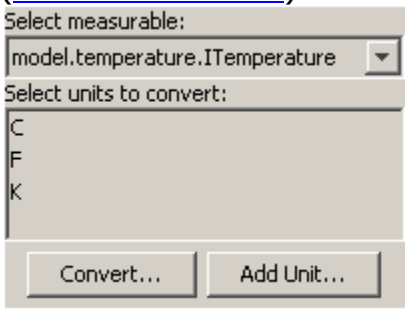


- **Exercise 4**: Discuss the change in complexity between the solutions to exercises 2 and 3. Derive a formula that expresses the complexity of a similar program involving N different temperature scales.
    - We make the students do some informal analysis here. We want them to stop writing code and think.
- **Exercise 5**: Write a program converting Celsius, Fahrenheit and Kelvin in all directions by viewing the N different temperature scales as nodes of a star-shaped graph. Discuss the complexity.
    - By changing the model to reflect the notion of a star-shaped graph, the students should experience at this point that an epsilon change in the specification only induces a delta change in the code for the conversion. Still the code for the user interfaces has to be modified more than desired. Moreover, most of the existing code has to be modified. The nagging question is whether or not there is a design that will allow us to add *minimal* code to the system *without* touching *any* of the *existing* code. This leads to the next exercise that guides students to capture the essence of the problem and capture its abstraction with appropriate concrete and abstract classes.
    - The program looks the same as the one from Exercise 3, but its implementation is different (download solution)
- **Exercise 6**: Complete the provided stub code that implements the following object model for the conversion problem.
    - A conversion function is an abstract bijection (i.e. a function with an inverse) from a subset of real numbers to another subset of real numbers
    - A measurement has a unit and a real value.
    - A unit is an abstract notion. It has a symbol. It has the abstract capability to provide a conversion function to convert from itself to Celsius, a concrete unit. It knows concretely how to convert from one measurement to another measurement.
    - Fahrenheit is a concrete unit.
    - Kelvin is a concrete unit.

- Example applet of the GUI (download solution):



- While we do not require this from our students, the demonstration program above also demonstrates the use of Java reflection to load classes at runtime. This demonstrates that the design is flexible enough to deal with temperature scales that were not incorporated right away. Try clicking on the "Add Unit..." button and enter "model.temperature.Reaumur" to load the Reaumur temperature scale at runtime.
- We also show our students that this design is not limited to converting temperatures, but that it can be applied to any unit conversion problem. The category of the units (temperature, length, volume, etc.) doesn't even have to be known, and new categories can be added at runtime. This introduces new challenges, like preventing conversion of liters to inches, but they can be solved too, as the demo applet below proves (download source).



The categories and units available are:

  - Category: model.temperature.ITemperature; Units: model.temperature.Celsius/Fahrenheit/Kelvin/Reaumur
  - Category: model.length.ILength; Units: model.length.Meter/Mile/Yard
  - Category: model.volume; Units: model.volume.Gallon/Liter/Pint

These are the only units available in this demo, but if you run the demo from the command line, you can add any class accessible on the classpath.

- We provide the students with stub code to ensure that they will do exactly what we ask them to do. The students will come to realize that should they need to add another temperature unit such as Reaumur, they only need to subclass the abstract unit class and implement only one method: the concrete factory method that provides the conversion from Reaumur to Celsius and back. Nothing else in the existing code needs to be changed. With a simple application of Java reflection, students could dynamically load *any* concrete unit class at run-time *without* recompiling any of the existing code.
- In summary, to be able to program for change, one should be able to capture the essence of the problem, separate the variants from the invariants and program to the highest level of abstraction.

What makes it so nifty?

- Teaches students about object-oriented design on a system that is rich enough to involve a multitude of design issues but small enough to be manageable by introductory level students. The system the students work on is familiar to the students and one to which they can relate.
- Forces students to consider flexibility, extensibility and robustness and apply appropriate design patterns to achieve in these design goals.
- Help students develop a sense and an appreciation for good designs by forcing them to write both good and bad designs and compare them against each other.
- Requires students to integrate a broad spectrum of skills and concepts (see skills list below).
- Focuses on the thought processes involved with the design of OO systems.
- Has an open-ended number of design solutions, each with its own trade-offs and pros and cons, that serve as means to stimulate the students' critical thinking.
- Channels the students down a path that fosters good object-oriented design by setting requirements that cut off the vast majority of inappropriate design possibilities.
- Reinforces the concept of an abstract function as a model for an abstract computation.

