PA5: Pro Paint (Due 11/1 @ 11:59PM)

The GitHub classroom link for this assignment is: https://classroom.github.com/a/BiuzQwDN

Installing wxWidgets

We are using the cross-platform wxWidgets library for this PA and the next one. First you need to install it so that CMake can find it.

Windows

- Make sure you've updated your Visual Studio 2017 to the latest version
- Download and install this, keeping the install directory the default: https://github.com/wxWidgets/wxWidgets/releases/download/v3.0.3/wxMSW-3.0.3-Setup.exe
- Then, download this archive and copy the lib folder into C:\wxWidgets-3.0.3 https://github.com/wxWidgets/wxWidgets/releases/download/v3.0.3/wxMSW-3.0.3_vc140_x64_Dev.7z (you have to extract it with 7-zip)
- Similarly, download this archive and copy the lib folder into C:\wxWidgets-3.0.3 https://github.com/wxWidgets/wxWidgets/releases/download/v3.0.3/wxMSW-3.0.3_vc140_x64_ReleaseDLL.7z

Mac

- From the terminal, do: `brew install wxmac`

Linux

- `sudo apt-get install libwxgtk3.0-dev`

Unit Tests

For this assignment, there are no graded tests. *However, we still expect you to write unit tests for PaintModel, and we also expect you to check your Travis warnings*.

Introduction

I have provided the code that creates a window with menus and a toolbar. Your job will be the hook up all of these menus and toolbars so their corresponding functionality works, and in the meantime, get some experience using design patterns and smart pointers.

In this assignment, you'll be implementing a "smart art" style paint program that will allow you to draw some simple shapes, move and change their colors, undo/redo, and load/save to image files. Once done, you will be able to make cool drawings.

Note on Smart Pointers

Internally, wxWidgets uses its own form of reference counting. So, if you have a pointer to a wx class such as a `wxMenu` or `wxFrame`, you should not use a smart pointer. If you use a smart pointer for such classes, it is likely you will get some sort of crash on exit (because your smart pointer will try to delete the object, but so will wx). You will lose points for a crash on exit.

However, for all of our custom-made classes, which include `PaintModel`, `Command` (and derived), `Shape` (and derived), you are expected to use smart pointers instead of raw pointers.

Model-View-Controller
Most GUI programs use some form of the "model-view-controller" design pattern, which we briefly discussed during lecture. The "model" is the underlying data representation of the document (in our case, a drawing). For our program, the "model" is the `PaintModel` class, which right now is pretty sparse but will have a lot of additional functions as you progress through the lab.

The "view" is the visualization of the model. It's possible to have multiple visualizations of the model at once, but in our program, there will be only one view at a time. When the model changes, the view should automatically update. In our program, the `PaintDrawPanel` class is the view.

The "controller" is what can apply changes to the model. These can be actions such as mouse clicks, menu commands, etc. In our program, the `PaintFrame` class is the controller.

So the basic idea is the end user interacts directly with the controller, which in turn makes changes to the model, and then the view is updated.

Provided Code
The skeleton of controller class (`PaintFrame`) is provided for you. However, you will actually have to make many of its functions do stuff.

The view class (`PaintDrawPanel`) should not require any changes during the lab. Notice that it has a `shared_ptr` the model, so it knows which model to draw. You will call the `PaintNow` from `PaintFrame` when you want to update the view.

The model class (`PaintModel`) does not have much yet. You will be adding quite a bit of code to this throughout the lab.

The `CursorCache` class is a flyweight of the different cursor images. You should not need to change this.

`Command` and `Shape` are provided abstract base classes. A "Command" is any action that mutates (modifies) the `PaintModel`. The reason these are separate is to help with Undo/Redo, which is done later in the lab. When a `Command` is created, it has a "start" point that corresponds to the point where on the drawing the command begins (we use the `wxPoint` class for points). As a `Command` is updated, it calls `Update` with the new point that the command is now at. When the `Command` is done updating, `Finalize` is called.

Let's say the user selects the rectangle drawing tool. When the user first clicks, a subclass of `Command` called `DrawCommand` is constructed with that point as the start point. Then as the user drags the mouse around (with the button still held), the command is updated. When the user releases the mouse, it's finalized.

