TASKMASTER:
An Architecture Pattern for GUI Applications

Robert C. Martin

James W. Newkirk

Bhama Rao.

Introduction

This paper describes the TASKMASTER architecture pattern for developing complex GUI applications. This pattern employs the principles of OOD to guide developers in creating GUI architectures that are flexible, easy to maintain, and are reusable. In this architecture we will find elements of Document/View and Model/View/Controller, thus it builds upon the successful architectures of others. However, you will also find that a number of problems inherent in those architectures have been addressed in a simple and easy to implement way.

The requirements for most GUI applications are usually very volatile. Users always want new features and extra ways of looking at and manipulating the data. Thus the software is always changing, sometimes in unexpected ways. For this reason, the software architecture of GUI applications should be very flexible and robust. It must allow changes to be made easily, and it must be highly decoupled so that when those changes are made they have a minimum effect. Otherwise, before too long, the continuous changes will cause the software to degrade into an unmaintainable morass.

The TASKMASTER architecture pattern helps to prevent this degradation, and provides for high levels of modification, maintenance, and reuse by creating high level abstractions that provide isolation boundaries between the various functions of the GUI. With this isolation in place, one can easily change the manner in which a particular object is drawn, or the way a certain field is edited, without affecting or recompiling other parts of the system. One can change the interactions that drive the creation and manipulation of objects on the screen without affecting any other parts of the software. One can add new GUI objects without affecting any of the code that manipulates the already existing objects. And one can also reuse the screen objects independently of the interactions that create and manipulate them.

The Roots

The TASKMASTER architecture has grown from the work that we at Object Mentor Inc. have done with some significant GUI applications we have been working on for the last several years. These applications are complex drawing tools that allow users to draw diagrams of many different sorts. The entities in these diagrams are knowledgeable of each other and have well defined relationships. Thus, there is a great deal of intelligence surrounding the way that they are manipulated and drawn.

The platform for this project is Windows 95, and the development tool is Borland C++ 4.5 using the OWL framework. However, the examples in this article will all be done using Visual C++ 4.2 and MFC. The fact that we can so easily change compilers and frameworks supports our claim that TASKMASTER is an architecture *pattern*, and not a platform specific architecture.

Resources for this article.

The code examples presented in this article are snippets from actual applications that was developed along with this article. You can download the actual source files for these applications from `http://www.oma.com/C++Report/TaskMaster/Examples`.

The diagrams used in this article conform to UML 1.0. You can download the description of this powerful design notation from `http://www.rational.com`.

The Problem

Consider a simple GUI application which allows users to draw a series of lines on a blank canvas. These lines are drawn when the user clicks a mouse button on a point, drags to another point, and then releases the mouse button. During this interaction the screen shows a “rubber-band” line which follows the mouse until the button is released. Once the mouse button is released the line remains on the screen and is added to a list of lines which the window maintains. We call this part of the applications the *task*. We can use this task to draw many lines on the screen.

Now consider what happens when we put another window on top of our window, and then move it away again. The lines that had been covered up need to be redrawn. The window that contains the lines receives a `PAINT` message. In response to this message it looks through its list of lines and redraws them all.

In simple form, such an application might look like listings 1 and 2. Here we see some of the code for controlling this simple application living inside the class `LineApplicationWindow`.

Listing 1: LineApplicationWindow.h

```
class LineApplicationWindow : public CFrameWnd
{
  public:
   LineApplicationWindow(GraphicFactory* aFactory);
   virtual ~LineApplicationWindow();

  private:
   CPoint          itsSecondPoint;
   CPoint          itsFirstPoint;
   GraphicFactory* itsGraphicFactory;

   vector<GraphicObject*> itsObjects;

   CPen* itsBluePen;

   afx_msg void OnMouseMove(UINT nFlags, CPoint point);
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
   afx_msg void OnPaint();

    DECLARE_MESSAGE_MAP()
};

#endif
```

Listing 2: LineApplicationWindow.cpp

```
LineApplicationWindow::LineApplicationWindow(GraphicFactory* aFactory)
: itsGraphicFactory(aFactory)
{
   Create (NULL, “Line Application”);
```

