3F6 - Software Engineering and Design

Handout 7

Design Patterns (II)

With Markup

Ed Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

Observer Pattern

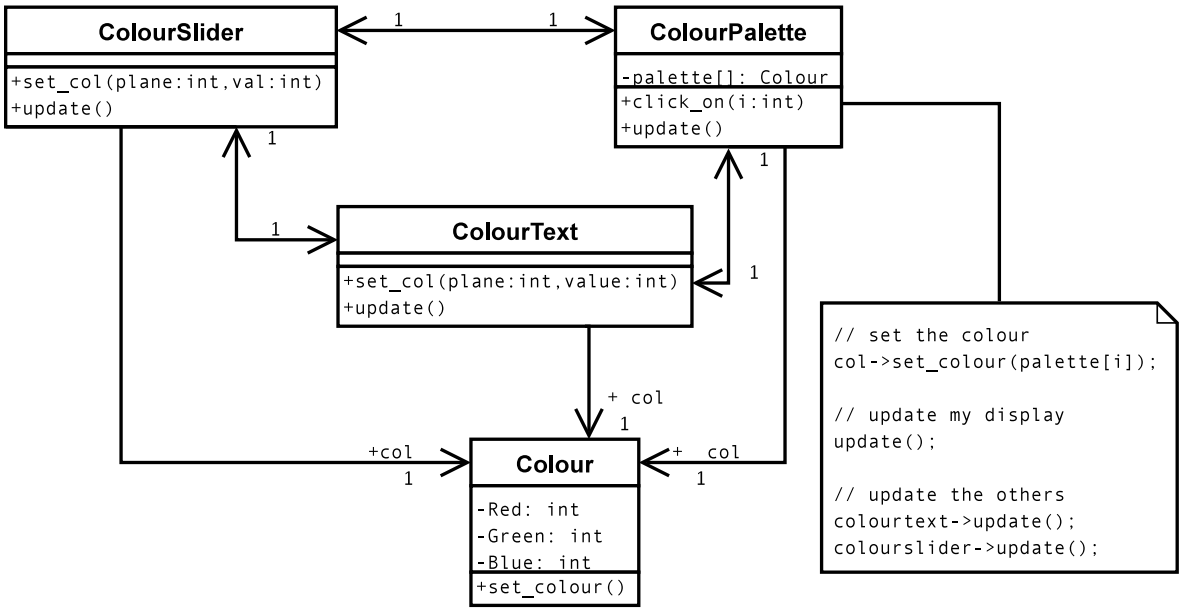
Problem

We want our drawing editor to be able to store and display coloured objects and we want to support multiple methods of choosing the pen and fill colours for a shape in the drawing. These include typing in values for Red, Green and Blue, selecting a colour from a palette, or using colour sliders.

Multiple interfaces for selecting a colour may be visible on the screen simultaneously and we want to arrange the software so that as the user makes a choice using one of the interfaces (e.g. by clicking on a colour in the palette), all the other interfaces update their displays to reflect the current choice of colour.

