CSCI3081S15-010-011-10

An-An Yu, Matthew Rahkola & Morgan Sieglaff

Self-Assessment:

An-An	***
Matthew	***
Morgan	***

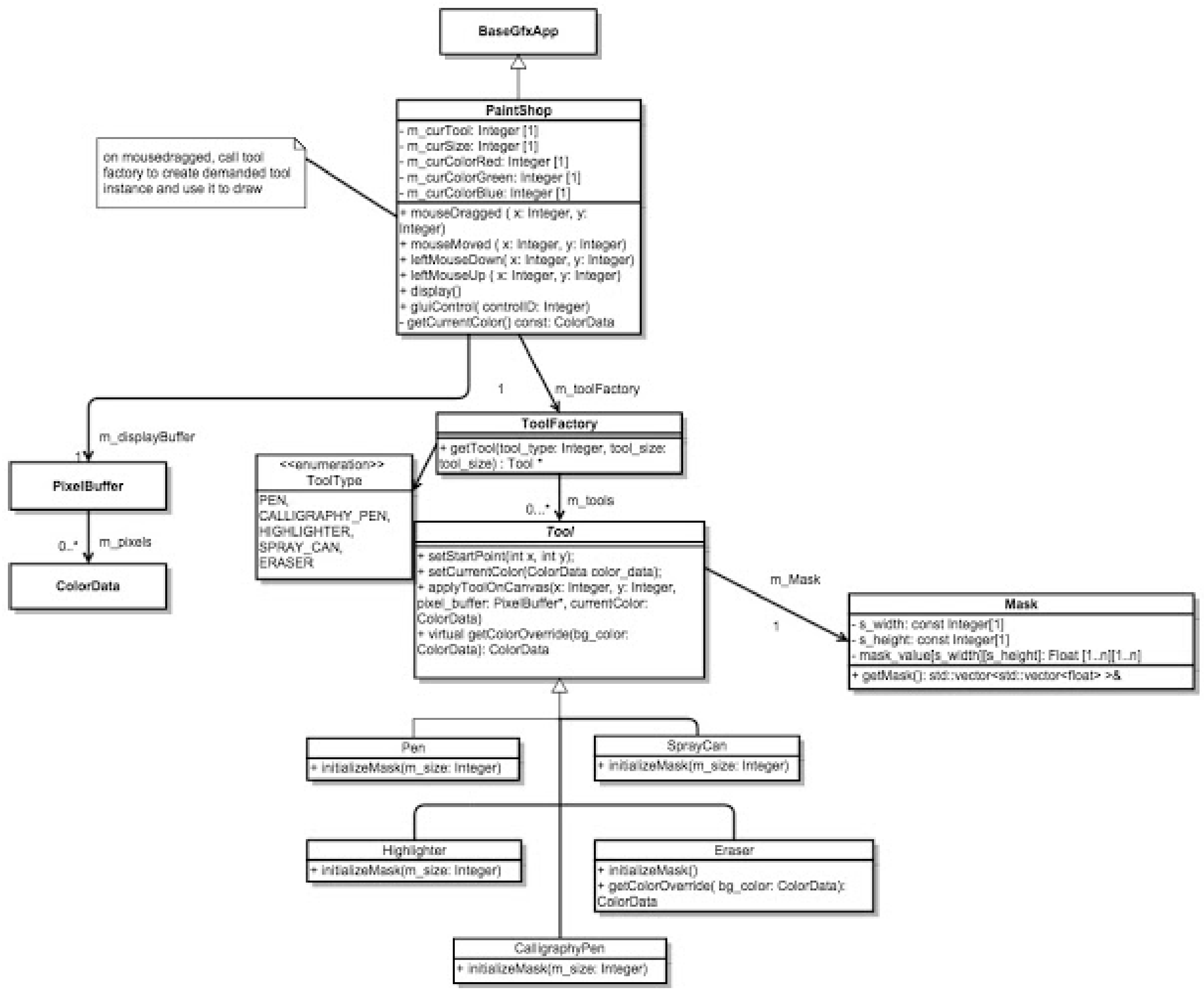
Github repository:

https://github.umn.edu/CSCI3081S15-010-011-10/PaintShop

Completeness:

We have met all the requirements of this iteration by implementing all tools specified as well as a special feature which allows the user to select three different sizes for each tool with the program feeling fast and responsive. We have discovered that GLUI has system dependencies that affects rectangular masks(Highlighter and Calligraphy Pen) by having an empty space in between the rectangle, this doesn't happen in our Mac OSX but other system dependent issues would also occur in different OS environments.

Simplified UML Diagram:


BaseGfxApp
PaintShop
- m_curTool: Integer [1]
- m_curSize: Integer [1]
- m_curColorRed: Integer [1]
- m_curColorGreen: Integer [1]
- m_curColorBlue: Integer [1]
+ mouseDragged (x: Integer, y: Integer)
+ mouseMoved (x: Integer, y: Integer)
+ leftMouseDown(x: Integer, y: Integer)
+ leftMouseUp (x: Integer, y: Integer)
+ display()
+ gluiControl(controlID: Integer)
- getCurrentColor() const: ColorData
on mousedragged, call tool factory to create demanded tool instance and use it to draw
m_displayBuffer
1
PixelBuffer
0..*
m_pixels
ColorData
1
m_toolFactory
ToolFactory
+ getTool(tool_type: Integer, tool_size: tool_size) : Tool *
<<enumeration>> ToolType
PEN,
CALLIGRAPHY_PEN,
HIGHLIGHTER,
SPRAY_CAN,
ERASER
0...*
m_tools
Tool
+ setStartPoint(int x, int y);
+ setCurrentColor(ColorData color_data);
+ applyToolOnCanvas(x: Integer, y: Integer, pixel_buffer: PixelBuffer*, currentColor: ColorData)
+ virtual getColorOverride(bg_color: ColorData): ColorData
m_Mask
1
Mask
- s_width: const Integer[1]
- s_height: const Integer[1]
- mask_value[s_width][s_height]: Float [1..n][1..n]
+ getMask(): std::vector<std::vector<float> >&
Pen
+ initializeMask(m_size: Integer)
SprayCan
+ initializeMask(m_size: Integer)
Highlighter
+ initializeMask(m_size: Integer)
Eraser
+ initializeMask()
+ getColorOverride(bg_color: ColorData): ColorData
CalligraphyPen
+ initializeMask(m_size: Integer)

Design 1: Using Factory Method Pattern

As a group, the most important design decision we made was to use a separate class, called ToolFactory to manage the different Tool Instances. Our main incentive to using this approach came from the idea that PaintShop should not depend on the subclasses of Tool. Using this method, we were able to improve the modularity of PaintShop because it is using only the public interface of Tool, rather than all of the subclasses. At a high level, this design implementation is important because PaintShop should not need to *maintain* the tools. There is no need for PaintShop to know when tools are created, or when the size of the tools change. All that is needed in the functionality of the program is to get requests from the user and to draw the appropriate response based on that request.

The second reason we found this design choice to be significant is because ToolFactory is able to manage various instances of tools by allocating them in the constructor and deleting them in the destructor. Because of this structure, it is not only reasonable, but also more efficient for PaintShop to defer to ToolFactory to handle the details of different tools, rather than try to tackle each instance on its own.

As is the case with all designs, there are also certain drawbacks that come along with having a ToolFactory class to manage the different instances of tools. For example, setting up PaintShop in this way creates a little more overhead when starting the program than other design implementations. This starting overhead is due to the fact that our approach begins the program by allocating and creating all possible tool instances so that the program can feel more responsive while in use. Another potential drawback that comes with having a separate ToolFactory class is simply that it requires using additional hierarchy. Because of this, we need more files for the program to work, which makes our design automatically a little less intuitive.

One alternative to the design implementation we chose would be to have PaintShop maintain instances of tools directly, such that we used a function called getTool(Tooltype t). This alternative approach would be beneficial to our program because it is such a straightforward idea that is easy to implement. Furthermore, it would give us less files and class hierarchies to deal with throughout the program. However, this alternative presents its own drawbacks as well. For example, if we didn't force PaintShop to only use Tool class's public interface, it could reflect low modularity. Beyond this, it could also be hazardous overall: if Pen had a method destroyOtherTools() that PaintShop called, this could cause problems with the functionality of the the program overall.

One more drawback to this alternative plan is that having such a large switch case inside PaintShop.cpp would initialize a specific mask for each case. This brute force method would then make the PaintShop.cpp file unnecessarily large. Beyond this, having all of the tool types in one place would make it impossible to model the characteristics between them all. For these reasons, we find our design implementation to be better than the alternative.

Design 2: Using Template Method Pattern

The second most important design decision that we decided to implement is that within our PaintShop program, the tool class uses a template method pattern.

The first reason for this implementation decision is that the use of a template method pattern increases our ability to reuse code segments. For example, each tool instance uses the same method to apply itself to the canvas--only their masks and color are different for each instance. Therefore, we are able to define the method applyToolOnCanvas() within the base class and let each subclass inherit this method, instead of having everyone implementing the identical method for themselves. This greatly reduces redundancy, which in turn helps prevent bugs within the program.

The second reason we believe that this is a good implementation for the project is because the behaviors that we have outlined pertaining to our five current tools can all be fully described by a Mask and a color override (which is only necessary for the Eraser because it is always white) . Therefore, their uniform behaviors allow them to be generalized using a template method pattern.

Although, admittedly, by using this template method we have less flexibility in the program. Namely, if we were to add new types of tools whose behaviors could *not* be described using masks and color override, then we would be forced to redefine our template of a Tool and modify each and every subclass that follows it.

An alternative to this would be to make our method applyToolOnCanvas() a virtual function. This would then increase the flexibility of our program overall by allowing us to add new types of tools in the future, without requiring that they need to use a mask or color override to affect the canvas. In this way, we could declare various types of tool behaviors and then let each of the subclasses decide which behaviors are appropriate to adopt-- similar to our class duck example with flyBehaviors and quackBehaviors.

Although, if we did decide to use this alternative instead of our current template method, it would result in our program having higher performance costs. This is because in C++ virtual functions require additional lookups at run time, which we are able to avoid with our current implementation. Therefore, in the end, this alternative would be more costly to implement in our PaintShop program when compared to template patterns.

Yet another alternative design that we could have implemented would be to merge our ToolFactory and Tool classes into one class. In doing this, our Tool class would handle both instance management and providing interfaces for each of the subclasses. For instance, we could allow Tool to keep a static vector of tool instances, provide a static method Tool* getTool(ToolType t) and the usual interface methods of Tool. However, our Tool::getTool(ToolType t) would then depend on its subclasses, while each subclass, in turn, would depend on the Tool class, thus creating a circular dependency. We didn’t feel comfortable dealing with this potentially complex issue, so we opted for a cleaner, more efficient design.