The `Shape` is the base class for shapes that can be drawn. A `Shape` is defined by a start and end, and the creation/update/finalize is much like the behavior for a `Command`. However, notice that there also is a TopLeft and BotRight point. These define the "bounding box" for the shape.

The `Intersects` and `Update` functions for the `Shape` class are implemented for you. Shapes can optionally override `Update`, but you will only end up needing to do that for `PencilShape`.

Running the Program
For this lab, you will mostly want to run the "main" target instead of the tests one, as that shows the UI. On Mac, don't forget to set your working directory.

Part 1: DrawCommand and Shapes

If you look at the `CommandType` enum, you'll see that there are different commands for DrawLine, Ellipse, Rect, and Pencil. We could implement a separate subclass of Command for each of those, but it turns out that all of the commands to draw a shape are identical, other than that they point to a different shape. So we only need one command for drawing, called `DrawCommand` (note: you can just declare/implement all of your Commands in Command.h/cpp and similarly the same for all your Shapes in Shape.h/cpp).

Create a DrawCommand class. Since `Command` takes in two parameters, you need to declare a constructor for `DrawCommand`. You also need to override the `Finalize`, `Undo`, and `Redo` functions since they're abstract in the base class. For `DrawCommand`, you'll also want to override Update.

For now, leave Undo/Redo empty Update should call `Command::Update` and then call `Update` on the `mShape` member (that you inherit from `Command`, since every command is associated with a shape). Similarly, `Finalize` just calls `Finalize` on the shape.

RectShape
Now make a Shape subclass called `RectShape`. For most shapes, you'll just need a constructor and to override the `Draw` function.

Notice that the Draw function takes in a `wxDC` – this is a "device context" that you can draw to. There's lots of different functions to draw that are outlined here. In a way, all our shapes are an implementation of the adapter design pattern.

You'll notice that one of the member functions for `wxDC` is `DrawRectangle`. There are three different versions of the function – use the one that takes in a `wxRect`. A `wxRect` can be constructed by two points – the top left and bottom right ones, which are members of shape.

PaintModel Updates

Now let's create commands. There's two parts to this. First, `PaintModel` needs functionality to manage the current active command. You could save the current active command in a `shared_ptr` to a `Command`. Then create four functions in `PaintModel`: `HasActiveCommand`, `CreateCommand`, `UpdateCommand`, and `FinalizeCommand`.

`HasActiveCommand` returns true if there's currently an active command.

`CreateCommand` takes in the `CommandType` and starting `wxPoint`. Use `CommandFactory::Create` to create the appropriate command. We haven't implemented the `CommandFactory::Create` yet, but you'll notice that the first parameter the function takes is a `shared_ptr` to the model. Use the `shared_from_this()` member function to get that.

The `UpdateCommand` function takes in a `wxPoint`, and calls Update on the active command.

Similarly, FinalizeCommand calls Finalize on the active command, and then clears the active command pointer (set it to `nullptr`).

You also need to update `PaintModel` to draw shapes when `DrawShapes` is called. Loop through your `mShapes` vector and call draw on every Shape `shared_ptr`.

CommandFactory

The `CommandFactory` is a factory method that creates commands. It takes in a `shared_ptr` to the model, a `CommandType`, and a start point. Based on the type, make the correct command.

Suppose the command is `CM_DrawRect`. In this case, create a `DrawCommand` `shared_ptr` (use `std::make_shared`) and a Shape `shared_ptr` (again use `std::make_shared`).

Next, add the shape to the model (via the `AddShape` function).

PaintFrame – Hooking up Mouse Clicks/Move

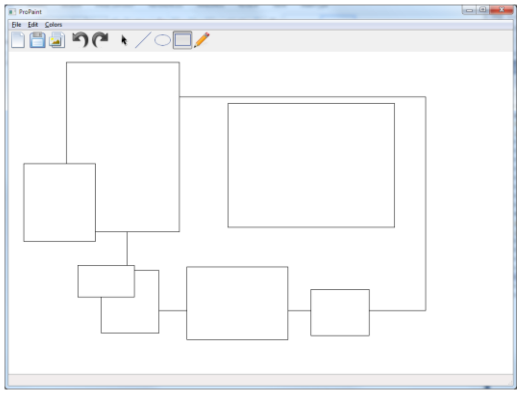
The last thing is to hook up what happens when a mouse is clicked/move. The `PaintFrame` uses an `mCurrentTool` variable to track the currently selected tool (the code for tool selection is given). The `OnMouseButton` function is called when the left mouse button is pressed or released. The "if" check in this function determines whether it's a press or release.

