1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.tar or .zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose, compilation and execution instructions, and any additional details you feel are relevant.

2 Learning Outcomes

In this assignment you will learn to

1. Change a class implementation while keeping the same interface.
2. Refactor an existing codebase to use inheritance and polymorphism.
3. Make a UML diagram to illustrate the design of the codebase.

3 Overview

In this assignment you will be refactoring the codebase that we have made in Assignments 1 and 2. As a starting point, you may use your own assignments, or use the code provided (solutions to Assignment 2 will be provided after the submission deadline).

We have made `Panel`, `FlowPanel`, `Button`, and `TextArea` classes. However, these 4 classes have some core features in common. All 4 draw rectangles on a window, within their parent component. Both `Panels` contain other components. We will gather these similar features into parent classes.

As before there is a `TestControl` class that connects and coordinates the functionality of your classes with a `View` class in order to run tests.

You will be required to make a UML diagram of the completed project.

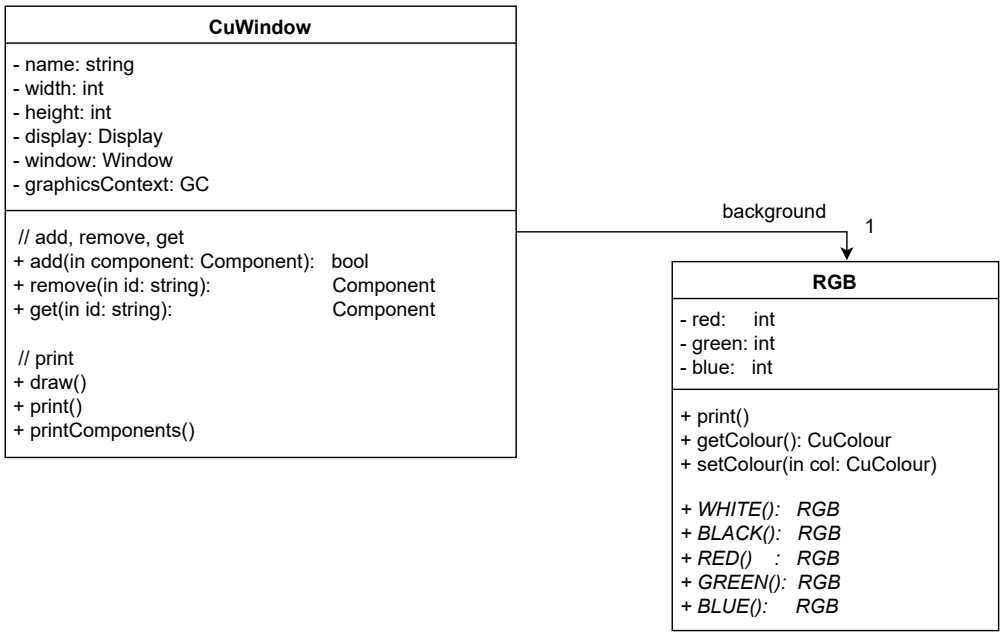
4 UML Diagram

Make a UML diagram of the completed project. You may omit the `TestControl`, `Tester`, and `View` classes. A partial UML diagram is provided to help you get started.

5 Classes Overview

This application will consist of 13 classes.

1. The `Rectangle` struct (Entity object):
   (a) Contains `Rectangle` information and functions.
2. The `RGB` class (Entity object):
   (a) Contains colour information.
3. The `Component` class, an abstract class (Entity object):



   (a) Contains the preferred draw location of the `Component` (as a `Rectangle`), and an `id` member used to identify the given `Component`.

4. The `Button` class (Entity object). Concrete subclass of `Component`.

5. The `TextArea` class (Entity object). Concrete subclass of `Component`.

6. The `ComponentList` class (Collection object). Data structure for `Component` pointers, used by the `Panel` class.

7. The `Panel` class (Collection, Entity). An abstract subclass of `Component`.

   (a) Manages a collection of `Components` that it will attempt to draw within its boundaries. How it attempts to draw the contained `Components` will be decided by the concrete derived classes.

   (b) Provides functions to add, delete, access, and print `Components`.

8. The `AbsolutePanel` class (Collection, Entity object). Concrete subclass of `Panel`.

   (a) Attempts to draw each contained `Component` in its *preferred* location.

9. The `FlowPanel` class (Collection, Entity object). Concrete subclass of `Panel`.

   (a) Attempts to draw each contained `Component` in a flow layout.