Listing 2: LineApplicationWindow.cpp (Continued)

```
   itsBluePen = new CPen(0,1,RGB(255,255,0));
      // when the pen is drawn in XOR mode it will be blue
}

LineApplicationWindow::~LineApplicationWindow()
{
   // delete the object stored in the vector
   vector<GraphicObject*>::iterator index;
   for(index = itsObjects.begin(); index != itsObjects.end(); ++index)
   {
      GraphicObject* anObject = *index;
      delete anObject;
   }

   delete itsBluePen;
}

void LineApplicationWindow::OnPaint()
{
   CClientDC dc(this);

   vector<GraphicObject*>::iterator index;
   for(index = itsObjects.begin(); index != itsObjects.end(); ++index)
   {
      GraphicObject* anObject = *index;
      anObject->Draw(dc);
   }
}

void LineApplicationWindow::OnLButtonUp(UINT nFlags, CPoint point)
{
  if(GetCapture() == this)
  {
    itsSecondPoint = point;

   CClientDC dc(this);

   GraphicObject* anObject =
          itsGraphicFactory->MakeLine(itsFirstPoint,itsSecondPoint);

    anObject->Draw(dc);
   itsObjects.push_back(anObject);

   ReleaseCapture();
  }

  return;
}


void LineApplicationWindow::OnLButtonDown(UINT nFlags, CPoint point)
{
  SetCapture();
  itsFirstPoint = itsSecondPoint = point;
}

void LineApplicationWindow::OnMouseMove(UINT nFlags, CPoint point)
{
   if(GetCapture() == this)
   {
      CClientDC dc(this);
```

Listing 2: LineApplicationWindow.cpp (Continued)

```
      // save the current dc parameters that we are changing
      int previousROP = dc.SetROP2(R2_XORPEN);
      CPen* currentPen = dc.SelectObject(itsBluePen);

      // draw the previous line
      dc.MoveTo(itsFirstPoint);
      dc.LineTo(itsSecondPoint);

      // draw the new line
      dc.MoveTo(itsFirstPoint);
      dc.LineTo(point);
      itsSecondPoint = point;

      // reset previous parameters
      dc.SetROP2(previousROP);
      dc.SelectObject(currentPen);
   }

   return;
}
```

The simplicity and elegance of this program is compelling. (See Figure 1.) Everything is pretty straightforward. When the left button goes down (`OnLButtonDown`), we capture the cursor (so that we know if it leaves our window) and record the point at which the button was clicked. For every mouse move event (`OnMouseMove`) we erase the current line and redraw it in the new position. (This is done by drawing in XOR mode. If you don't understand this, don't worry about it.) Then we record the new position. When the left button finally comes back up (`OnLButtonUp`) we create an instance of a `GraphicLine` from the `GraphicFactory` and place it in the vector that holds the list of lines. Upon reception of a PAINT event (OnPaint) we simply iterate through the list of `GraphicObjects` telling each one to draw.

Document/View

The simplicity of this program masks some potential problems. For example, what if we wanted to show two windows. One with the lines drawn as before, but another text window displaying a scrolling list of lines in the following format:

```
Line((0,0),(1,1));
Line((5,2),(8,3));
```

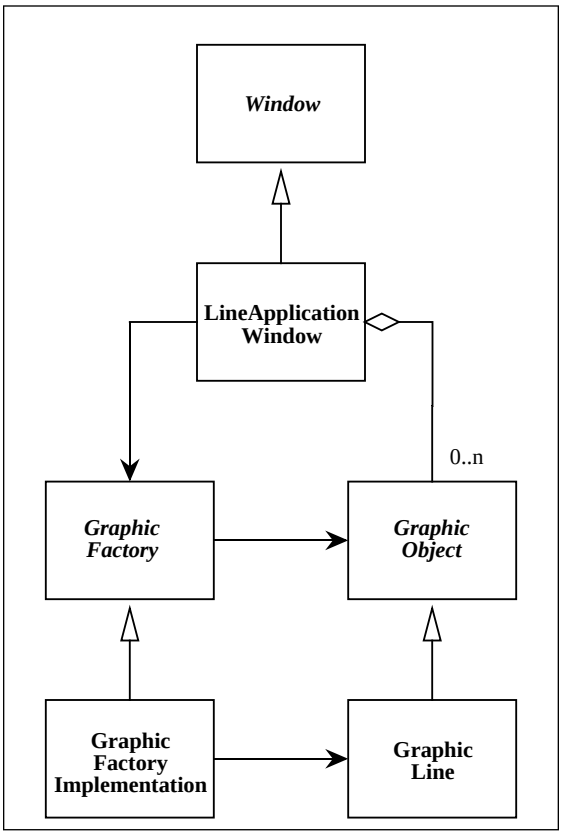
Each time a new line was added to one window, it would appear on the other. Clearly we are not currently set up to do that.

Or, what if we wanted to store all the created lines into a file and then read them back later? Presumably we could do this by adding the appropriate `Save` and `Store` methods to the `LineApplicationWindow` class. However, why should those methods be in the same class with the event functions such as `OnLButtonDown`? We would rather that the code that knew how to read and write lines to a file was reusable separately from the code that knows how to create and display the lines.

This is an instance of a violation of the Open Closed Principle (OCP)[1]. By definition the window class cannot be closed to gross changes in the way that the data is displayed. However, such

1. The Open Closed Principle, Robert C. Martin, C++ Report, January 1996

Figure 1:



changes should not affect the way that the data is stored or retrieved. By putting both functions together in the same class, we find that we cannot close data manipulation functions against changes in the way that the data is displayed. By the same token, we cannot close the code that manages the display of the data against changes in the way that the data is stored and manipulated. Clearly some kind of separation of concerns is needed.