Solution 1

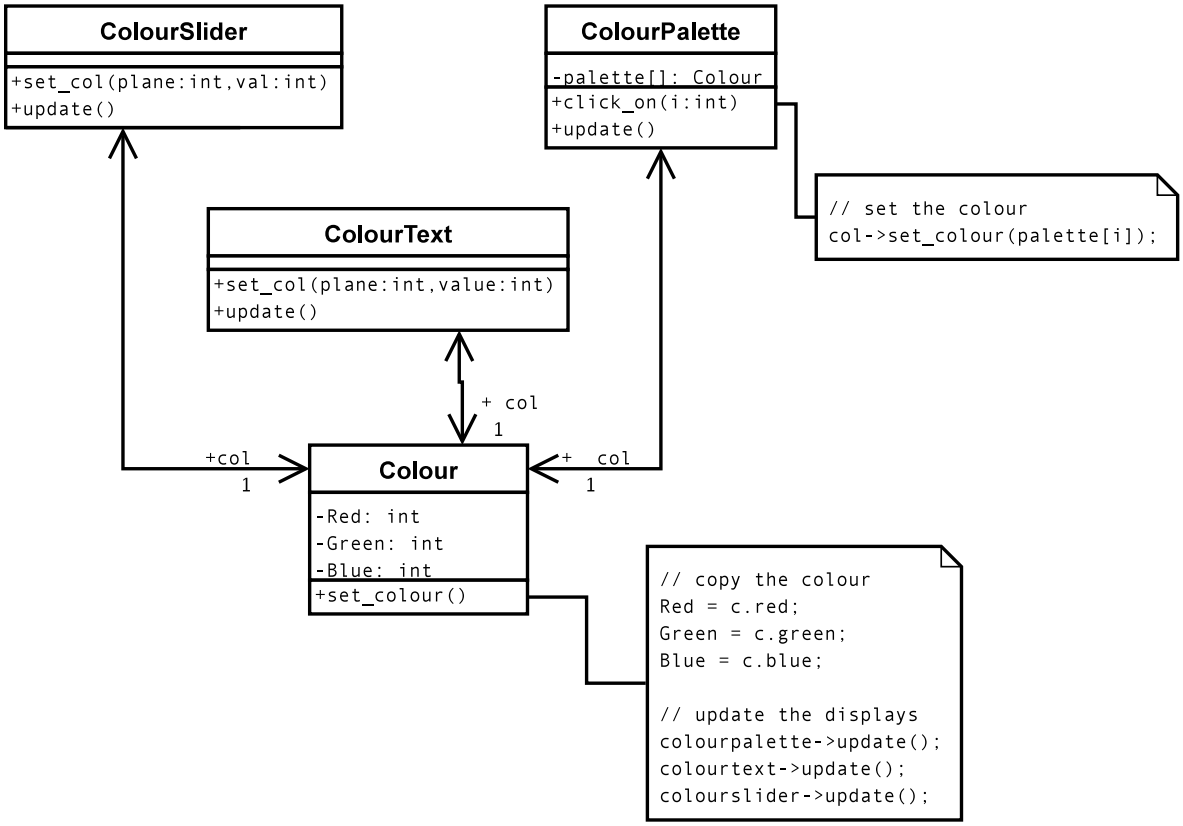
Add code into each colour selection method to update all the other displays.



Pros:	-	simple and efficient for small systems
Cons:	-	since every interface must link to every other interface, the code for all interfaces must be updated when a new interface (eg a colour wheel) is introduced. Quadratic growth!

Solution 2

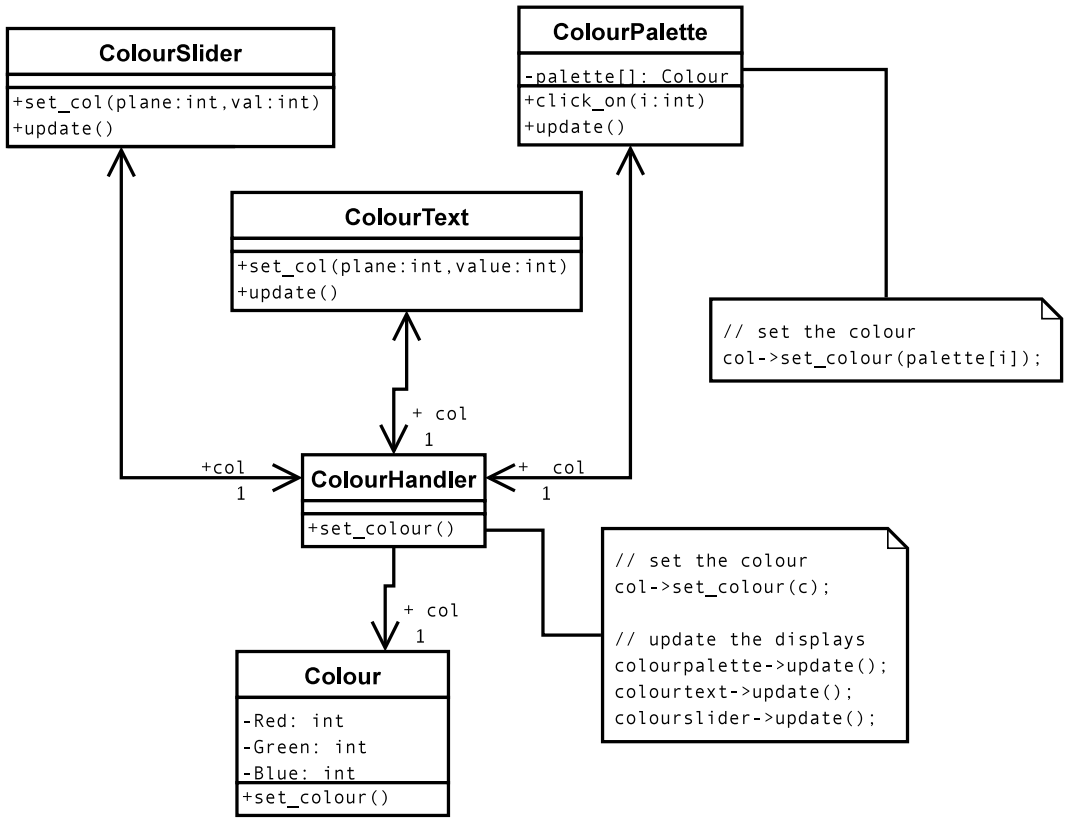
Delegate the updating task by moving the code that updates the colour selectors into the `Colour` class.