Target audience

This project is intended for students in the second semester of an objects-first curriculum where they have already seen polymorphism, design patterns and component framework systems (see prerequisites below).

Ideas and Skills Involved

The Temperature Calculator is not really about converting temperature values to different units. Instead, it teaches students some of the bigger concepts in software engineering, such as:

- Striving to capture appropriate abstractions and delineate the invariants from the variants.
- Creating loose and abstract coupling between cooperating objects in the system to achieve correctness, robustness, security, flexibility, and extensibility.
- Using and building components and frameworks.
- Using anonymous inner classes to instantiate objects on-the-fly, especially in the context of commands.
- Using the closure properties of inner classes to directly access both instance and final local variables.
- Numerous compelling demonstrations of polymorphic behavior, especially those that are difficult to replicate with conditional statements.

Length of time students typically work on it

We have a 1.5 hour lab to help students get started on the assignment. We expect students to spend from 6 to 8 hours on the project. We give them a week to complete the project.

Prerequisite material

This project assumes the following material has been covered in class:

- Javadoc and UML diagrams
- Abstract structure and behavior
- Polymorphism
- Using both inheritance and composition as means of extending functionality
- Delegation model programming
- Design patterns, particularly MVC, factory method, template method and command
- Closure and anonymous inner classes
- Developing simple GUIs with Swing.
- Rudiments of complexity analysis using big Oh notation.
- Rudiments of Java generics.

It is important that this assignment be given in the context of a comprehensive instruction on object-oriented programming that stresses abstract decomposition.

Difficulties to watch for

- Understanding the modeling of behaviors as objects so that they can be passed to other objects and executed later.
- Excessive hard-coding of behaviors
- Tightly coupled implementations
- Console I/O in Java
- Integer arithmetic vs. floating point arithmetic

Instructional Materials

- Assignment
- Sample instructor's solutions

Reference

1. J. Adams, OOP and the Janus Principle, *Proceedings of the 37th SIGCSE*, March 2006, 359-363.

OOPSLA 2006 Educator's Symposium: Nifty Assignment

Temperature Calculator

Programming for Change

Dung "Zung" Nguyen, Mathias Ricken

Rice University, Houston, TX USA

Provided Stub Code

Click here to download the provided stub code that is to be used in the following programming assignment.

The Initial Problem

Your cousin Pierre has invited you to spend two weeks with his family in Paris, France. Before you leave, however, he warns you: "It's 35 degrees here!" Remembering that France measures temperatures in degrees Celsius, you frown... Does Pierre want you to pack your winter coat or your bathing suit? To figure that out, you need to convert 35 degrees Celsius to Fahrenheit.

The Constant Solution

Of course, we all know how to solve this problem. Simple algebra is sufficient, we may just have to look up the formula before we can apply it: F = (C / 5) * 9 + 32. In this case, C = 35, so the solution is

F = (35 / 5) * 9 + 32 = 7 * 9 + 32 = 63 + 32 = 95

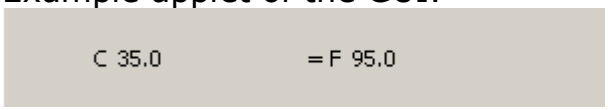
35 degrees Celsius are equivalent to 95 degrees Fahrenheit. Pack the bathing suit.

By performing this little calculation, we have solved our simple problem: We know how hot it is in Paris and can pack accordingly. Since we have solved

the problem for just one value, the 35 degrees Celsius, this solution is called a "constant solution". The good thing about this program is that it is simple, easy to understand, and most importantly it correctly solves the original problem. However, if we change slightly the problem to converting 30 degrees Celsius to Fahrenheit, the program no longer is applicable. It is correct, but not flexible.