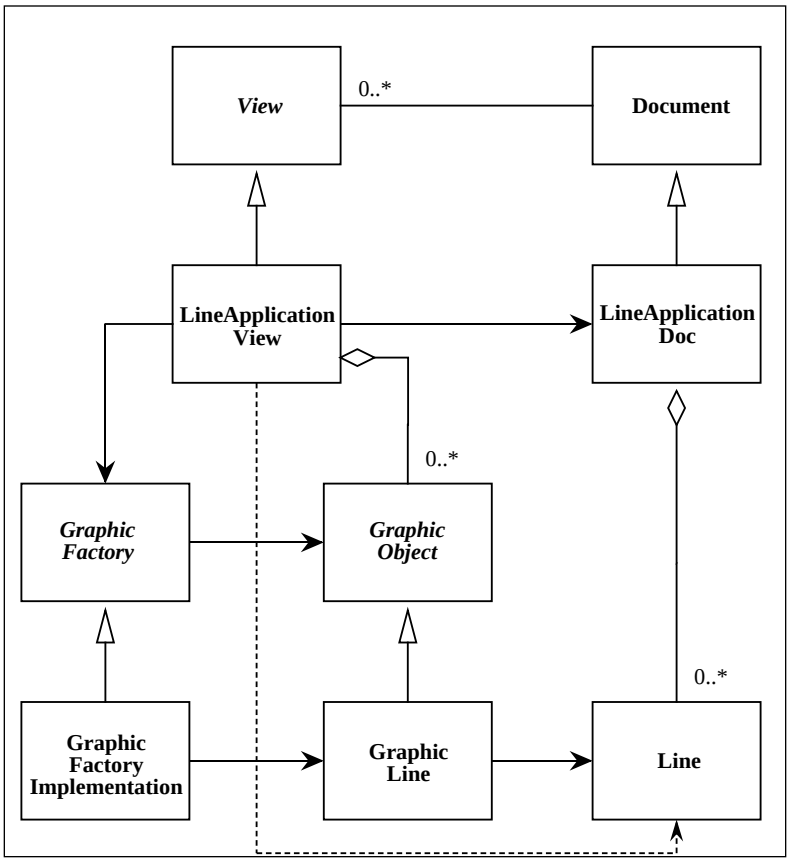
It was issues such as these that motivated the Document/View split in many of the current frameworks. These are also the issues that partially motivated the Model/View split in the Model/View/Controller paradigm used in Smalltalk nearly two decades ago.

We need to separate representation and manipulation from creation and display. That is, the mechanisms that store and manipulate the data should be separate from, and reusable independently of, the mechanisms that display that data to humans; or present that data to other computers. Figure 2 shows such a split.

Here we see two fundamental splits. First we see that there is an class named `Line`; and another similar class named `GraphicLine`. The `Line` class is a pure mathematical model of a line segment; it knows nothing about how to display or create a line on a GUI. `GraphicLine` on the other hand contains a `Line`. It knows nothing about the mathematical model of a line segment, but knows how to draw a line segment on a GUI.

We see a parallel split between `LineApplicationDoc` and `LineApplicationView`. `LineApplicationDoc` contains a list of `Line` objects. It knows how to save and restore them to a file. But it knows nothing about how to draw them, or otherwise present them to a user. On the

Figure 2: Document/View Solution.



other hand, `LineApplicationView` contains a list of `GraphicObjects` and knows how to draw them. It also knows how to interpret the mouse to create `Line` objects which it passes to the `LineApplicationDocument`.

An OBSERVER[1] pattern is set up between the `LineApplicationView` and the `LineApplicationDoc`. The view observes the document and is informed whenever the document is changed. Thus, when a new `Line` object is added to the document, the view is notified and the appropriate `GraphicObject` is added to its list and drawn.

There is a distinct asymmetry in the relationships of this split. `LineApplicationView` knows about `LineApplicationDoc`; and `GraphicLine` knows about `Line`. However, the reverse is not true. `LineApplicationDoc` could be reused with a completely different view; and `Line` could be reused with a completely different presentation mechanism. Thus, this structure achieves our goal of separating the manipulation of the data from its presentation. We can now present the data in many completely different ways, without affecting the code that manipulates that data in any way.

Listing 3 through Listing 6 shows snippets of the code that implements this example. Only the most important bits of code are shown. There is quite a bit more code in this example, which can be downloaded from the website.

