Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	4, Object-Oriented Data Type Construction
Lecturer:	Dr. Markus Lumpe

To invent, you need a good imagination and a pile of junk.

Thomas A. Edison

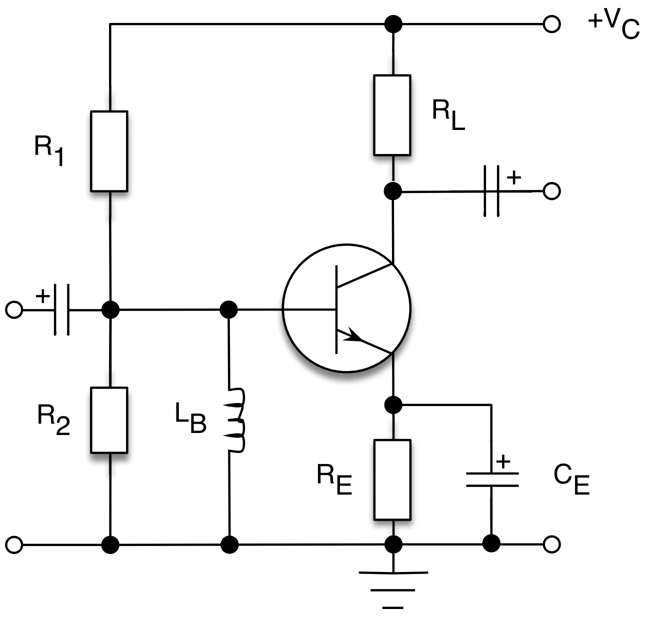

Figure 1: A Hypothetical Emitter Amplifier Circuit[1].

Lab 4: Object-Oriented Data Type Construction

Consider Figure 1, which shows a hypothetical single-stage amplifier circuit. Do not be alarmed, we are not going to analyze this circuit or start doing electronics in COS3008. It just serves as a motivation to experiment with object-oriented inheritance and method overriding, and to demonstrate that code reuse and incremental refinement are the key ingredients in object-oriented programming and data type construction.

The interesting features in the amplifier circuit with respect to object-oriented software design are the capacitor C_E and the inductor L_B. In this circuit, both serve as a frequency-dependent resistor to control the amplifier gain and frequency range. The specifics are not important and would rather distract from the task ahead. There is, however, an intriguing aspect: both *the capacitor and the inductor can be used in the place of a resistor* when we need a frequency-dependent resistor in a circuit. (Naturally, this description is simplified.)

In software terms, this is a familiar scenario. It is called polymorphism: objects of different types can respond uniformly to a given operation. The type of polymorphism we are interested in is called "subtype polymorphism" that arises from object-oriented inheritance and method overriding. To model the required domain concepts (here a passive resistor, a capacitor, and an inductor), we create a corresponding class hierarchy and set up as root class an abstract class, called `ComponentBase`, which both encapsulates the common behavior and defines the variable features through the definition of pure virtual methods. In the solution, we then can create three subclasses that implement the respective variable behavior: a class `PassiveResistor` that models electric resistance of a resistor (frequency independent), a class `Capacitor` that implements *capacitive reactance* of a capacitor (decreases with frequency), and a class `Inductor` that captures the inductive reactance of an inductor (increases with frequency).

[1] This circuit just serves as illustration. It may not work in reality.

Conditional Compilation

The test driver provided for this tutorial task makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

The test driver (i.e., main.cpp) uses `P1`, `P2`, `P3`, `P4`, and `P5` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
// #define P3
// #define P4
// #define P5
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX` … **`#endif`** are grayed out, if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX` … **`#endif`** enables conditional compilation. XCode does not use this color coding scheme.

In addition, the implementations for the methods `toUnitValue()` and `setBaseValue()` defined in class `ComponentBase` are given in `ComponentBaseAuxiliaries.cpp`. Please analyze the implementation. These two methods provide the infrastructure to process, scale, and normalize the specification of resistor, capacitor, and inductor values.