10. The `CuWindow` class (Collection, Control, Entity object).

    (a) Manages the window, and the `Components` that are drawn on the window.

11. The `View` class (Boundary object):

    (a) Presents a menu, takes input from the user

12. The `TestControl` class (Control object):

(a) Manages the interaction of the other objects in order to run tests.

13. The `Tester` class (???):

    (a) Provides testing functionality.

6 Instructions

Download the starting code from Brightspace. It includes some global functions that you are to use for testing as well as some classes. All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION (except for `ComponentList`). This print function should display the metadata of the class using appropriate formatting.

Your finished code should compile into an executable called `a3` using the command `make all` or simply `make`. The Makefile is provided for you. Your submission should consist of a single zip file with a suitable name (e.g., `assignment3.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included, a directory structure (if applicable), compiling and running instructions, and any other information that will make the TAs life easier when they mark your assignment.

Though you will have the Assignment 1 and 2 solutions to work from, pay close attention to the interface (the function prototypes), and the application requirements, as some of them have changed.

6.1 Encapsulation

You should apply the `const` keyword to existing classes wherever possible. That means any function that can be made `const` should be, and any parameter that can be made `const` should be. The only exceptions are `(Test)Control`, `Tester`, and `View` classes - you do not need to apply `const` to these classes.

6.2 The Rectangle struct

This is provided for you in the file `defs.h`.

6.3 The RGB class

This is provided for you in the Assignment 2 solution, or you may use your own version. You may make any modifications you deem necessary.

6.4 The Component Class

Every GUI element that is drawn on a window is a `Component`.

1. Member variables:

   (a) Should have a `Rectangle` which is the preferred location and dimensions of the `Component`.

   (b) Should have a `string id` that is the identifier of this `Component`.

2. You should make *two* Constructors - one which takes a `Rectangle` and a `string id` parameter. The other should take 4 ints (`x, y, width, height`) and a `string id` parameter.

3. Member functions:

   (a) `bool overlaps` - this should take another `Component` as an argument, and return `true` if the preferred `Rectangle` of one overlaps the preferred `Rectangle` of the other.

(b) `void print` - this should be virtual, but it is up to you whether you want to make this pure virtual or not.

(c) `void draw` - this should be a pure virtual function that takes a `Display` pointer, a `Window`, a `GC`, and a `Rectangle` where the `Component` should be drawn.

6.5 The ComponentList Class

The public interface of this class should be the following:

```
ComponentList();
~ComponentList();

// Add comp to the back of the list
bool add(Component* comp);
// Add ta at the given index
bool add(Component* comp, int index);
// Remove the Component with the given id.
// Return nullptr if the Component is not found
Component* remove(const string& id);
// Remove the Component at the given index.
// Return nullptr if the index is out of bounds
Component* remove(int index);
// Return the Component with the given id.
// Return nullptr if the Component is not found
Component* get(const string& id) const;
// Return the Component at the given index.
// Return nullptr if the index is out of bounds
Component* get(int index) const;
// Return the number of elements currently stored in the list
int getSize() const;
```

This is essentially the `TAArray` interface (minus the `isFull` function, which is no longer needed). It should operate the same as `TAArray`, however, you are to implement this using a *doubly-linked list*.

Observe that you can use the `TAArray` implementation (replacing each occurence of `TAArray` with `ComponentList`), and it will work correctly so that your tests pass. But you will lose marks for not making it a doubly-linked list (though this is preferable to having nothing working at all).

6.6 The Button Class

This class should inherit from `Component`.

1. Member variables:

   (a) A `string label`. The text that is displayed on the `Button`.

   (b) `RGB` values for the border colour and fill colour of the `Button`. These parameters should have reasonable default values. OPTIONAL - add a third `RGB` value for when the `Button` is clicked. There will be a bonus mark for making your `Button` clickable.

2. Constructor: The constructor will be similar to the existing constructor, however, you should add a `string id` parameter just *before* the `label` (similar to the `TextArea` constructor where `id` comes just before `text` in the parameter list).

3. Member functions:
   (a) `void print`. Override the `Component::print` function. Print out the `label`, the `id`, and the preferred `Rectangle` dimensions. You can also print the colours if you wish.
   (b) `void draw`. Override the `Component::draw` function. Draw the `Button` using the given `Rectangle` *parameter*. Note that the preferred `Rectangle` is only a suggestion - when drawing components, use the `Rectangle` given as a parameter.