In this assignment, you'll be changing the problem slightly to see how design choices affect the complexity of the model. To ensure that your model and view are decoupled and can thus be changed independently from each other, you'll also write two user interfaces (views) for each model: A graphical user interface (GUI) and a text interface executing in the console. We have provided the code for the constant solution to get you started (click link to execute example or download source).

Example applet of the GUI:



The Variable Solution

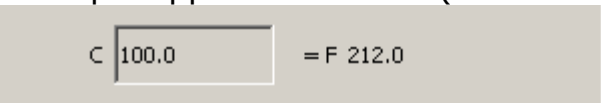
Intuitively, the problem doesn't really change when we change the temperature in Celsius to 30 degrees. We should still be able to solve it using the same means; the general problem is about converting any temperature measured in the Celsius scale to a value in degrees Fahrenheit. We should be able to change the temperature without necessitating a change in the code. Since the temperature can vary, this solution is called a "variable solution".

Exercise 1a: Identify the variant and the invariant of converting a temperature from Celsius to Fahrenheit.

Exercise 1b: Draw a UML diagram of the model. It should consist of a single static method.

Exercise 1c: Write a program that allows the user to convert *any* temperature in Celsius to its corresponding value in Fahrenheit. You can start by modifying the code for the constant solution, but remember to provide the two user interfaces! Watch out for integer arithmetic being carried out instead of floating point arithmetic.

Example applet of the GUI (reverse engineering is prohibited!):



Converting from Celsius to Fahrenheit and Back

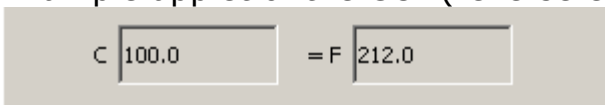
The solution you developed for Exercise 1 separated the variant from the invariants, and we can now convert any temperature in Celsius to the Fahrenheit scale. One shortcoming should be immediately apparent, though, especially when you look at your graphical user interface: You can only convert from Celsius to Fahrenheit, but not the other way! By using algebra and solving the above formula for C, we can derive a formula to convert degrees Fahrenheit to degrees Celsius:

C = 5/9 * (F - 32)

Exercise 2a: Draw a UML diagram of a changed model that consists of two static methods and allows conversion in both directions.

Exercise 2b: Write a program that implements this design. Again, provide two user interfaces.

Example applet of the GUI (reverse engineering is prohibited!):



Adding More Temperature Scales

The program developed for Exercise 2 allows the user to convert any temperature in either of the two temperature scales, Fahrenheit or Celsius, to the corresponding value in the other scale. Unfortunately, these aren't the only temperature scales in use: In many sciences, temperatures are measured in Kelvin, a scale whose origin is "absolute zero", the lowest possible temperature. Here is a formula to put Kelvin in relation to the Celsius scale:

K = C + 273.2

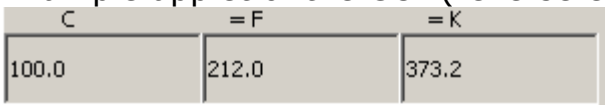
Your next program should support conversions in any direction between Celsius, Fahrenheit and Kelvin.

Exercise 3a: If you follow the pattern from the first two solutions of using one static method per possible conversion, how many methods will you need for this model?

Exercise 3b: Draw the UML diagram of the changed model, with one static method for each possible conversion.

Exercise 3c: Write a program that implements this design. Again, provide two user interfaces.

Example applet of the GUI (reverse engineering is prohibited!):



Analysis of our Approach

You are probably getting annoyed by now, so let's not write more code for a moment and just analyze our approach so far. Your program now supports conversion between three temperature scales in any direction, for any input value.

Exercise 4a: Compare the model from Exercise 2 to the model from Exercise 3. What is the ration between the number of methods in the models of Exercise 2 and Exercise 3?

There are even more temperature scales. One that has fallen out of use in the 20th century is the Reaumur scale developed by the French naturalist Rene-Antoine Ferchault de Reaumur, with the freezing point of water at its origin (just like Celsius) and the boiling point of water at 80.

Exercise 4b: Assume we added full support for the Reaumur scale to our program. Using our current approach of one method per possible conversion, how many methods will you need?