If the left mouse is pressed, and the `mCurrentTool` is `ID_DrawRect`, create a `CM_DrawRect` command via the model's `CreateCommand` function (access the model via `mModel`). Use `event.GetPosition()` to get the position of the mouse click. Whenever you change the model, you need to redraw the view. You can do this by calling `mPanel->PaintNow()`.

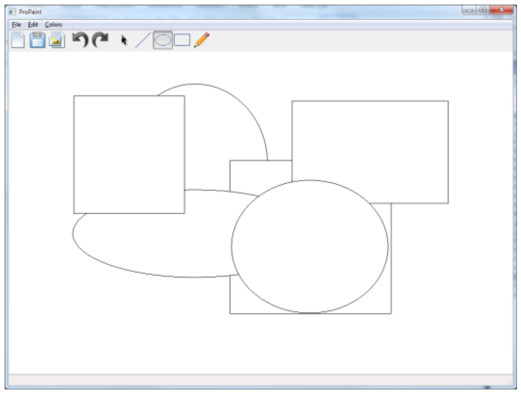
If the left mouse button is released, and there is an active command in the model, update the command one last time and then finalize it. And of course, paint!

Finally, `OnMouseMove` is called when the mouse moves in the drawing area. If there's an active command, update it (and paint).

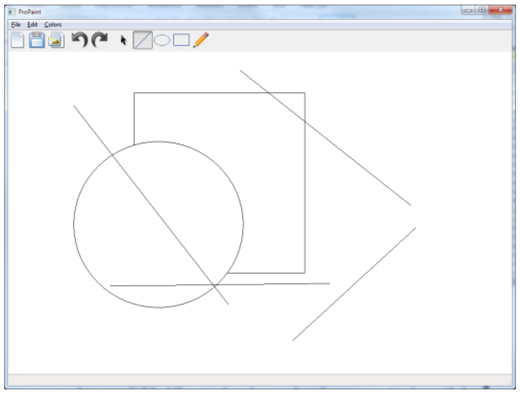
You should now be able to draw rectangles when you select the rectangle tool from the toolbar:



Ellipses
Ellipses work like Rectangles. Use `dc.DrawEllipse` to draw one. Create an `EllipseShape` class, add the code to create the command in the `CommandFactory`, and make sure the frame issues the `CM_DrawEllipse` if the `mCurrentTool` is `ID_Ellipse`. You should then be able to draw ellipses.



Lines
Lines are different in that when drawn, draw with the start point and end point – don't use the top left bottom/right bounds. Everything else is like Ellipse/Rect. Use `dc.DrawLine`. Yay, lines!



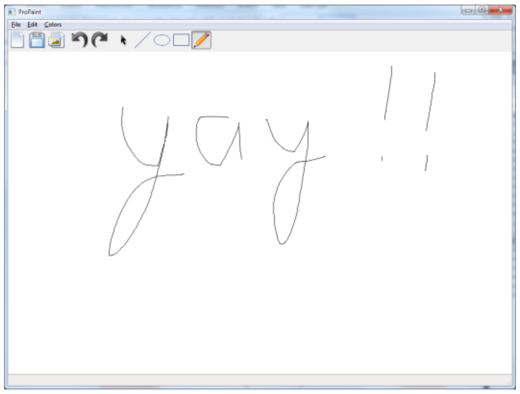
Pencil

The `PencilShape` is a bit more complex. Add a vector of `wxPoints` as a member variable, to save all the points that the user draws. The constructor then adds the first point to the vector. You also need to override Update and Finalize. Update should call `Shape::Update` first, then add the `newPoint` to the vector.

The finalize function is slightly more complex. Loop through all of the points in the vector, and compute mTopLeft and mBotRight from all the points. Keep in mind that you want a box around all of the points drawn, so it's possible that `mTopLeft` and `mBotRight` are not actually equal to one of the points (for example, the top left may have the x from one point by a y from another one).

To draw the pencil shape, use `dc.DrawLines` which takes in the number of points and a pointer to the points. (Note: If you have only one point in your point vector, then use `dc.DrawPoint`).

You should now be able to draw with the pencil.