6.7 The TextArea Class

This class should inherit from `Component`.

1. Member variables:
   (a) A `string text`. The text that is displayed on the `TextArea`.
   (b) `RGB` values for the border colour and fill colour of the `TextArea`.
2. Constructor: The constructor will be the same as the existing constructor.
3. Member functions:
   (a) `void print`. Override the `Component::print` function. Print out the `id`, the preferred `Rectangle` dimensions, and the `text`.
   (b) `void draw`. Override the `Component::draw` function. Draw the `TextArea` within the given `Rectangle` parameter.

6.8 The Panel Class

This class should inherit from `Component`. It is an abstract class whose main function is to add a `ComponentList` and functions for adding, getting, and removing `Components`.

In addition to the functions outlined below, this class should duplicate all the functions listed for the `ComponentList` class (`add`, `remove`, `get`, etc) in Section 6.5. There are two ways to do this - using composition (that is, adding a `ComponentList` member variable to the `Panel` class) or by making `Panel` inherit from `ComponentList`. Either is acceptable.

1. Constructors: This class should have the same constructors, with the same parameters, as the `Component` class.
2. All member functions listed in Section 6.5.
3. Additional member functions:
   (a) `void print`. Override the `Component::print` function. Print out the `id`, the preferred `Rectangle` dimensions, and the number of contained `Components`.
   (b) `void printComponents`. Print out all the contained `Components`.

6.9 The AbsolutePanel Class

This class should inherit from `Panel`. It is concrete subclass of `Panel` whose main function is attempt to draw all `Components` in an absolute layout using the *preferred* `Rectangle` of each `Component`.

The behaviour is slightly different from the `Panel` class of assignment 1. In the assignment 1 `Panel` class, we did not add a `Button` to the current `Panel` if it overlapped another `Button`.

In the `AbsolutePanel`, we do not check for overlap when we add a `Component`. Instead, we check for overlap when drawing the `Components`. When you go to draw the `Components`, a `Component` is drawn only if it does not overlap a previously drawn `Component`. That means you will have to keep track of which `Components` were drawn and which were not, as you attempt to draw all the contained `Components`.

1. Constructors: This class should have the same constructors, with the same parameters, as the `Component` class.
2. Member functions:
   (a) `void print`. Print out the type of layout (that is, "absolute layout"), the `id`, the preferred `Rectangle` dimensions, and the number of contained `Components`.
   (b) `void draw`. Override the draw function according to the instructions above. As in the previous two assignments, a `Panel` should be invisible, but we will draw a rectangle around it for debugging purposes. In this function, when calling `draw` on the contained `Component`, you should pass in the preferred `Rectangle` of that `Component` as a parameter[1].

6.10 The FlowPanel Class

This class should inherit from `Panel`. It is concrete subclass of `Panel` whose main function is attempt to draw all `Components` in a *flow* layout (as implemented in assignment 2).

NOTE: In assignment 2, we implemented a deep copy of the `FlowPanel` class. That is no longer a requirement, as it is a bit trickier to do with polymorphism.

1. Constructors: This class should have the same constructors, with the same parameters, as the `Component` class.
2. Member functions:
   (a) `void print`. Print out the type of layout (that is, "flow layout"), the `id`, the preferred `Rectangle` dimensions, and the number of contained `Components`.
   (b) `void draw`. Override the draw function so that the `Components` are drawn in a flow layout as described above. As in the previous two assignments, a `Panel` should be invisible, but we will draw a rectangle around it for debugging purposes.

6.11 The CuWindow Class

The `CuWindow` class handles the X11 logic for making a display, opening a window and getting a graphics context for drawing. It should contain a single `Panel` (which we will call the `root`).

1. Memory management.
   (a) `Components` (other than the root `Panel`) are not created in the `CuWindow`. A user should create a new `Component`, make any necessary changes to it, and add the `Component` pointer to the `CuWindow`. At this point, the responsibility for the dynamic memory is transferred from the user to the `CuWindow`. As such, when a `CuWindow` is destroyed, it should delete every `Component` that it contains.
   (b) If a user calls `get` (which returns a pointer to a `Component` to the user, but does not remove the `Component` from the `CuWindow`), the responsibility for deleting that `Component` still lies with the `CuWindow`.
   (c) If a user calls `remove` then responsibility for deleting that `Component` is transferred to the user.
2. Member variables:
   (a) `int width, int height`: the current width and height of the window in pixels.
   (b) `string name`: The name of the window (which should be displayed at the top)
   (c) A `Panel root` to hold `Components`. By default this should be an `AbsolutePanel`. It is currently not shown on the partial UML diagram provided.