Exercise 4c: Using your answers to Exercises 3a, 4a and 4b, derive a formula - *f* - that allows you to calculate the number of methods, given the number of temperature scales - n: *f(n)* =

Exercise 4d: What is the complexity (i.e. the number of methods) of this approach, expressed in "big O" notation?

A Better Approach

As you can tell from your answer to Exercise 4c, the number of methods with our current approach grows very quickly. Even if you add just one more scale, you will have to write many more methods - two for each of the scales that were already there. If you had 100 scales and added a 101st, you would have to write 200 additional methods! There has to be a better way.

Let's reconsider what we are doing: We are converting temperatures, and even though we can express the temperature in different scales, resulting in different values, the meaning of that value is still the same: 100 degrees Celsius is the same as 212 degrees Fahrenheit, which is the same as 373.2 Kelvin. That means that we do not have to be able to convert from Fahrenheit to Kelvin directly, we can convert from Fahrenheit to Celsius first and then convert from Celsius to Kelvin! As long as we can somehow, using the conversion formulas we have, get from each scale to each other scale, our program can still perform conversions between arbitrary scales.

Exercise 5a: Using this new approach, what is the minimum number of methods we need to convert between three scales? Four scales? *n* scales?

Exercise 5b: What is the complexity (i.e. the number of methods) of this new approach, expressed in "big O" notation?

If we draw a directed graph (a "web" with "dots" and "arrows" between the dots) of this minimal arrangement with the temperature scales as nodes ("dots") and the conversion methods as edges ("arrows"), the graph is star-shaped: There is one node in the middle, all other nodes are arranged around it, and there is an arrow going from the central node to each of the outer nodes, and an arrow going from each of the outer nodes back to the central node.

Exercise 5c: Draw a UML diagram of the model using the new approach. The model should have the minimum number of methods to still perform arbitrary conversions between Celsius, Fahrenheit, and Kelvin.

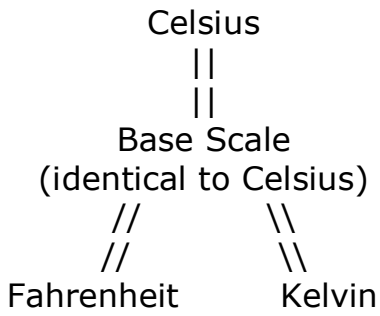
Exercise 5d: Write a program that implements this design. Again, provide two user interfaces; they should look the same as in the program for Exercise 3c.

Separating Variants and Invariants Again

Our new approach lets us add new temperature scales by adding only a small, constant number of methods. We say that an "epsilon" (i.e. small)

change in the specification will only cause a "delta" (i.e. proportional or manageable) change in the code. Theoretically, we can quite easily add an arbitrary number of scales. Can we come up with a design that allows us to do that without touching existing code *at all*? To do that, we need to separate variants from invariants again.

Before doing that, we are going to make one change to our graph of temperature scales: Let's put Celsius both in the center of the graph *and* on the outside. Even though the scale in the center is a Celsius scale, from now on we will call it "base scale". It may look like we just made our life more complicated, since we now have four nodes instead of just three, but this is actually a simplification: Before Celsius was treated differently because it was in the center; now Celsius is on the outside as well, and all scales are treated the same. This is what the graph looks like now:



To be more precise, let's give our conversion functions names. These are the six conversion functions our program will be able to handle:

CF(x) - convert *x* from Celsius to Fahrenheit
CK(x) - convert *x* from Celsius to Kelvin
FC(x) - convert *x* from Fahrenheit to Celsius
FK(x) - convert *x* from Fahrenheit to Kelvin
KC(x) - convert *x* from Kelvin to Celsius
KF(x) - convert *x* from Kelvin to Fahrenheit

And these are the six conversion functions we actually write:

CB(x) - convert *x* from Celsius to base scale
BC(x) - convert *x* from base scale to Celsius
FB(x) - convert *x* from Fahrenheit to base scale
BF(x) - convert *x* from base scale to Fahrenheit
KB(x) - convert *x* from Kelvin to base scale

BK(x) - convert *x* from base scale to Kelvin

Since the base scale is identical to the Celsius scale, the functions *CB* and *BC* are just the identity function: *CB(x) = BC(x) = x*. We can now express the six functions in the first group using the six functions in the second group:

CF(x) = BF(CB(x)) *CK(x) = BK(CB(x))*
FC(x) = BC(FB(x)) *FK(x) = BK(FB(x))*
KC(x) = BC(KB(x)) *KF(x) = BF(KB(x))*

To convert from one temperature scale to another, we first apply the input value to the function that takes us to the base scale, and then apply the result to a second function, the one that takes us from the base scale to the target scale:

1. To convert from Fahrenheit to Kelvin, we first apply *FB* to *x*; then we apply *BK* to the result of the first application.
2. To convert from Celsius to Fahrenheit, we first apply *CB* to *x*; then we apply *BF* to the result of the first application.
3. To convert from Fahrenheit to Celsius, we first apply *FB* to *x*; then we apply *BC* to the result of the first application.

Or abstractly:

4. To convert from a source scale "s" to a target scale "t", we first apply *sB* to *x*; then we apply *Bt* to the result of the first application.

Exercise 6a: What differs and what always remains the same when performing conversions this way? Identify the variants and the invariants. Remember that we can pass functions as data in an object-oriented language like Java.

If we want to add Reaumur as a fourth scale to our model, all we need to do is provide two additional functions:

RB(x) - convert *x* from Reaumur to base scale
BR(x) - convert *x* from base scale to Reaumur

These two functions, together with the six others, give us the ability to handle the six conversions from and to Reaumur:

CR(x) = BR(CB(x)) FR(x) = BR(FB(x)) KR(x) = BR(KB(x))

RC(x) = BC(RB(x)) RF(x) = BF(RB(x)) RK(x) = BK(RB(x))

Exercise 6b: What three pieces of data are necessary to define a temperature scale in our model? Hint: These are the invariants you identified earlier.

In functional languages, a function that is passed as data so it can later be applied is called a lambda; in object-oriented languages, it is also called the command design pattern. For this exercise, you can implement the *ILambda* interface for this purpose.

```
public interface ILambda<R, P> {
	public abstract R apply(P param);
}
```

It is interesting to note that the two functions that describe a temperature scale (e.g. *RB* and *BR* for the Reaumur scale) are not just any functions: They are inverses of each other, i.e. *BR(RB(x)) = x*. A function that has an inverse is called a bijection. We have therefore provided an *IBijection* interface that extends *ILambda* and is able to provide its own inverse, which is also an *IBijection*.

```
public interface IBijection<R, P> extends ILambda<R, P> {
	public abstract IBijection<P, R> getInverse();
}
```

Since an *IBijection* can provide its inverse, one *IBijection* instance can be used to represent both the conversion function to the base scale and the function from the base scale back. The class that represents an abstract temperature scale, AUnit, therefore needs only two fields for the variants, and one method expressing the invariant.

Just like in physics, you need to keep track of both a number and a unit (i.e. the temperature scale). To make sure the correct temperature scale remains associated with the number, it is a good idea to introduce a class to represent this number-unit pair. We have provided the Measurement class for this purpose.

Exercise 6c: Using the *IBijection* interface and the AUnit and Measurement classes, draw a UML diagram of the model with the Celsius, Fahrenheit, Kelvin, and Reaumur temperature scales.

Exercise 6d: Write a program that implements this design. Again, provide two user interfaces. We have provided some stub code to help you get started.

Press the button on the applet to see an example of the GUI (reverse engineering is prohibited!):

Good luck!

OOPSLA 2006 Educator's Symposium: Nifty Assignment

Temperature Calculator

Programming for Change

Dung "Zung" Nguyen, Mathias Ricken

Rice University, Houston, TX USA

Website with Applets and Updated Materials

A version of this document with working applets and updated materials is available at:

http://www.cs.rice.edu/~mgricken/research/tempcalc/website

Archive

Download all solutions as ZIP file.