Problem 1

The abstract class `ComponentBase` is given as

```
#pragma once

#include <string>
#include <istream>
#include <ostream>

class ComponentBase
{
private:
  double fBaseValue;           // base value of component
  std::string fMajorUnit;      // major unit for component
  std::string fMinorUnits;     // minor units for component

  // converts base value to unit value (object remains unchanged)
  // Example: the value of a register 56000.0 in unitless form becomes 56.0 kOhm
  void toUnitValue( double& aValue, std::string& aUnit ) const noexcept;

  // converts unit value to base value (updates object)
  // Example: the value of a register 56.0 kOhm becomes 56000.0 in unitless form
  void setBaseValue( double aValue, const std::string& aUnit ) noexcept;

public:
  ComponentBase(double aBaseValue,
                const std::string& aMajorUnit,
                const std::string& aMinorUnits ) noexcept;

  // required virtual destructor (default implementation)
  virtual ~ComponentBase() {}

  // Getters and Setters:
  double getBaseValue() const noexcept;
  void setBaseValue( double aBaseValue ) noexcept;
  const std::string& getMajorUnit() const noexcept;
  const std::string& getMinorUnits() const noexcept;

  // virtual member functions
  // returns true if aValue exceeds a magnitude
  virtual bool mustAdjust( double aValue ) const noexcept = 0;

  // returns component dependent scalar
  virtual const double getScalar() const noexcept = 0;

  // returns (frequency-dependent) passive resistance value
  virtual double getReactance( double aFrequency = 0.0 ) const noexcept = 0;

  // Component functions:
  // returns (frequency-dependent) voltage drop
  double getPotentialAt( double aCurrent,
                         double aFrequency = 0.0 ) const noexcept;

  // returns (frequency-dependent) current flowing through a resistor
  double getCurrentAt( double aVoltage,
                       double aFrequency = 0.0 ) const noexcept;

  // I/O operators (friends of ComponentBase):
  // Reads text string "56.0 kOhm" to set base value to 56000.0 (register)
  friend std::istream& operator>>(std::istream& aIStream, ComponentBase& aObject);

  // Writes text string "56.0 kOhm" obtained from base value 56000.0 (register)
  friend std::ostream& operator<<(std::ostream& aOStream, const ComponentBase& aObject);
};
```

The abstract class `ComponentBase` is quite complex. The private features address the required data conversion (scaling) of the base value and unit, so that we can effectively specify resistor, capacitor, and inductor values. Somebody has already implemented those features. You need to study them in order to use them in your solution. A great help is the C++ online reference: http://www.cplusplus.com/reference.

ComponentBaseAuxiliaries.cpp:

```
#include "ComponentBase.h"

#include <cassert>

void ComponentBase::toUnitValue( double& aValue, std::string& aUnit ) const noexcept
{
  // get base value and minor units
  aValue = getBaseValue();
  std::string lPrefixes = getMinorUnits();

  for ( size_t i = 0; i < lPrefixes.size(); i++ )
  {
    // stop scaling at maximum unit
    if ( mustAdjust( aValue ) && (i < lPrefixes.size() - 1) )
    {
      aValue *= getScalar();
    }
    else
    {
      if ( i > 0 )
      {
        aUnit += lPrefixes[i];
      }

      aUnit += getMajorUnit();

      break;
    }
  }
}

void ComponentBase::setBaseValue( double aValue, const std::string& aUnit ) noexcept
{
  std::string lMajorUnit = getMajorUnit();
  std::string lMinorUnits = getMinorUnits();

  // test basic features (raw unit too long and not containing major unit)
  assert( aUnit.size() >= lMajorUnit.size() &&
          aUnit.find( lMajorUnit ) != std::string::npos );

  // test scale features, aUnit[0] must be contained in minor units
  assert( lMinorUnits.find( aUnit[0] ) != std::string::npos );

  // adjust base value
  double lMultiplier = 1.0;

  // i in 0 .. n
  size_t i = lMinorUnits.find( aUnit[0] );
  double lRawValue = aValue;

  // adjust raw value first
  for ( ; i > 0 ; i-- )
  {
    if ( mustAdjust( lRawValue ) )
      lRawValue /= getScalar();
    else
      break;
  }

  // adjust multiplier
  for ( ; i > 0; i-- )
    lMultiplier *= 1.0/getScalar();

  setBaseValue( lRawValue * lMultiplier );
}
```

These two private member functions preform data pre- and post-processing for the I/O operators. The method `toUnitValue()` allows for the internal base value to be converted into an external unit value. For example, a based value 56000.0 is converted to "56 kOhm" when we are working with a passive register. Similarly, the method `setBaseValue()` converts an input of "470 uF" to a base value 0.00047 for a capacitor. The I/O operators have to locally declare and use two variables of type `double` and `string`, respectively, in order to interact with the methods `toUnitValue()` and `setBaseValue()`. In addition, the behavior of these methods is controlled by methods `mustAdjust()`, `getScalar()`, `getMajorUnit()`, and `getMinorUnits()`, which yield results specific to the underlying domain concept (i.e., resistor, capacitor, or inductor). Please note that both, `toUnitValue()` and `setBaseValue()` use reference parameters. We use *call-by-reference*. For `toUnitValue()`, we use *out* parameters, whereas `setBaseValue()` has *in* parameters.