Notice that in some ways this code is very similar to our first example. The new `LineApplica-`

1. Design Patterns, Gamma, et. al, Addison Wesley, 1995

tionView class looks very similar to the previous LineApplicationWindow class. However there are some important differences

Listing 3: LineApplicationView.h

```
class LineApplicationView : public CView
{
  public:
   LineApplicationView();
   virtual ~LineApplicationView();

   LineApplicationDoc* GetDocument();

  private:
   vector<GraphicObject*> itsObjects;

   CPen*           itsBluePen;
   CPoint          itsFirstPoint;
   CPoint          itsSecondPoint;
   GraphicFactory* itsGraphicFactory;

   virtual void OnInitialUpdate();
   virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
   virtual void OnDraw(CDC* aDC);
   afx_msg void OnLButtonDown(UINT, CPoint);
   afx_msg void OnLButtonUp(UINT, CPoint);
   afx_msg void OnMouseMove(UINT, CPoint);
   virtual void DeleteObjects();

   DECLARE_DYNCREATE(LineApplicationView)
   DECLARE_MESSAGE_MAP()
};
```

Listing 4: LineApplicationView.cpp

```
void LineApplicationView::OnDraw(CDC* aDC)
{
  vector<GraphicObject*>::iterator index;
  for(index = itsObjects.begin(); index != itsObjects.end(); ++index)
  {
   GraphicObject* anObject = *index;
   anObject->Draw(*aDC);
  }
}

void LineApplicationView::OnLButtonUp(UINT nFlags, CPoint point)
{
  if(GetCapture() == this)
  {
   itsSecondPoint = point;
   ReleaseCapture();

   LineApplicationDoc* document = GetDocument();
   Line* newLine = new Line(itsFirstPoint, itsSecondPoint);
   document->AddLine(newLine);
  }

  return;
}

void LineApplicationView::OnLButtonDown(UINT nFlags, CPoint point)
{
  SetCapture();
  itsFirstPoint = itsSecondPoint = point;
```

Listing 4: LineApplicationView.cpp (Continued)

```
}

void LineApplicationView::OnUpdate(CView* sender, LPARAM lHint,
                           CObject* pHint)
{
  if(Line* aLine = dynamic_cast<Line*>(pHint))
  {
   GraphicObject* anObject = itsGraphicFactory->MakeLine(*aLine);
   itsObjects.push_back(anObject);

   CClientDC dc(this);
   OnPrepareDC(&dc);
   anObject->Draw(dc);
  }
  else
  {
   CView::OnUpdate(sender, lHint, pHint);
  }

  return;
}
```

Listing 5: LineApplicationDoc.h

```
class LineApplicationDoc : public CDocument
{
  public:
   LineApplicationDoc();
   virtual ~LineApplicationDoc();

   virtual void AddLine(Line*);
   virtual void GetLines(vector<Line*>& lines) const;

  private:
   CObArray itsLines;

   virtual void DeleteContents();
   virtual void Serialize(CArchive& ar);
            void ClearArray();

   DECLARE_DYNCREATE(LineApplicationDoc)
   DECLARE_MESSAGE_MAP()
};
```

Listing 6: LineApplicationDoc.cpp

```
void LineApplicationDoc::AddLine(Line* aLine)
{
  itsLines.Add(aLine);
  SetModifiedFlag();
  UpdateAllViews(NULL, 0, aLine);
}
```

In `LineApplicationView::OnLButtonUp` we are creating instances of class `Line` rather than `GraphicLine`. Also, rather than adding the instance to our own list, we instruct the document to add it to its list.

In `LineApplicationDoc::AddLine` we not only add the instance of the `Line` into the document's list, but we also call the function `UpdateAllViews`, passing the instance of the `Line` back in the "hint" parameter. This function will find all the views associated with this document

and invoke their `OnUpdate` functions.

`In LineApplicationView::OnUpdate` we check to see if the 'hint' being passed in is an instance of a `Line`. And if so we use the `Line` to create a `GraphicObject` which we then add to the view's list of `GraphicObjects`. Then we draw the newly created `GraphicObject`.

By adopting the Document/View architecture we have made it possible to completely replace the way that the `Line` instances are displayed, without affecting the way that they are stored or "Serialized" (i.e. read and written from files). We have also made it possible to display these lines in many different ways, and in many different windows, within the same application.

Push Model vs. Pull Model

The OBSERVER relationship between the `LineApplicationView` and `LineApplicationDoc` classes conforms to the *push* model. This means that the data that the observing view is interested in is "pushed" along with the message that notifies the observing view that the document has changed. In our example, the `Line` instance was "pushed" in the "hint" argument of the `OnUpdate` function.

The alternative to the "push" model is to use the "pull" model. In the "pull" model, no data is sent in the `OnUpdate` message. Upon receipt of the `OnUpdate` message, the views must pull all the data from the document and redisplay it all.

Clearly the push model is more efficient than the pull model. However, the pull model is more general. When using the push model, the `OnUpdate` message must contain enough information to tell the view exactly what changed. As the document becomes more complicated, and the kinds of manipulations become more varied, the complexity and amount of information that will have to be pushed to the view will increase.