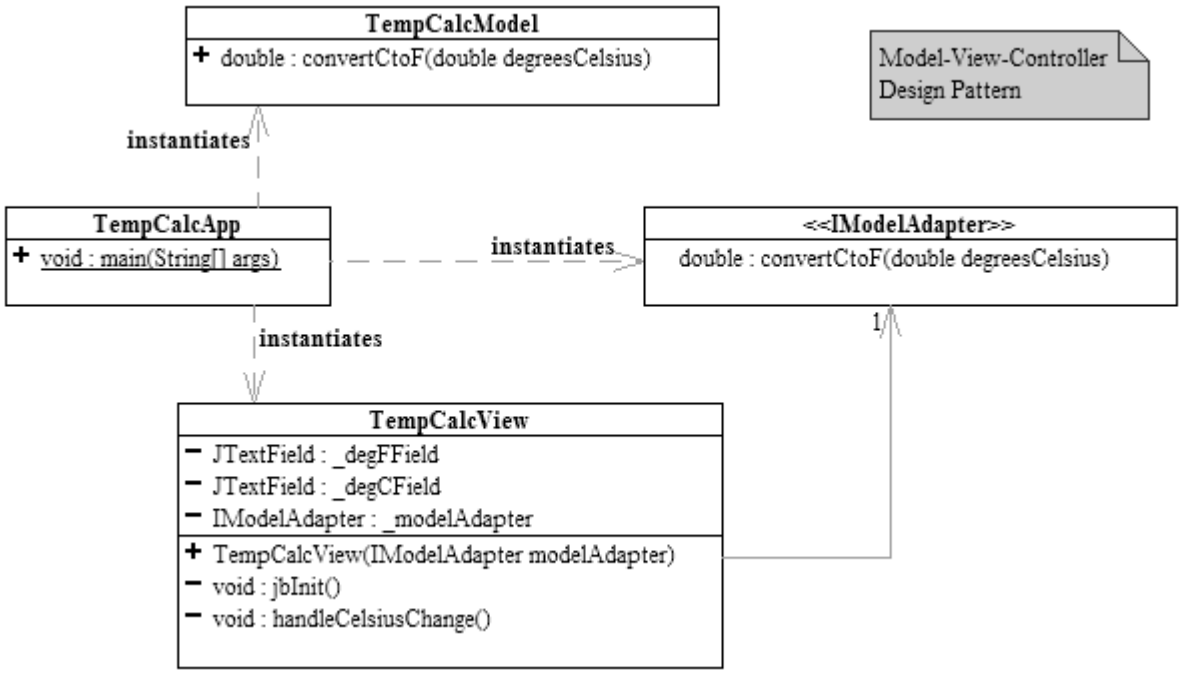
Solutions

Exercise 1a: Identify the variant and the invariant of converting a temperature from Celsius to Fahrenheit.

The variant is the numeric value of the temperature. The invariant is the conversion formula from Celsius to Fahrenheit.

Exercise 1b: Draw a UML diagram of the model. It should consist of a single static method.

This is the UML diagram of the entire program. The model consists of just the TempCalcModel class.

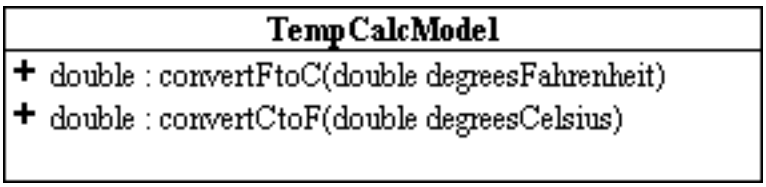


Exercise 1c: Write a program that allows the user to convert *any* temperature in Celsius to its corresponding value in Fahrenheit. You can start by modifying the code for the constant solution, but remember to provide the two user interfaces! Watch out for integer arithmetic being carried out instead of floating point arithmetic.

Download solution

Exercise 2a: Draw a UML diagram of a changed model that consists of two static methods and allows conversion in both directions.

This is the UML diagram of just the model. The rest of the program is similar to the one depicted in Exercise 1b.



Exercise 2b: Write a program that implements this design. Again, provide two user interfaces.