Please note, both `toUnitValue()` and `setBaseValue()` are decorated with the **`noexcept`** keyword. The methods are not expected to throw an exception, and if they do, then the program cannot continue. Method `setBaseValue()` performs two assertion checks. If either one fails, the program terminates. This is a debug feature. When we compile the program in Release mode, the assertion checks are disabled at runtime.

To implement abstract class `ComponentBase`, recall Ohm's law: $R = \frac{V}{I}$, the potential difference (voltage) across an ideal conductor is proportional to the current flowing through it. We can rewrite this formula to obtain the voltage drop at a resistor for a given current: $V = R * I$, and the current flowing through the resistor for a given voltage: $I = \frac{V}{R}$. The value R can be obtained by calling method `getReactance()` that takes a frequency as argument.

You cannot test the abstract class `ComponentBase` at this stage. We need a class that implement all the pure virtual members first. We start with class `PassiveResistor`.

The class `PassiveResistor` is given as

```
#pragma once

#include "ComponentBase.h"

class PassiveResistor : public ComponentBase
{
public:
  // constructor with a default value
  PassiveResistor( double aBaseValue = 0.0 ) noexcept;

  // returns true if aValue exceeds a magnitude (1000.0)
  bool mustAdjust( double aValue ) const noexcept override;

  // returns component dependent scalar (1.0/1000.0)
  const double getScalar() const noexcept override;

  // returns (frequency-dependent) passive resistance value (base value)
  double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class `PassiveResistor` does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class `PassiveResistor` inherits all public methods of class `ComponentBase`.) Please note that we make no claim that the class `PassiveResistor` faithfully implements the behavior of a passive resistor (e.g., we do not model the conversion from electrical to thermal energy). We just model electric resistance in this tutorial.

You may notice that we do not need to define explicit I/O operators for class `PassiveResitor`. We can reuse the existing ones from class `ComponentBase`. It works due to *dynamic method invocation*. In the operators, we call the service functions `toUnitValue()` and `setBaseValue()`, which in turn rely on the class-specific variants of `mustAdjust()` and `getScalar()` and the domain-specific values returned by `getMajorUnit()` and `getMinorUnits()`. For `PassiveResistor`, the major unit has to be set to "`Ohm`" whereas the minor units are set to "`OkM`". "Ohm" refers to the SI unit for resistors. "OkM" in turn enumerate possible unit factors: O – 10^0, k – 10^3, M – 10^6. Hence, when use input a `PassiveResistor` object, the input operator will format it according to the principles of a passive resistor. That is, it will use "Ohm" as unit and scale by 0.001. For example, if the base value of the resistor is `5600.0`, then the output operator needs to produce "5.6 kOhm".

Class `PassiveResistor` does not define instance variables for major and minor units. They are defined in the base class `ComponentBase`. You have to use the base class constructor for `ComponentBase` to initialize the major and minor units (in the member initializer list of the constructor for `PassiveResistor`).

Defining the class `PassiveResistor` in this way allows us to write

```
void runP1()
{
  PassiveResistor lR;

  cout << "Enter resistor value: ";
  cin >> lR;
  cout << "Passive resistor value: " << lR << endl;
  cout << "Current at 9.0 V: " << lR.getCurrentAt( 9.0 ) << " A" << endl;
  cout << "Voltage drop at 200 mA: " << lR.getPotentialAt( 0.2 ) << " V" << endl;
}
```

which should produce the following output

```
Enter resistor value: 56 Ohm
Passive resistor value: 56 Ohm
Current at 9.0 V: 0.160714 A
Voltage drop at 200 mA: 11.2 V
```

Problem 2

Using the abstract class ComponentBase, we can construct a class Capacitor to model "capacitive reactance." The capacitive reactance is the frequency-dependent resistance value of a capacitor given by the formula $X_C = \frac{1}{2\pi f C}$. Both, the value of the capacitor and the signal frequency significantly contribute to the capacitive reactance. In general, the capacitive reactance is inversely related to the signal frequency.

We define class Capacitor as a public subclass of class ComponentBase:

```
#pragma once

#include "ComponentBase.h"

class Capacitor : public ComponentBase
{
public:
  // constructor with a default value
  Capacitor( double aBaseValue = 0.0 ) noexcept;

  // returns true if aValue exceeds a magnitude (<1.0)
  bool mustAdjust( double aValue ) const noexcept override;

  // returns component dependent scalar (1000.0)
  const double getScalar() const noexcept override;

  // returns (frequency-dependent) passive resistance value (capacitive
  // reactance)
  double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class Capacitor does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class Capacitor inherits all public methods of class ComponentBase.) Please note that we make no claim that the class Capacitor faithfully implements the behavior of a capacitor. We just model capacitive reactance in this tutorial.