[1] Probably not the greatest design, but hey, it works and keeps things simple-ish. Feel free to implement a better design.

(d) An `RGB` member for the background colour of the window.

3. In addition, these member variables are necessary to maintain and draw on an X11 window:
   (a) `Display* display`: Connection to the X server.
   (b) `Window window`: To store the XID of the window that we opened.
   (c) `GC gc`: A graphics context (so we can draw on the window).
4. Constructors:
   (a) A constructors that take `name, width, height, background` as arguments and initializes the member variables appropriately. The background should be an `RGB` object. The constructor should also open a display, a window, and create a graphics context.
   (b) You should also have a destructor. This should free the graphics context, destroy the window, and close the display, and clean up any memory necessary.
5. Member functions (note there are some changes to these function definitions from Assignment 2):
   (a) You should make getters and setters as needed.
   (b) `add` - This function should take a `Component` pointer as an argument. Return true if the `FlowPanel` is added, and false otherwise. Note that this is the same behaviour as in Assignment 1.
   (c) `remove` - This function should find the `Component` with the given `id`, remove it, and return the pointer. Return `nullptr` if no such `Component` exists.
   (d) `get`: This function should return a pointer to the `Component` with the given `id`, or else `nullptr` if no such `Component` exists.
   (e) A `draw` function. You should first fill the window with a rectangle to "blank" everything out and provide a background colour (using the `RGB` member variable for the colour). Then you should draw the `root Panel`.

   `NOTE:` X11 does not synchronize by default. Thus, if there are changes being made to `CuWindow` and we attempt to draw, it may not be rendered properly. It is HIGHLY recommended that at the top of this draw function, before doing anything else, you sleep a bit so that any changes to `CuWindow` can be completed. Run the command `usleep(100000)` as the very first line of the `draw` function. You will also need `#include <unistd.h>` at the top of your file.
   (f) A `print` function. This should print (to the console, not to the window) the name of the window and the number of `Components`.
   (g) A `printComponents` function. This should print out all the `Components`.

6.12 The TestControl Class

This class has been done for you. It interacts with your classes and the `View` to run a series of tests and output your mark.

6.13 The View Class

This class has been done for you. It interacts with the user and returns values to the control object

6.14 The `main.cc` File

These have been done for you. `main.cc` is compiled into an executable `a3`. The `a3` executable runs tests and gives you your mark.

7 Grading

The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are where marks are earned. The third category, **Deductions** is where you are penalized marks.

7.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). The test suite will automatically allocate the marks, though they can be adjusted by the marking TA if some anomaly is found.

General Requirements

- All marking components must be called and execute successfully to earn marks.
- All data handled must be printed to the screen to earn marks (make sure the various `print` functions print useful information).

Automated tests [21 marks]:

- 1. [2 marks] TextArea and Button test.
- 2. [3 marks] Panel test.
- 3. [4 marks] Component in FlowPanel test
- 4. [4 marks] Component in AbsolutePanel test
- 5. [4 marks] Panels in FlowPanel test
- 6. [4 marks] Components in CuWindow test.

Rendering tests [6 marks]:

- 7. [3 marks] Render Test 1 (see Figure 1 for expected output)
- 8. [3 marks] Render Test 2 (see Figure 2 for expected output)

Memory tests - run with valgrind [4 marks]:

- 9. [2 marks] Panel memory tests.
- 10. [2 marks] CuWindow memory tests.

UML Diagram [5 marks]:

- 11. [5 marks] UML Diagram.

Possible bonus marks:

- 1. [1 mark] Make your components "clickable" - output a message to `cout` saying which component is clicked.
- 2. [1 mark] Have your `Buttons` change colour when clicked.

Please document in your README how a TA should test these capabilities.

Requirements Total: 36 marks (plus 2 bonus)

Figure 1: Render Test 1: A Window using an `AbsolutePanel` as root. Note that in the second figure, `Panel 3` is removed, so `Panel 4` can be rendered, but `Panel 4` now blocks `Panel 5` from being rendered.