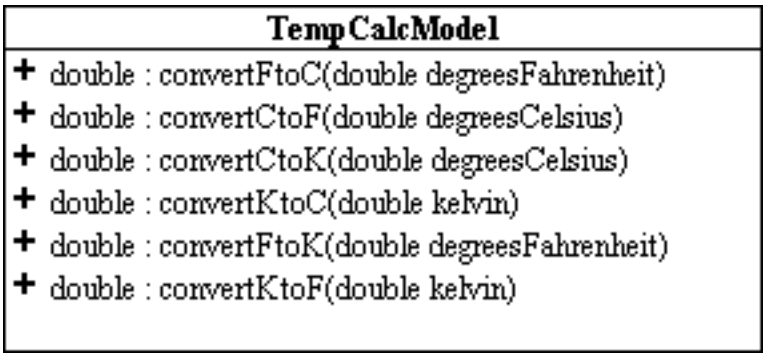
Download solution

Exercise 3a: If you follow the pattern from the first two solutions of using one static method per possible conversion, how many methods will you need for this model?

There are three temperature scales, each needing conversion functions to the two other scales: 3*2 = 6.

Exercise 3b: Draw the UML diagram of the changed model, with one static method for each possible conversion.

This is the UML diagram of just the model. The rest of the program is similar to the one depicted in Exercise 1b.



Exercise 3c: Write a program that implements this design. Again, provide two user interfaces.

Download solution

Exercise 4a: Compare the model from Exercise 2 to the model from Exercise 3. What is the ration between the number of methods in the models of Exercise 2 and Exercise 3?

The model for Exercise 2 required two methods; the one for Exercise 3 six. The ratio is 1:3.

Exercise 4b: Assume we added full support for the Reaumur scale to our program. Using our current approach of one method per possible conversion, how many methods will you need?

There are four temperature scales, each needing conversion functions to the three other scales: 4*3 = 12.

Exercise 4c: Using your answers to Exercises 3a, 4a and 4b, derive a formula - *f* - that allows you to calculate the number of methods, given the number of temperature scales - n: *f(n)* =

Each of the *n* temperature scales needs conversion functions to the *n-1* other scales. Therefore, *f(n) = n*(n-1)*.

Exercise 4d: What is the complexity (i.e. the number of methods) of this approach, expressed in "big O" notation?

f(n) = n(n-1) =* n^2*-n*. The dominating term is n^2, therefore the number of methods is in $O(n^2)$. The size of the program grows quadratically.

Exercise 5a: Using this new approach, what is the minimum number of methods we need to convert between three scales? Four scales? *n* scales?

Except for the first scale, which serves as a connection between the scales, every scale needs two conversion functions: to the first scale and back. The first scale doesn't need any conversions.

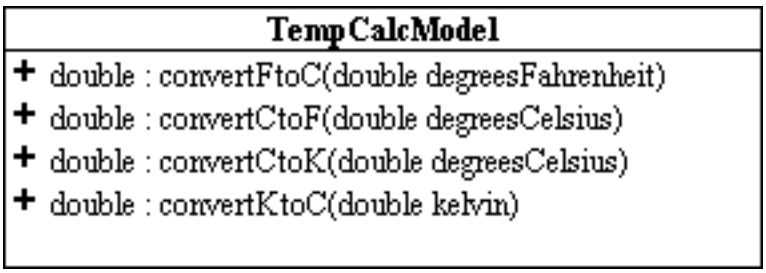
Three scales: 4 methods
Four scales: 6 methods
n scales: 2*(n-1) methods

Exercise 5b: What is the complexity (i.e. the number of methods) of this new approach, expressed in "big O" notation?

The dominating term of *2*(n-1) = 2n - 2* is *2n*. Therefore, the number of methods is in *O(n)* and we have achieved linear growth.

Exercise 5c: Draw a UML diagram of the model using the new approach. The model should have the minimum number of methods to still perform arbitrary conversions between Celsius, Fahrenheit, and Kelvin.

This is the UML diagram of just the model. The rest of the program is similar to the one depicted in Exercise 1b.



Exercise 5d: Write a program that implements this design. Again, provide two user interfaces; they should look the same as in the program for Exercise 3c.

Download solution

Exercise 6a: What differs and what always remains the same when performing conversions this way? Identify the variants and the invariants. Remember that we can pass functions as data in an object-oriented language like Java.