Part 2: Undo/Redo and File>New

For undo/redo, you will create two stacks in `PaintModel` – an undo stack and redo stack. When finalizing a command, push it on the undo stack. Then to "undo" a command, you pop it off the undo stack, do whatever the command does when it is "undone" and push it onto the redo stack.

To “redo” a command, you pop it off the redo stack, do whatever the command does when it’s “redone” and push it onto the undo stack. In this way you could keep undoing/redoing the same command over and over.

First, add these stacks to the `PaintModel`. They should contain shared pointers to the command. Then when finalizing a command, push it onto the undo stack. Then add `CanUndo`/`CanRedo` functions to `PaintModel`, which should return true or false if you can or can’t undo/redo (you can’t if there’s nothing on the corresponding stack!).

Then also add `Undo` and `Redo` functions to paint model, which perform the behavior outlined in the first paragraph of this section.

Next, we need to tell `DrawCommand` what to do when the command is undone or redone. For `DrawCommand`, if it’s undone, remove the shape from the model. If it’s redone, add it back to the model. This works because the `DrawCommand` holds a reference to the shape the whole time, so even though the shape isn’t in the model’s `mShape` vector anymore, it’s not deleted.

The functions that trigger undo/redo in the `PaintFrame` are the appropriately-named `OnUndo` and `OnRedo`. These can call the corresponding model functions and paint the panel.

However, if you try to run the program you’ll notice that the undo/redo menu options are greyed out. You need to programmatically enable these commands only when undo/redo is possible.

There are two different functions to call, one for the toolbar button, and one for the menu items. For the toolbar, use:

```
mToolbar->EnableTool(wxID_REDO, false);
```

For the menu, use:

```
mEditMenu->Enable(wxID_REDO, false);
```

The first parameter is the ID of the item. This will be either `wxID_REDO` or `wxID_UNDO`. The second parameter is a Boolean true/false if it should be enabled or disabled.

Call these at the appropriate times to enable/disable undo/redo. For example, at the end of `OnMouseButton` you can enable/disable them based on whether the model says you can undo or redo. Similarly, every time a command is undone or redone, you need to update whether you still can redo or undo. (It probably makes sense to put “update undo/redo buttons” in a separate function, and call that when you need to).

As a final piece of bookkeeping, make sure that every time you add a new command to the model, you clear the redo stack.

You should now be able to undo/redo commands using either the toolbar, the menus, or the keyboard shortcuts (which vary depending on the platform).

File>New
The “New” command triggers when the user selects either File>New or the leftmost tool on the toolbar (the blank page). If you look at the `OnNew` function in `PaintFrame`, you’ll see that it calls New on the model. However, that doesn’t do anything right now.

So fix `PaintModel::New` so that it clears out all the model data – any active commands, the shapes vector, and the undo/redo stacks should all be cleared, which will give you a fresh new model.

Part 3: Colors and Selections

Wx uses the `wxPen` class to control the outline of shapes – you can set the color and the width, among other things. The `wxBrush` class controls the fill color of shapes, among other things. You can set the dc to use a specific pen or brush using `SetPen` and `SetBrush`.

Add a `wxPen` and `wxBrush` member to `PaintModel` to track the current pen and brush. Then create functions to get/set the width and color of the pen, and the color of the brush. Use the wxColour class to save colors. Also add getter functions to get the whole pen and brush.

There are a few built-in pens and brushes, so you can use this to initialize the members initially. Use `wxBLACK_PEN` to get the default black pen and `wxWHITE_BRUSH` to get the default white brush. Note that both of these are actually pointers, so you’ll have to dereference them. Make sure you reset the pen/brush to these defaults in the New function, too.

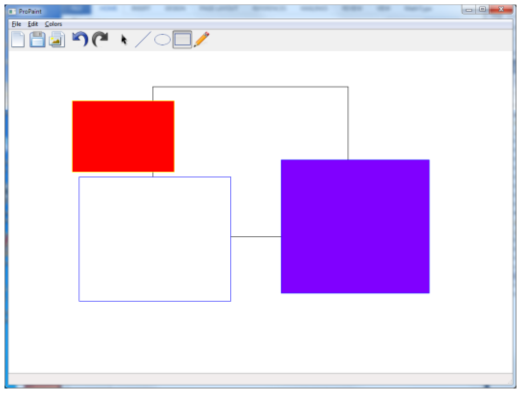
Now you also should add pen/brush member variables to the Shape base class, and getter/setter functions for them. Then all of your draw functions should call `dc.SetPen` and `dc.SetBrush` before they draw anything.