Pros:	-	coding and maintenance simplified linear growth
Cons:	-	clutters `Colour` objects with update code
	-	'Leaky' design
	-	`Colour` type no longer suitable for declaring general purpose colour variables

Solution 3

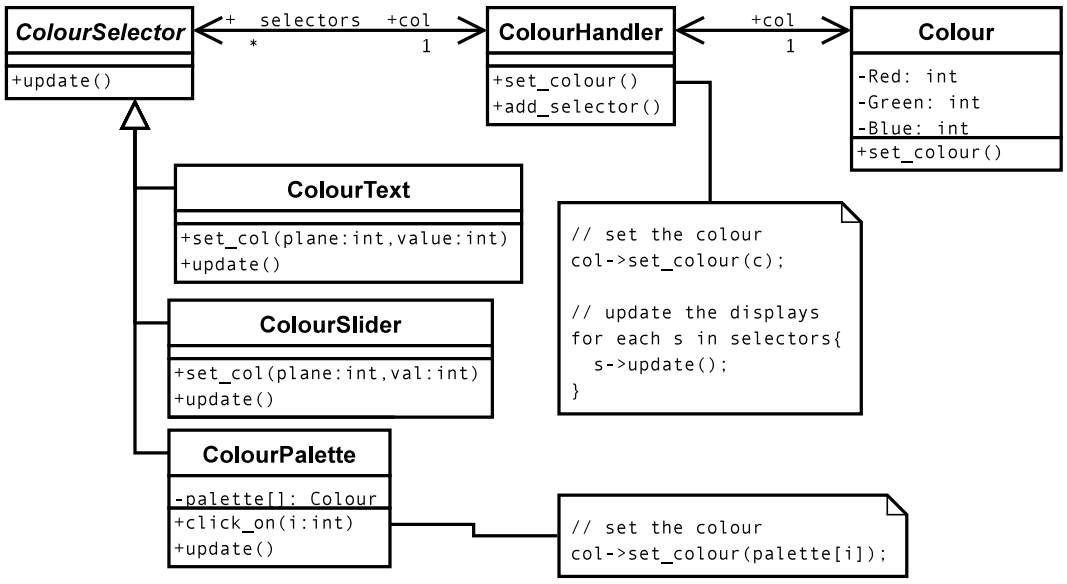
Separate out the `Colour` class from the class that handles updating the displays i.e. introduce an intermediary into the previous solution which keeps `Colour` simple and allows the software to still handle the process of notifying the selection methods to update themselves.



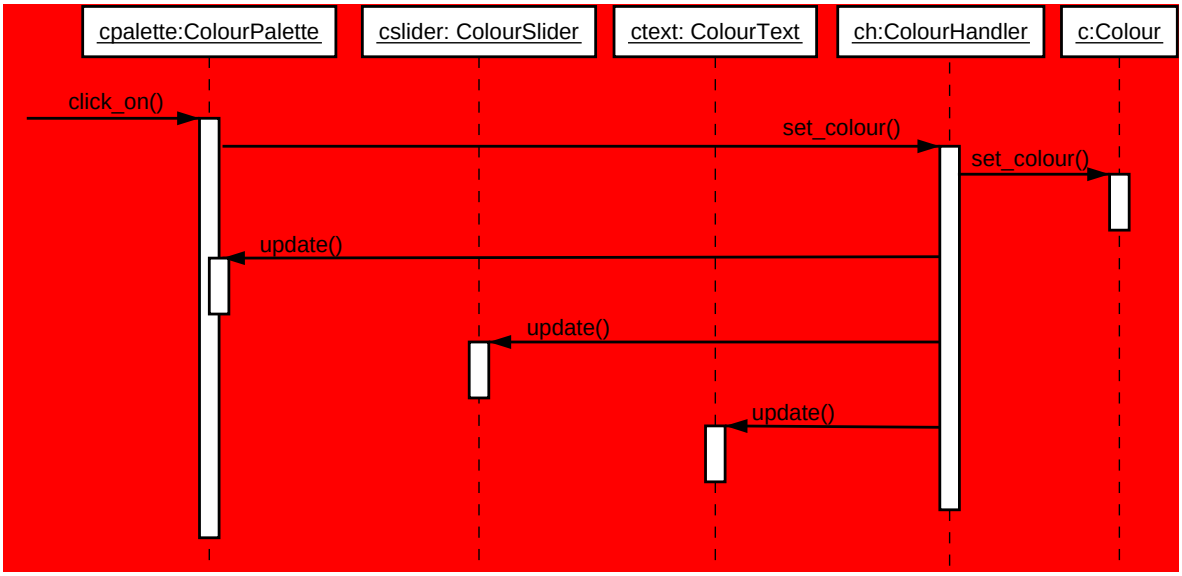
Pros:	-	`Colour` class decluttered and clean separation between representing colours and updating the displays
Cons:	-	`ColourHandler` still needs explicit references to every colour interface object in order to update it.