For example, suppose that we wrote an application in which we could draw lines, circles, and squares. If we used the push model, the `OnUpdate` method would have to query the 'hint' to see what kind of object it was. This could amount to an if/else chain of dynamic casts; a blatant violation of the OPC! Suppose also that we could move, stretch, and delete those lines, circles, and squares. Then, not only would we have to pass the changed object into `OnUpdate`; but we would also have to tell `OnUpdate` the kind of change that it had experienced. If deleted, it would have to be erased. If stretched or moved, it would have to be erased and redrawn. If merely added, it would have to be drawn for the first time.

This extra information could be passed as some kind of enum in the second hint field of the `OnUpdate` method. However, this would turn the `OnUpdate` method into a horrible stew of nested if/else and switch statements. A better approach would be to use COMMAND[1] objects. Consider listing 7.

Listing 7: Pushing Commands

```
class ViewInterface
{
  public:
   virtual void AddNew(Shape*) = 0;
   virtual void Delete(Shape*) = 0;
   virtual void Replace(Shape* old, Shape* new) = 0;
};
```

1. Another of the patterns from the Design Patterns book.

Listing 7: Pushing Commands

```
class ViewCommand
{
  public:
   virtual void Execute(ViewInterface&) = 0;
};
```

All the views could implement the `ViewInterface` class by using multiple inheritance. This is a good example of the Interface Segregation Principle (ISP)[1]. The interface needed by the `ViewCommand` class is kept separate from the view class so that the `ViewCommand` class does not have to depend upon the views.

Now, when a view has decided that a square, circle, or line has been added, removed, moved, or stretched, it can create a derivative of the `ViewCommand` class that knows how to manipulate the ViewInterface class appropriately. This command instance can be passed to the document which will then pass it in the 'hint' argument of the `UpdateAllViews` member function. All the views will therefore receive this instance of the `ViewCommand` in the 'hint' argument of their OnUpdate member functions. They will then invoke the `Execute` function and pass themselves as its argument:

```
void SomeView::OnUpdate(CView* v, LPARAM lp, CObject* hint)
{
   if (ViewCommand* cmd = dynamic_cast<ViewCommand*>(hint))
   {
      cmd->Execute(this);
   }
   else
      CView::OnUpdate(v,lp,hint); // defer to base class.
}
```

By using this technique, we can add arbitrary complexity to the push model interface without creating a rats nest of if/else and switch statements in the `OnUpdate` functions of all the views.

The Taskmaster Architecture

What is that COMMAND mechanism really doing? It seems to be allowing one part of the view to communicate to another part of the view! Decisions that are made in the interactive part of the view are being communicated to the part of the view that manages what to display. Perhaps another separation is order.

We have identified two different parts of the view object. There is the part that handles the interactions with the mouse and keyboard, and then there is the part that handles display and update. And these two appear to be quite separate. The interaction part sends messages to the document, and the display part receives `OnUpdate` messages from the document.

Whenever there are two aspects to one object, we should consider whether those aspects should be separated into two objects. We can use the Open/Closed Principle (OCP) as a way of determining the value of such a separation: Are there changes that could be made to the interaction part of the view, that the display part of the view should be closed to? Would we want to reuse the display

1. The Interface Segregation Principle, Robert C. Martin, C++ Report, Aug '96

portion of the view with several different variations of the interaction part?

Consider what would happen if we wanted to change the way that users create lines. Rather than having them depress the button at the start point, drag to the end point, and then release the button; we have them press and release the button at the start point, move the mouse to the end point, and then press and release the button again. In the above example, this would cause some rather drastic changes to `OnLButtonUp`, `OnLButtonDown`, and `OnMouseMove`; but would not affect `OnUpdate` at all! It seems likely that different applications that use different styles of interactions will want to reuse the `OnUpdate` function, and all the other aspects of object display.

Model/View/Controller

This was the issue that drove the View/Controller split in the MVC paradigm. Indeed there are many different interaction schemes which could be used to create and manipulate objects. These interaction schemes need not be tightly coupled to the display mechanism. Thus, in those instances where we expect the interactions to change frequently, or where we expect many different application to display the same data but manipulate it differently, separating the interaction from the view is probably a good idea.

In MVC the object that controlls the interaction is called the *controller*. The controller is responsible for intercepting all the events from the user interface and interpreting them into commands that manipulate the model.