You also need to update the `CommandFactory` so it sets the shape’s pen/brush to the current one in the model. Note that after these changes, everything should still be drawing in their default colors.

Color Dialog Boxes
Conveniently, there’s a built-in dialog box for picking colors. The use of it is something like this:
```
wxColourData data;
data.SetColour(/*pass in current color*/);

wxColourDialog dialog(this, &data);
if (dialog.ShowModal() == wxID_OK)
{
	// Use dialog.GetColourData() to get the color picked
	 // and do something with it
}
```
The show modal call says to show the dialog box, and require the user to do something with it. The `wxID_OK` means that the user selected a color, so you should process it.

Use this dialog in `PaintFrame`'s `OnSetPenColor` and `OnSetBrushColor`, and hook it up to your model. If everything is in working order, you should be able to change the colors for the next shape you draw, which looks like this:



Pen Width
Next implement `OnSetPenWidth`. There's a `wxTextEntryDialog` that you can use to request the user to enter a string. Use this to request the user enter an integer value from 1-10. The documentation for this class is here.

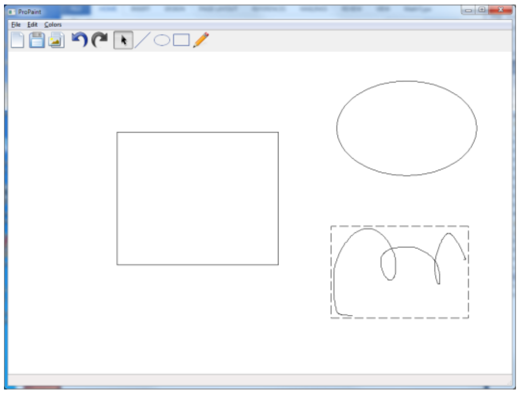
Remember, you need to validate whether the user entered an integer from 1-10. Only set the pen width in this case. In any event, you should now be able to change the width as well.

Basic Selection
It would be nice if you could change the colors of an existing shape. To do this, we need a way to select a shape first.

I'd first recommend adding a member function to Shape called `DrawSelection` – if a shape is selected, we'd like to draw a dotted line rectangle around the shape, so you know it's selected. You can draw this with the `wxBLACK_DASHED_PEN` and `wxTRANSPARENT_BRUSH`, respectively. Make sure it's a little bit larger than just the top left/bottom right, so even if you have a `RectShape`, you can see it selected.

The process of picking a selected shape should probably be implemented in `PaintModel`. So `PaintModel` needs some way to track the selected shape, if any (a shared pointer would be good), and also a function that given a point, tries to select a shape that intersects at that point. You want to select the topmost shape first, so you should reverse loop through the shape vector, and see if any intersect with the point. The first one that intersects should be selected. Then in `DrawShapes`, after all the shapes are drawn, you should call `DrawSelection` on the selected shape.

Finally, hook it up in `OnMouseButton` so if the current tool is `ID_Selector`, it tries to select a shape. You should be able to click on and select a shape now, which looks like this:



Note that if your selection for pencil shapes doesn't look right, it means you may have not computed the top left/bottom right points in finalize properly.

If you click where there isn't any shape, the selection should go away. Similarly, when "New" is called in `PaintModel`, the selection should also go away.

SetPenCommand and SetBrushCommand

Now that you have a selection, you can implement `SetPenCommand` and `SetBrushCommand`. The basic idea is if the user has a selected shape, and they change the pen color/width or the brush color, you should also change the pen or brush of the selected shape. This should automatically create and finalize these two new types of commands.

For both of these commands, you need to save the "old" and "new" pen (or brush) in order to support undo/redo for the commands. So if a `SetPenCommand` is "undone" go back to the old pen, for example.

So implement the commands, the code in the factory, and whatever extra glue, and you should be able to change the pen color/width or brush color of selected shapes, and undo/redo these changes.

Unselect

You should also hook up the Edit>Unselect command (`OnUnselect`). This should just unselect a shape, if there is one current selected. Note that this has to be enabled first (it's `ID_Unselect`). So when a selectin happens, enable it, if the selection goes away, disable the command.