Good Solution

We can remove this dependency by creating an abstract base class to represent the concept of a colour selector. The colour handler class can then merely maintain a list of colour selectors without worrying about what kind of colour selector they are.



This leads to the following behaviour when the user clicks on a colour in the colour palette:



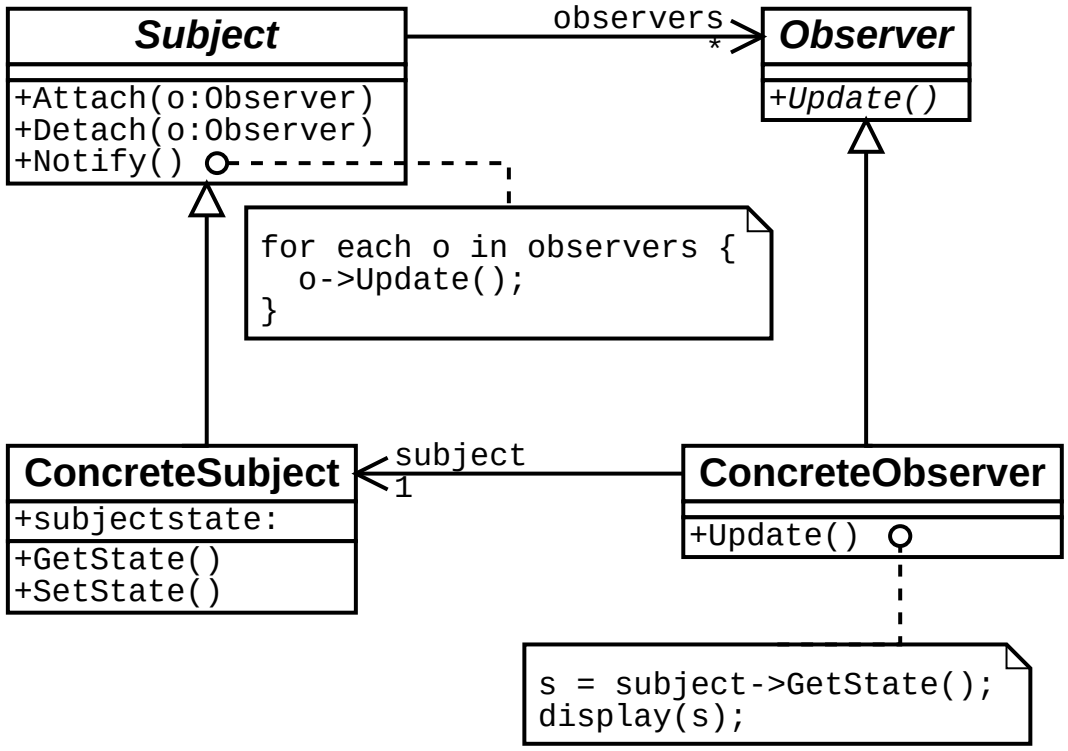
Code Example

```
class ColourSelector {
 public:
  virtual void update(Colour c)=0;
  ColourHandler* ch;
};

// ----------------------------------------

class ColourHandler {
 public:
  set_colour(Colour c);
  add_selector(ColourSelector* s);
  remove_selector(ColourSelector* s);
 private:
  Colour * col;
  set<ColourSelector*> selectors;
};

ColourHandler::set_colour(Colour c){
  col->set_colour(c);
  set<ColourSelector>::iterator iter;
  for(iter = selectors.begin();
      iter != selectors.end(); iter++){
    (*iter)->update(c);
  }
}

ColourHandler::add_selector(ColourSelector* s){

  s->ch=this;  // assign the selector a colour handler
  selectors.insert(s);  // and record selector

}
```

```
class ColourPalette : public ColourSelector {
 public:
  virtual void update(Colour c);
  void click_on(int i);
 private:
  int current_cell;
  vector<Colour> palette;
};

ColourPalette::update(Colour c){
  // update palette colour selector with current colour
  if (palette[current_cell]==c) highlight(current_cell);
  etc
}

ColourPalette::click_on(int i){


  current_cell = i;
  ch->set_colour(palette[i]);


}
```

The Observer Design Pattern

This is also sometimes known as the Model-View-Controller (MVC) pattern. The key idea is that it separates the model (or document (or colour)) from the user interface display of that state. The model only needs to know that it has a set of observers, not the details of each observer.