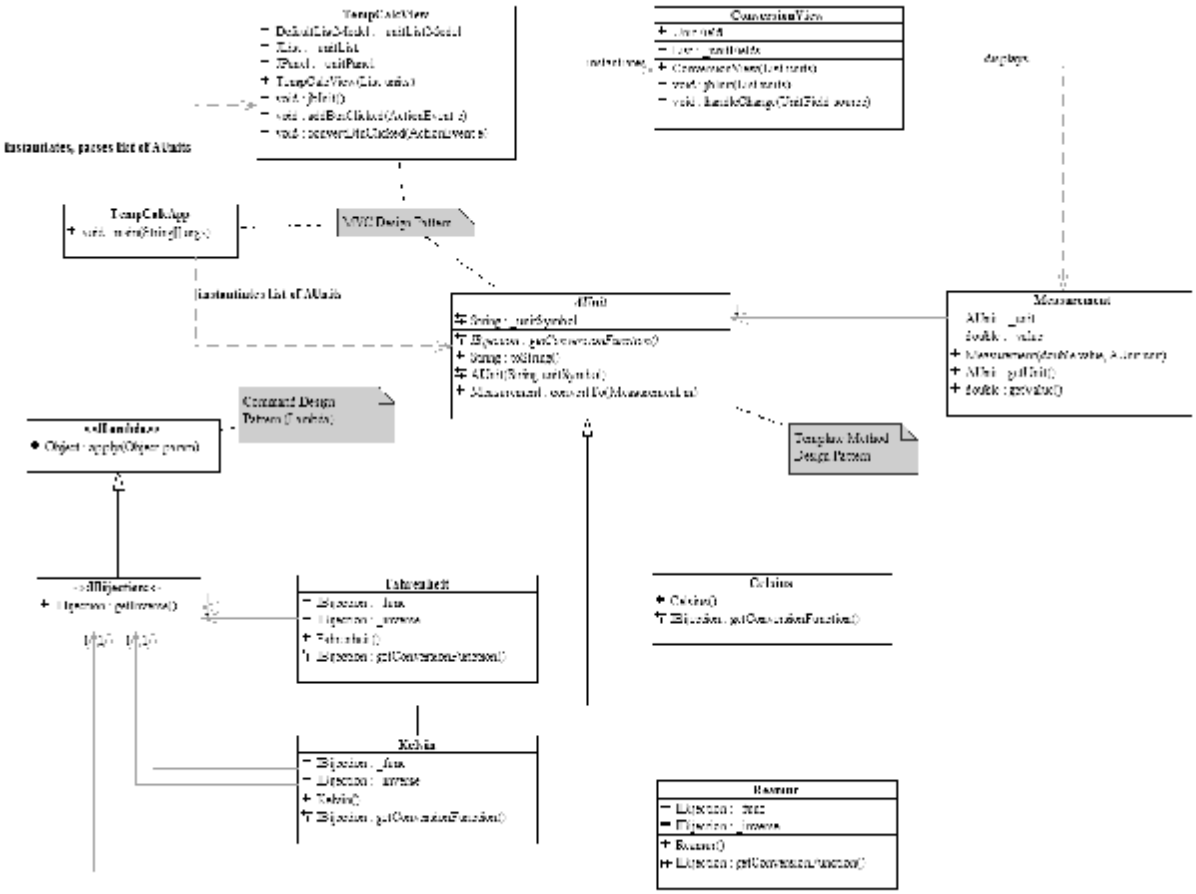
The variant is the choice of the two conversion functions: the function to convert to the base scale, and the function to convert back from the base scale. The invariant is the process of applying the input value to the first function, and then applying its result to the second function to obtain the final result.

Exercise 6b: What three pieces of data are necessary to define a temperature scale in our model? Hint: These are the invariants you identified earlier.

1. The function to convert to the base scale
2. The function to convert back from the base scale
3. A name or symbol for the temperature scale

Exercise 6c: Using the *IBijection* interface and the AUnit and Measurement classes, draw a UML diagram of the model with the Celsius, Fahrenheit, Kelvin, and Reaumur temperature scales.

This is the UML diagram of the entire program. The model consists of everything except for the TempCalcApp, TempCalcView and ConversionView classes.



Exercise 6d: Write a program that implements this design. Again, provide two user interfaces. We have provided some stub code to help you get started.

Download solution