Tasks

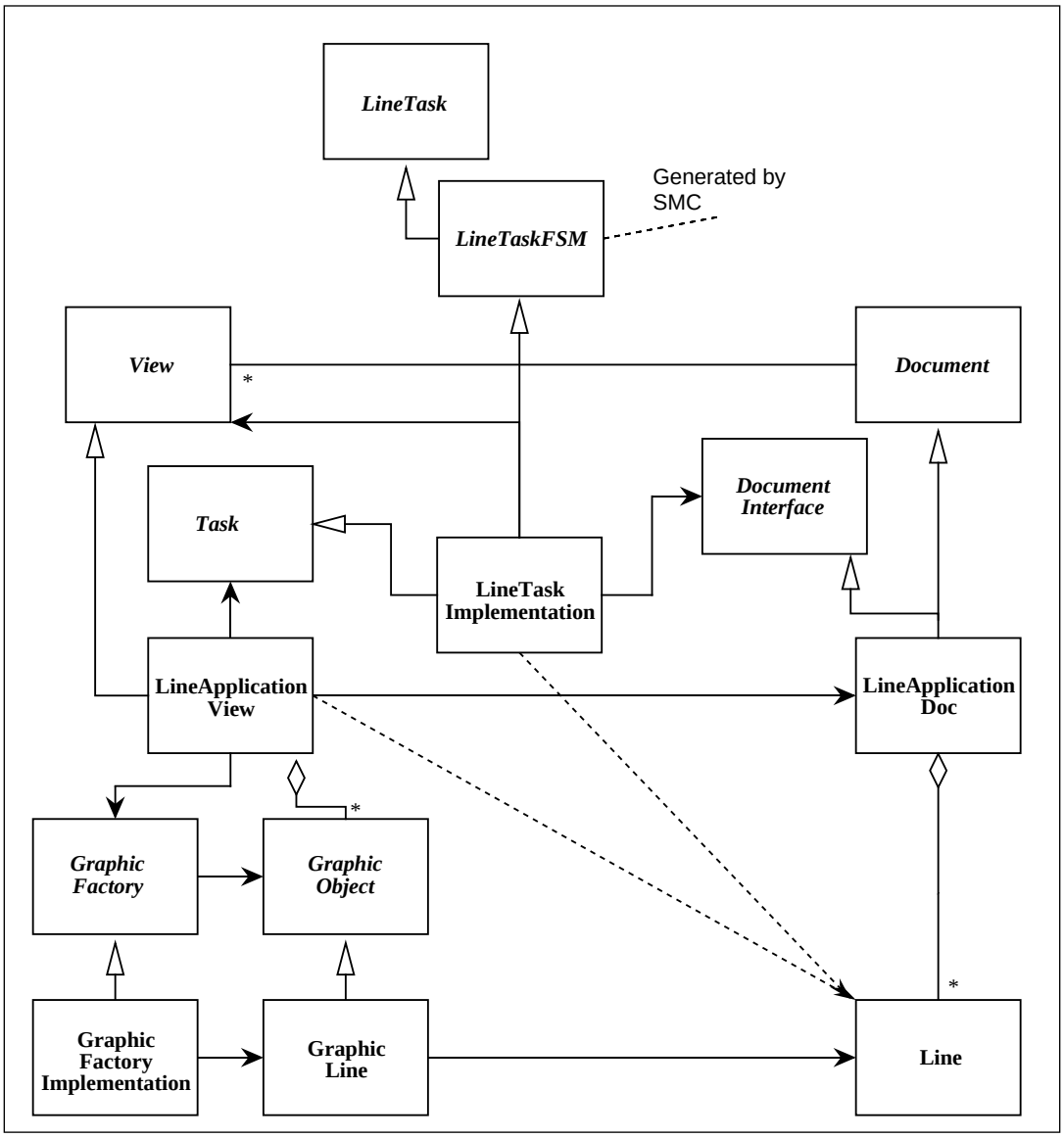
In the TASKMASTER architecture (See Figure 3.) things are a bit different. All the events coming in from the user interface are received by the `View`. However, those events are then delegated to an object that is derived from the `Task` interface. Each derivative of Task encapsulates an interaction with the user and eventually communicates with the document. The appropriate derivative of Task is selected by a menu command or click on a palette item, etc.

Figure 3 is quite similar to the Document/View model shown in Figure 2, but has some extra classes and interfaces. We see that the `LineApplicationView` is associated with the `Task` class. It is through this association that the user interface events received by the `View` are delegated to the `Task`. `LineImplementationTask` is the object that encapsulates the interaction. We will discuss it in more detail later. For the moment simply understand that this object interacts with the user and knows when a `Line` object has been created.

`LineTaskImplementation` is associated with an abstract class called `DocumentInterface`. This is another instance of the Interface Segregation Principle (ISP). This class contains nothing but the pure virtual function `AddLine(Line*);`. Notice that `LineApplicationDoc` inherits from `DocumentInterface`. This is the pathway by which the `LineTaskImplementation` communicates with the `LineApplicationDoc` without having to depend upon it. Thus the `LineApplicationDoc` class can be freely modified without affecting any of the derivatives of `Task`.

`LineTaskImplementation` also has an association with `View`. This is needed since the task must call member functions of View in order to run the interaction. One example of this is the `View::Setcapture` method which must be managed by the interaction.

Figure 3: Taskmaster



Finite State Machines

In TASKMASTER, we consider all `Task` derivatives to be finite state machines. The events that drive the FSM are the mouse and keyboard events that are delegated by the View to the `Task`. Indeed, the `Task` class consists of little more than a set of pure virtual functions representing these events.