Disadvantages

This pattern can lead to a large amount of computational overhead. For example consider gradually moving a slider bar in the colour selector example. This will generate several `set_colour` calls to the `ColourHandler` which in turn will generate n times that many `update` calls to the n colour selectors.

Abstract Factory Pattern

Problem

By using the Observer pattern, we are able to separate the representation (underlying model) from the user interface. Further, the model does not need to know anything about the user interface (indeed, we could use this approach to design our application to run in batch mode without ever invoking a user interface).

For example, when using a Unix machine we create X windows interface objects unless the system is running Gnome in which case we create GTK+ objects. If we are using a Windows machine, we create MFC based interface objects.

But how do we create these platform specific interfaces?

Solution 1

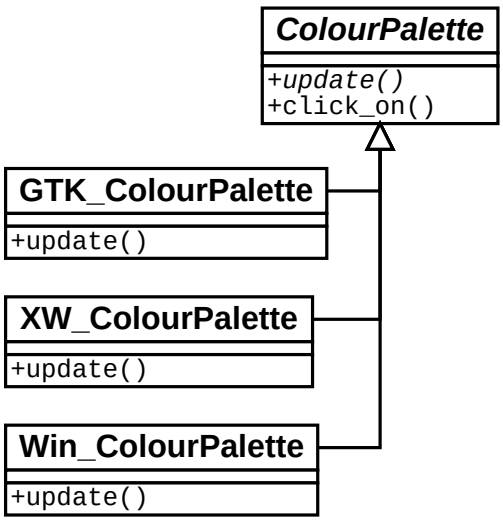
At every place in the code where we create a graphical interface component, we test for the machine type and create the appropriate object:

```
// create a colour palette selector
#ifdef __UNIX__
#ifdef __GTK__
GTK_ColourPalette* p = new GTK_ColourPalette;
#else
XW_ColourPalette* p = new XW_ColourPalette;
#endif
#else
#ifdef __WINDOWS__
Win_ColourPalette* p = new Win_ColourPalette;
#else
#error unknown system type
#endif
#endif
```

Pros:	-	very simple for small problems Does this sound familiar?
Cons:	-	since the code to handle multiple platforms is strewn throughout the whole program, an excessive amount of work will be needed to support a new platform
	-	a system of any complexity would be virtually unmaintainable

Solution 2

We could localise the behaviour by deriving the classes `GTK_ColourPalette`, `XW_ColourPalette` and `Win_ColourPalette` from a common base class `ColourPalette`. Then we can define a single function to create the various kinds of colour palette.