The implementation for the inherited virtual method getReactance() has to model the capacitive reactance, which requires the value of π. Use the following approach:

Before any #inlcude statement in your compilation unit, add the line

```
#define _USE_MATH_DEFINES
```

and include cmath afterwards. This will allow you to use the constant M_PI of type double in your program.

You may notice that we do not need to define explicit I/O operators for class Capacitor. We can reuse the existing ones from class ComponentBase. It works due to *dynamic method invocation*. In the operators, we call the service functions toUnitValue() and setBaseValue(), which in turn rely on the class-specific variants of mustAdjust() and getScalar() and the domain-specific values returned by getMajorUnit() and getMinorUnits(). For Capacitor, the major unit has to be set to "F" whereas the minor units are set to "Fmunp". "F" refers to the SI unit for capacitors. "Fmunp" in turn enumerate possible unit factors: F – 10^0, m – 10^{-3}, u – 10^{-6}, n – 10^{-9}, p – 10^{-12}. Hence, when use input a Capacitor object, the input operator will format it according to the principles of a capacitor. That is, it will use "F" (Farad) as unit and scale by 1000.0. For example, if the base value of the capacitor is 0.0000047, then the output operator needs to produce "4.7 uF" (we use the letter 'u' instead of the Greek symbol μ).

Class `Capacitor` does not define instance variables for major and minor units. They are defined in the base class `ComponentBase`. You have to use the base class constructor for `ComponentBase` to initialize the major and minor units (in the member initializer list of the constructor for `Capacitor`).

Defining the class `Capacitor` in this way allows us to write

```
void runP2()
{
  Capacitor lC;

  cout << "Enter capacitor value: ";
  cin >> lC;
  cout << "Capacitor value: " << lC << endl;

  // create a temporary passive resistor object to properly format value
  cout << "XC at 60 Hz: " << PassiveResistor( lC.getReactance( 60.0 ) ) << endl;
  cout << "Current at 9 V and 60 Hz: " << lC.getCurrentAt( 9.0, 60.0 ) << " A"
       << endl;
  cout << "Voltage drop at 2 mA and 60 Hz: " << lC.getPotentialAt( 0.002, 60.0 )
       << "V" << endl;
}
```

which should produce the following output

```
Enter capacitor value: 470 uF
Capacitor value: 470 uF
XC at 60 Hz: 5.64379 Ohm
Current at 9 V and 60 Hz: 1.59467 A
Voltage drop at 2mA and 60Hz: 0.0112876 V
```

Note, we use `PassiveResistor( lC.getValue( 60.0 ) )` to convert the capacitive reactance into a plain resistor value in order to obtain the desired formatted output.

Notre: Formatted input for doubles eats the letters "f" and "F" with for double values! So, the input 470F is read as 470.0 of type float, not 470 Farad. To avoid this issue, always put a space between the number of the unit symbol.

Problem 3

Using the classes `PassiveResistor` and `Capacitor`, we can construct a time series to determine the signal frequency at which a 470 μF (0.00047) capacitor reaches a capacitive reactance of less than 1 Ohm. A do-while loop that increments the signal frequency by 50 Hz in each step works best for this purpose.

```
void runP3()
{
  Capacitor lC( 0.00047 );

  cout << "Capacitor value: " << lC << endl;

  double lXC = 0.0;
  double lFrequency = 50.0;

  do
  {
    lXC = lC.getReactance( lFrequency );
    cout << "XC at " << setw( 5 ) << lFrequency << " Hz:\t"
         << PassiveResistor( lXC ) << endl;
    lFrequency += 50.0;
  } while (lXC > 1.0);
}
```

We should reach a capacitive reactance of less than 1 Ohm in seven steps:

```
XC at    50 Hz:     6.77255 Ohm
XC at   100 Hz:     3.38628 Ohm
XC at   150 Hz:     2.25752 Ohm
XC at   200 Hz:     1.69314 Ohm
XC at   250 Hz:     1.35451 Ohm
XC at   300 Hz:     1.12876 Ohm
XC at   350 Hz:     0.967507 Ohm
```

Problem 4

Using the abstract class `ComponentBase`, we can construct a class `Inductor` to model "inductive reactance." The inductive reactance is the frequency-dependent resistance value of an inductor given by the formula $X_C = 2\pi f L$. Both, the value of the inductor and the signal frequency significantly contribute to the inductive reactance. In general, the inductive reactance changes proportionally with the signal frequency.