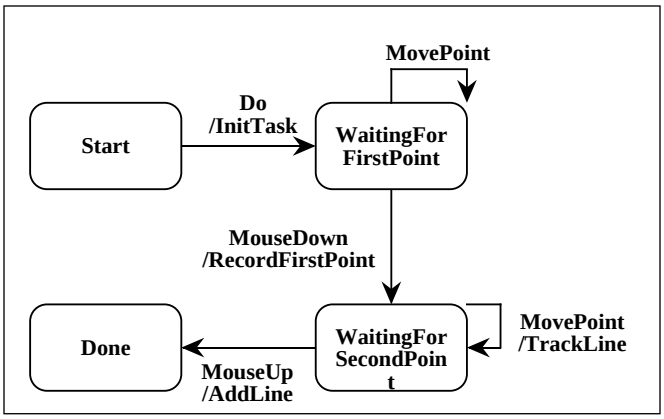
There are numerous ways of implementing finite state machines. One can use nested switch-case statements, or some kind of table driven approach. My favorite scheme in C++ was documented in the wonderful article “Finite State Machines: a model of behavior for C++” by Immo Hüneke, C++ Report, January, 1991. (This was in the pre-glossy “glory” days of the C++ Report. That particular issue was 24 pages long, was edited by Rob Murry, and had a full page ad for Borland’s Turbo C++ on page 3.)

This style of FSM is more generically documented as the STATE pattern in the *Design Patterns*

book. The particular variation of this that we use in TASKMASTER is a pattern called: THREE LEVEL FSM[1]. This pattern is convenient because it yields well to automatic code generation. We use a tool called SMC[2] to generate all our finite state machine code.

The finite state machine for adding a line is shown in Figure 4. The operation of this state machine is quite simple. It starts its life in the `Start` state. When the task is associated with the view, it is sent the `Do` event which kicks everything off.

Figure 4: LineTaskFSM



The following table describes the rest of the operation of this simple FSM. You read it as follows: "If we are in the `Start` state, and we receive the `Do` event, then we go to the `WaitingFor-FirstPoint` state and call the `InitTask` function.

Table 1: Line Task State Transition Table

Current State	Event	Next State	Action
Start	Do	WaitingForFirstPoint	InitTask
WaitingForFirstPoint	MouseDown	WaitingForSecondPoint	RecordFirstPoint
WaitingForFirstPoint	MovePoint	WaitingForFirstPoint	
WaitingForSecondPoint	MouseUp	Done	AddLine
WaitingForSecondPoint	MovePoint	WaitingForSecondPoint	TrackLine

Referring back to Figure 3, we have created a class named `LineTask`. This class as four pure virtual functions, one for each of the actions of the finite state machine. We call this class the *context* of the FSM. One of the classes generated by SMC is named `LineTaskFSM`. This class inherits from `LineTask`; allowing it to invoke the action functions declared there. Finally, we implement the action function in a class named `LineTaskImplementation`. This class inherits from `LineTaskFSM` in order to implement the pure virtual action functions. It also inherits from

1. From the first *Pattern Languages of Program Design*, Coplien & Schmidt, Addison Wesley, 1995, p383
2. SMC is freely available in the public domain. You can get it from the freeware section of our website: `http://www.oma.com`

`Task` so that it can receive the events from the view. This is, once again, an example of the ISP. We don't the view to know anything about the details of the task, so the view communicates with the task through an abstract base class. This lets us change the task without affecting the view.

Task isolation

Our example application has only one task. However, a real application would have dozens or hundreds. And each would be isolated from the associated view and document classes. The tasks can be changed without affecting any other part of the system. Indeed, the interactions can be reused in other applications that have different document and view classes. Also the document and view classes can be reused in systems that have different interactions.

Task switching

In a more elaborate application, tasks will be short lived entities. They will be associated with a view as a result of some kind of command; perhaps a menu choice, a keyboard shortcut, or a click in the appropriate button of a palette or toolbar. Once associated with the view, the task will continue to collect events and communicate with the document class until its lifecycle ends. This may be as a result of completing its job, or because it was somehow cancelled. Then another task will replace it.

Thus, the current task within a view represents the global state of that view. It defines how the view will react to events. In order to implement this, a suite of other interfaces are required within `Task`. These interfaces provide for task cancellation, task restart, backstepping, pausing, etc. The exact complement of interfaces will depend upon the GUI and application domains.

Taskmaster in (some) detail.

Listings 8 through 13 show portions of the Taskmaster implementation of our sample application. The bulk of the document and view classes have not changed, so they are not shown here. What is shown is the structure surrounding the tasks. Notice how all of the interaction code from the first two examples has been moved into the class `LineTaskImplementation`.