DeleteCommand

Similarly, if a shape is selected, the user should be able to issue a `DeleteCommand` via Edit>Delete. This should remove the shape from the model. Note that the "undo" for this might be a little janky because you'll add a shape back on the top, and we have no way of specifying the depth. This is a minor oversight, but if you really want to fix it, you can also add a depth variable and sort shapes by depth, rather than the index in the array. (But you aren't required to do this if you don't want to).

Note that this also has to be enable first (`ID_Delete`).

MoveCommand
The `MoveCommand` is a bit more complicated. This allows you to move the selection around. If the user currently has a selection, and they mouse over that selection, you want to change the cursor to the `CU_Move` cursor (if they mouse off, it should change back to `CU_Default`). This is the standard "four arrows" cursor that indicates you can move the object.

When the `CU_Move` cursor is active, the user should be able to click to drag and move the object. This requires adding the new `MoveCommand`.

However, in order to support the idea of a move offset, you need to add some more code to Shape. Add a `wxPoint` to track the offset, which should just be initialized to 0, 0. Next, change `GetBounds` as well as your draw/draw selection functions, as necessary, to use the offset variable as part of their calculations. The idea is the `MoveCommand` won't change the actual values of `mStartPoint`, `mEndPoint`, etc., but just the offset. Note that `dc.DrawLines` takes an optional x and y offset as the 3rd and 4th parameters.

Finally, implement `MoveCommand` as well as the factory code. You'll need to override the `Update` function in `MoveCommand` so that the user can drag the object around and see it move.

Part 4: Exporting/Importing an Image

Finally, we want to be able to save our drawing to a standard file format, or load a standard image into our drawing. Luckily, wx makes this not too complicated.

Export
So the process for exporting is something like this:

1. Ask the user for a file to export to (standard file dialog box)
2. Create a bitmap in memory, and draw all the shapes to the bitmap
3. Save the bitmap to the specified file name, with the specified file format.

We'll allow the user to export PNG, BMP, or JPEG, since those are the most commonly used one. The export process unsurprisingly begins in `OnExport`.

To create the dialog box, you need to use a `wxFileDialog`. The documentation outlines how to create an "open file" dialog (which will be for import) and a "save file" dialog (what we want for export).

Note that you want the user to be able to pick whether they want to export .png, .bmp, or .jpeg/.jpg as appropriate.

Next, you should have a member function in `PaintModel` that takes in the file name as well as the size of the bitmap (which is a wxSize class). You just want to grab the size from the draw panel in this case, which you can do with something like `mDrawPanel->GetSize()`.

The function should look at the file name, and from the extension determine the correct `wxBitmapType` you are saving to – it should be either `wxBITMAP_TYPE_PNG`, `wxBITMAP_TYPE_BMP`, or `wxBITMAP_TYPE_JPEG`.

To create, draw to, and save the bitmap, you'll use code like this:

```
wxBitmap bitmap;
// Create the bitmap of the specified wxSize
bitmap.Create(size);

// Create a memory DC to draw to the bitmap
wxMemoryDC dc(bitmap);
// Clear the background color
dc.SetBackground(*wxWHITE_BRUSH);
dc.Clear();
// Draw all the shapes (make sure not the selection!)
DrawShapes(dc);
// Write the bitmap with the specified file name and wxBitmapType
bitmap.SaveFile(fileName, type);
```

Import

Importing also requires creating a file dialog (this time an "open" one). By default, it should show all .png, .bmp, and .jpeg/.jpg files.

The `PaintModel` should have a `wxBitmap` member to store the imported bitmap. By default, nothing will be loaded in it, which you can use the `bitmap.IsOK()` member function to determine – if it's not ok, nothing is loaded.

Then you could have a load function in `PaintModel` that first clears out the `PaintModel` via new, and then loads in the bitmap via `bitmap.LoadFile`.

Then your `DrawShapes` function should first decide if there's a loaded bitmap – if there is, draw that first before anything else. Don't forget that in `PaintModel::New`, you want to clear out the bitmap if one is loaded (just set it equal to a newly-constructed `wxBitmap`).

Grading Rubric

Part	**Points**
Part 1 – DrawCommand and Shapes	25
Part 2 – Undo/Redo and File>New	15
Part 3 – Colors/Selections	25
Part 4 – Export/Import	15
Your PaintModel test cases	10
Code Quality (including the style guide)	10
Total	**100**