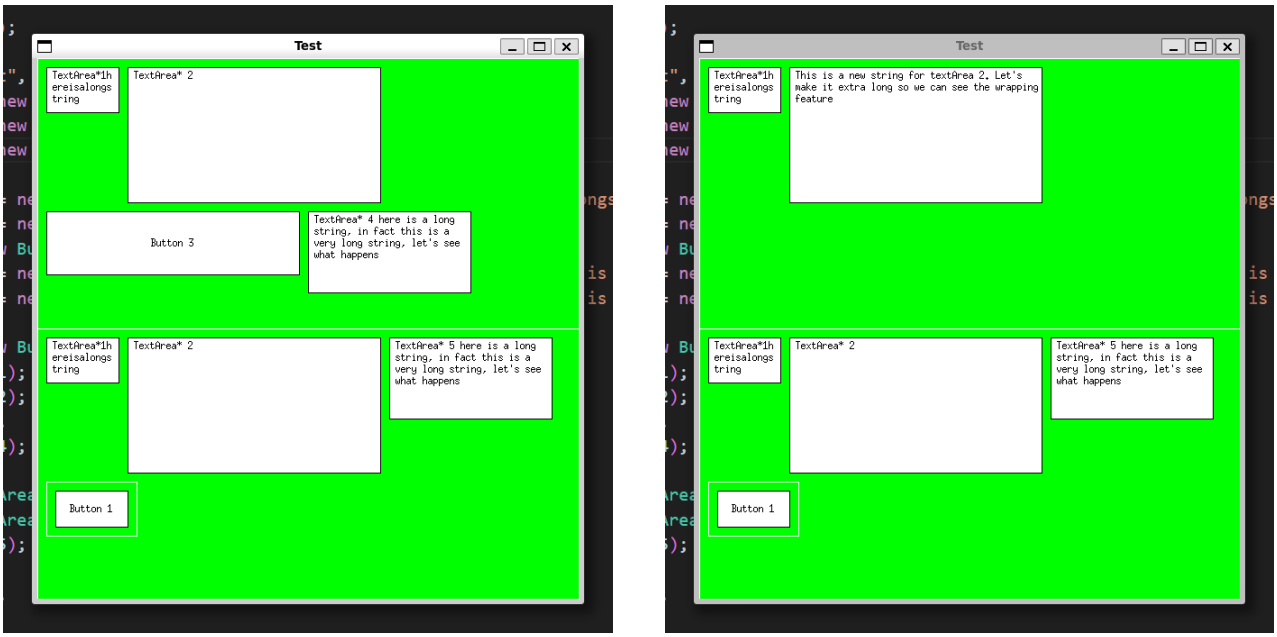


Figure 2: Render Test 2: `2 FlowPanels` in the `root`. Note that in the second figure, `TextArea 4` is removed, and `Button 3` is made too large to be drawn. Also the text in `TextArea 2` is changed. `Button 1` is within a `FlowPanel` within a `FlowPanel` within `root`.

7.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Apply "`const`"-ness to your program.
  - Print statements, getters, and any member function that does not change the value of any member variables should be `const`.
  - Any parameter object (passed by reference) that will not be modified should be `const`.
- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation of member variables (statically or dynamically)
- Proper instantiation of objects (statically or dynamically)
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
- Reusing existing functions wherever possible *within reason*. There are times where duplicating tiny amounts of code makes for better efficiency.
- Proper error checking - check array bounds, data in the correct range, etc.

7.2.1 Constraints: 14 marks

This also includes proper use of virtual functions.

1. 2 marks: Proper implementation and `const`-ing of the `Component` class.
2. 6 marks: Proper implementation and `const`-ing of the `ComponentList` class
   (a) `ComponentList` is a doubly-linked list.
3. 2 marks: Proper implementation and `const`-ing of the `Button` and `TextArea` classes.
4. 2 marks: Proper implementation and `const`-ing of the `Panel` class.
5. 2 marks: Proper implementation and `const`-ing of the `AbosolutePanel` and `FlowPanel` classes.

Constraints Total: 14 marks

Requirements Total: 36 marks

Assignment Total: 50 marks

7.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed.

7.3.1 Documentation and Style

1. Up to 10%: Improper indentation or other neglected programming conventions.
2. Up to 10%: Code that is disorganized and/or difficult to follow (use comments when necessary).

7.3.2 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.
5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

7.3.3 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf` or `scanf`, do use `cout` and `cin`).
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided for initialization and testing purposes.

7.3.4 Unapproved libraries:

- Up to 100%: The code must compile and execute in the default course VM provided. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*