To model a capacitor, we define class `Inductor` as a public subclass of class `ComponentBase`:

```
#pragma once

#include "ComponentBase.h"

class Inductor : public ComponentBase
{
public:
  // constructor with a default value
  Inductor( double aBaseValue = 0.0 ) noexcept;

  // returns true if aValue exceeds a magnitude (<1.0)
  bool mustAdjust( double aValue ) const noexcept override;

  // returns component dependent scalar (1000.0)
  const double getScalar() const noexcept override;

  // returns (frequency-dependent) passive resistance value (inductive
  // reactance)
  double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class `Inductor` does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class `Capacitor` inherits all public methods of class `ComponentBase`.) Please note that we make no claim that the class `Inductor` faithfully implements the behavior of a capacitor. We just model capacitive reactance in this tutorial.

The implementation for the inherited virtual method `getReactance()` has to model the capacitive reactance, which requires the value of π. Use the following approach:

Before any #inlcude statement in your compilation unit, add the line

`#define _USE_MATH_DEFINES`

and include `cmath` afterwards. This will allow you to use the constant `M_PI` of type `double` in your program.

You may notice that we do not need to define explicit I/O operators for class `Inductor`. We can reuse the existing ones from class `ComponentBase`. It works due to *dynamic method invocation*. In the operators, we call the service functions `toUnitValue()` and `setBaseValue()`, which in turn rely on the class-specific variants of `mustAdjust()` and `getScalar()` and the domain-specific values returned by `getMajorUnit()` and `getMinorUnits()`. For `Inductor`, the major unit has to be set to "`H`" whereas the minor units are set to "`Hmunp`". "H" refers to the SI unit for inductors. "Hmunp" in turn enumerate possible unit factors: H – 10^0, m – 10^{-3}, u – 10^{-6}, n – 10^{-9}, p – 10^{-12}. Hence, when use input an `Inductor` object, the input operator will format it according to the principles of a inductor. That is, it will use "H" (Henry) as unit and scale by 1000.0. For example, if the base value of the inductor is `0.2`, then the output operator needs to produce "200 mH".

Class `Inductor` does not define instance variables for major and minor units. They are defined in the base class `ComponentBase`. You have to use the base class constructor for `ComponentBase` to initialize the major and minor units (in the member initializer list of the constructor for `Inductor`).
Defining the class `Inductor` in this way allows us to write

```
void runP4()
{
  Inductor lL;

  cout << "Enter inductor value: ";
  cin >> lL;
  cout << "Inductor value: " << lL << endl;

  // create a temporary passive resistor object to properly format value
  cout << "XC at 10 kHz: " << PassiveResistor( lL.getReactance( 10000.0 ) )
       << endl;
  cout << "Current at 9 V and 10 kHz: " << lL.getCurrentAt( 9.0, 10000.0 )
       << " A" << endl;
  cout << "Voltage drop at 2 mA and 10 kHz: "
       << lL.getPotentialAt( 0.002, 10000.0 ) << " V" << endl;
}
```

which should produce the following output

```
Enter inductor value: 0.2 H
Inductor value: 200 mH
XC at 10 kHz: 12.5664 kOhm
Current at 9 V and 10 kHz: 0.000716197 A
Voltage drop at 2 mA and 10 kHz: 25.1327 V
```

Note, we use `PassiveResistor( lC.getValue( 10000.0 ) )` to convert the inductive reactance into a plain resistor value in order to obtain the desired formatted output.

Problem 5

Using the classes `PassiveResistor` and `Inductor`, we can construct a time series to determine the signal frequency at which a 520 mH (0.52) inductor reaches a capacitive reactance of more than 1 kOhm. A do-while loop that increments the signal frequency by 50 Hz in each step works best for this purpose.

```
void runP5()
{
  Inductor lL( 0.52 );

  cout << "Inductor value: " << lL << endl;

  double lXC = 0.0;
  double lFrequency = 50.0;

  do
  {
    lXC = lL.getReactance( lFrequency );
    cout << "XC at " << setw( 5 ) << lFrequency << " Hz:\t"
         << PassiveResistor( lXC ) << endl;
    lFrequency += 50.0;
  } while (lXC < 1000.0);
}
```

We should reach an inductive reactance of more than 1 kOhm in seven steps:

```
XC at    50 Hz:     163.363 Ohm
XC at   100 Hz:     326.726 Ohm
XC at   150 Hz:     490.088 Ohm
XC at   200 Hz:     653.451 Ohm
XC at   250 Hz:     816.814 Ohm
XC at   300 Hz:     980.177 Ohm
XC at   350 Hz:     1.14354 kOhm
```