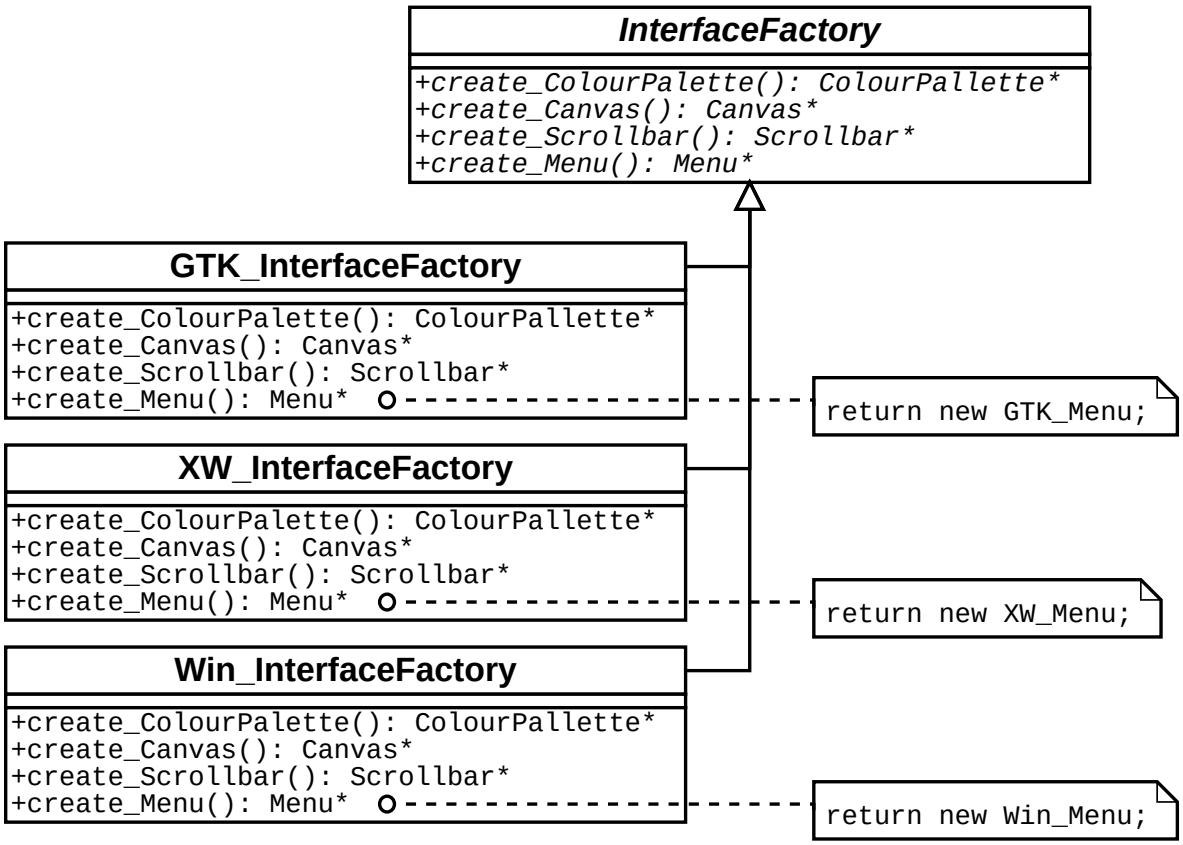

```
// create a ColourPalette
ColourPalette* create_colourpalette(){
#ifdef __UNIX__
#ifdef __GTK__
  return new GTK_ColourPalette();
#else
  return new XW_ColourPalette();
#endif
#else
#ifdef __WINDOWS__
  return new Win_ColourPalette();
#else
#error unknown interface type
#endif
}
```

Pros:	-	some localisation of platform dependent code achieved since most of the system will interact only with the platform independent `ColourPalette` class
Cons:	-	still requires `#ifdef` code for every widget create routine
	-	still difficult to maintain

We can improve it further by putting the functions in different files. But this becomes an ad-hoc implementation of...

Good Solution

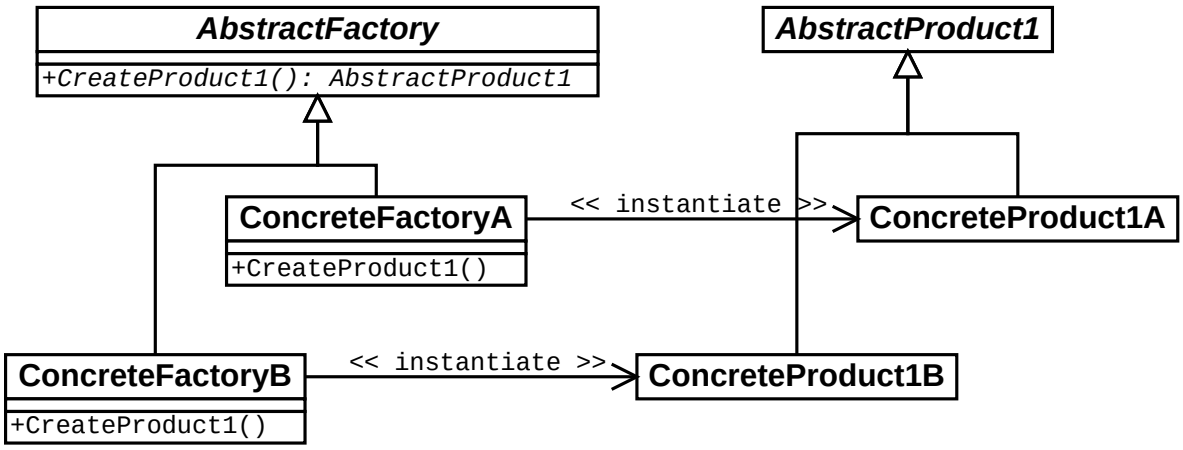
We can collect all the platform dependent code into a set of classes, one for each platform, derived from a common base class. Each of the platform specific classes knows how to create all the user interface components for that platform and the base class provides virtual functions for creating abstract components:



This approach allows us to eliminate all of the platform dependent code (all the `#ifdef`s) except for a single instance where we decide which kind of `InterfaceFactory` to make.

The Abstract Factory Design Pattern

This pattern separates the choice of product line from the choice of producer by creating abstract concepts for both. This allows new producers to be created very easily since only a new subclass of `AbstractFactory` is needed to create the new set of products (e.g. user interface components for MacOS).



Disadvantages

Although creating new factories is easy, creating new product lines is harder - the interface to `AbstractFactory` and every subclass of it needs to be updated to include a function to construct the new kind of object.

Also, this approach results in large numbers of virtual functions in the code since nothing really knows the type of objects it is dealing with. This can result in a significant performance degradation if such calls are made frequently.

Proxy Pattern

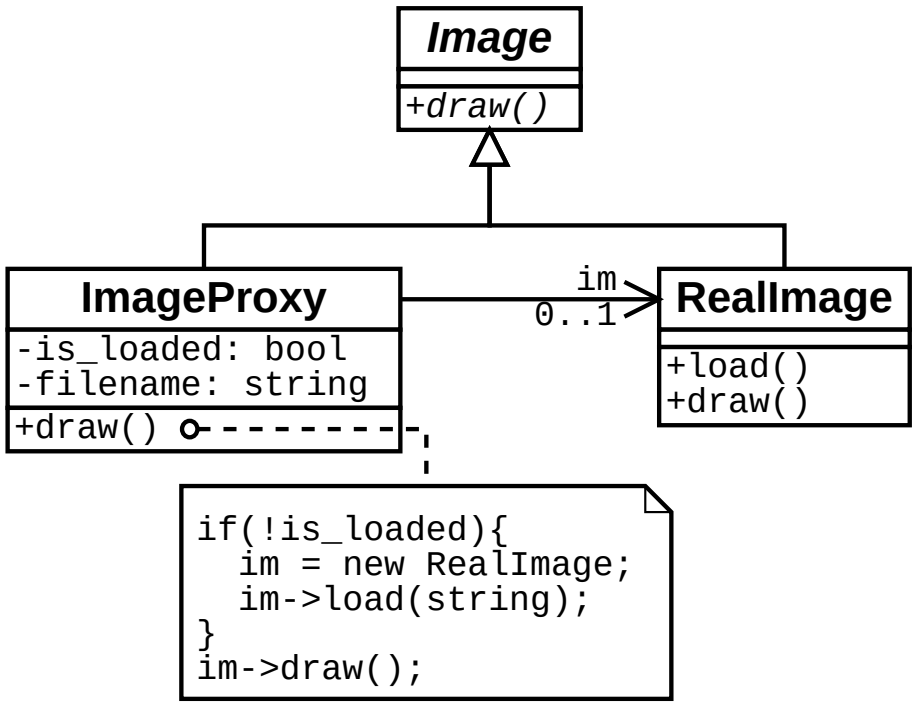
Problem 1

Large documents in a drawing editor containing many images may result in very large files on disk that take a long time to load (e.g. a large Powerpoint file).

Not all images will be needed immediately and could be loaded on demand when they are needed. But what can be used in place of the image until it has been loaded?

Good Solution

Create a class that provides exactly the same interface as `Image` and use that class to stand in for real images:

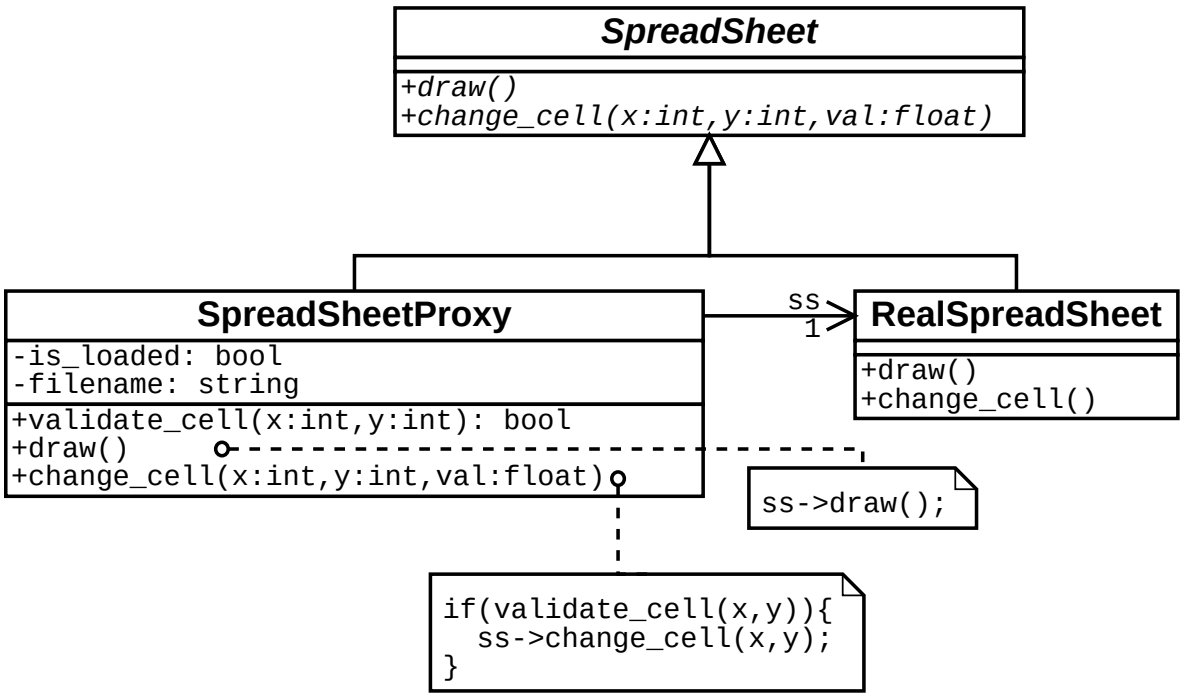


Problem 2

We may wish to embed spreadsheets into the drawing editor. These may be embedded so that subsequent users of the editor are restricted from changing some or all of the values in the spreadsheet.

Same Good Solution

We can create a class that provides exactly the same interface as the spreadsheet, but checks to see if the user is allowed to change the value of a cell before calling the function in the real spreadsheet.

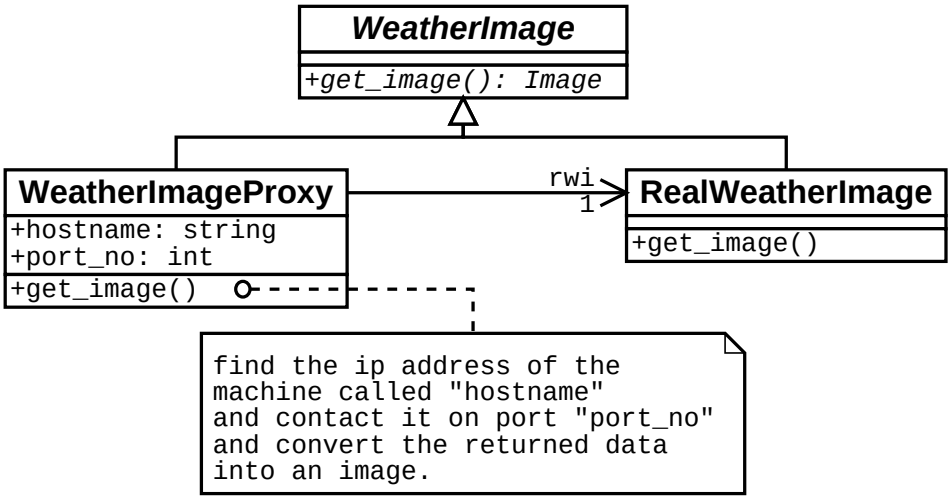


Problem 3

We may have a service running on a remote machine which we wish to access from a drawing editor. For example, we may have a computer which can access a weather satellite and can serve satellite images to us. In this case, we want the document to always display the latest satellite image and to update itself at regular intervals.

Same Good Solution Yet Again

The same abstract base class is used in both the weather satellite program and the drawing editor program. In the satellite program we use an object of one subclass and in the drawing editor we use an object of a different subclass. The object in the drawing editor pretends to be exactly the same as the object in the weather program, but fulfills its functionality by contacting the real object over the computer network and receiving the results of the operations in the same way.



The Proxy Design Pattern

This pattern is used in three distinct ways known as virtual, security and remote proxies (corresponding to the three examples above, respectively).

The generic idea is that the software contains a class that pretends to be another class and objects of the proxy class carry out their functionality by contacting the real object (whether that is on disk, in the same program space or on another computer on the other side of the planet.

The last version of this pattern, the remote proxy is the key technique used in CORBA and will be discussed in more detail in later lectures.

Disadvantages

A minor problem is that using proxies can lead to confusion over identity. We may have a real object `r` and a proxy for that object `p` but if we test `(p==r)` we will get `false`. Similarly two proxies for the same real object are distinct objects. It is, however, possible to design additional code to avoid these problems.