Listing 8: LineTask.h

```
class LineTask
{
  public:
   LineTask();
   virtual ~LineTask();

   virtual void InitTask() = 0;
   virtual void RecordFirstPoint() = 0;
   virtual void AddLine() = 0;
   virtual void TrackLine() = 0;
};
```

Listing 9: LineTask.sm (the input to SMC)

```
FSMName  LineTaskFSM
Context  LineTask
Initial  Start
Header   linetask.h
{
Start
{
```

Listing 9: LineTask.sm (the input to SMC) (Continued)

```
   Do          WaitingForFirstPoint       InitTask
}

WaitingForFirstPoint
{
   MouseDown   WaitingForSecondPoint      RecordFirstPoint
   MovePoint   WaitingForFirstPoint       {}
}

WaitingForSecondPoint
{
   MouseUp     Done                       AddLine
   MovePoint   WaitingForSecondPoint      TrackLine
}

Done
{}
}
```

Listing 10: Task.h

```
class Task
{
  public:
      Task();
      virtual ~Task();

      virtual void Start() = 0;
      virtual void LeftMouseUp(UINT nFlags, const CPoint& point) = 0;
      virtual void LeftMouseDown(UINT nFlags, const CPoint& point) = 0;
      virtual void MouseMove(UINT nFlags, const CPoint& point) = 0;
};
```

Listing 11: DocumentationInterface.h

```
class DocumentInterface
{
  public:
   DocumentInterface();
   virtual ~DocumentInterface();

   virtual void AddLine(Line*) = 0;
};
```

Listing 12: LineTaskImplementation.h

```
class LineTaskImplementation : public LineTaskFSM,
                               public Task
{
  public:
   LineTaskImplementation(CView* theView,
                          DocumentInterface* theDoc);
   virtual ~LineTaskImplementation();

   // member function defined in task
   virtual void Start();
   virtual void LeftMouseUp(UINT nFlags, const CPoint& point);
   virtual void LeftMouseDown(UINT nFlags, const CPoint& point);
   virtual void MouseMove(UINT nFlags, const CPoint& point);

   // member functions defined in the actions of the FSM
   virtual void InitTask();
   virtual void RecordFirstPoint();
```

Listing 12: LineTaskImplementation.h (Continued)

```
   virtual void AddLine();
   virtual void TrackLine();
private:
   CPen*                itsBluePen;
   CPoint               itsFirstPoint;
   CPoint               itsSecondPoint;
   CPoint               itsTemporaryPoint;
   GraphicFactory*      itsGraphicFactory;

   // for update purposes
   CView*               itsView;
   DocumentInterface*   itsDocument;
};
```

Listing 13: LineTaskImplementation.cpp

```
void LineTaskImplementation::Start()
{
  Do();
}

void LineTaskImplementation::LeftMouseUp(UINT nFlags,
                                  const CPoint& point)
{
  itsTemporaryPoint = point;
  MouseUp();
}

void LineTaskImplementation::LeftMouseDown(UINT nFlags,
                                   const CPoint& point)
{
  itsTemporaryPoint = point;
  MouseDown();
}

void LineTaskImplementation::MouseMove(UINT nFlags,
                                const CPoint& point)
{
  itsTemporaryPoint = point;
  MovePoint();
}


void LineTaskImplementation::AddLine()
{
  if(itsView->GetCapture() == itsView)
  {
   itsSecondPoint = itsTemporaryPoint;

   // release the mouse
   ReleaseCapture();

   Line* newLine = new Line(itsFirstPoint, itsSecondPoint);
   itsDocument->AddLine(newLine);
  }

  return;
}

void LineTaskImplementation::RecordFirstPoint()
{
  itsView->SetCapture();
  itsFirstPoint = itsSecondPoint = itsTemporaryPoint;
```

Listing 13: LineTaskImplementation.cpp (Continued)

```
}

void LineTaskImplementation::TrackLine()
{
  if(itsView->GetCapture() == itsView)
  {
   CClientDC dc(itsView);
   itsView->OnPrepareDC(&dc);

   // save the current dc parameters that we are changing
   int previousROP = dc.SetROP2(R2_XORPEN);
   CPen* currentPen = dc.SelectObject(itsBluePen);

   // draw the previous line
   dc.MoveTo(itsFirstPoint);
   dc.LineTo(itsSecondPoint);

   // draw the new line
   dc.MoveTo(itsFirstPoint);
   dc.LineTo(itsTemporaryPoint);
   itsSecondPoint = itsTemporaryPoint;

   // reset previous parameters
   dc.SetROP2(previousROP);
   dc.SelectObject(currentPen);
  }

  return;
}

void LineTaskImplementation::InitTask()
{}
```

Notice that we are still using the push model here. Notice also that all the comments that I made about COMMAND objects still apply, except that now they can be created by the individual tasks rather than in the view.

But it's not simple anymore!

True. The additional decoupling and isolation has increased the complexity of the structure several fold. A simple glance at Figures 1, 2, and 3 will serve to convince you of that. Also consider that the total line count of example 1 is only 396 lines; whereas example 2 has grown to 1263 and example 3 to 1985 (a good year). So we have multiplied the number of lines of code (including comments) by nearly a factor of five!

Why incur this extra expense? For small applications you probably wouldn't. But when the applications get large, that factor of five will decrease to nearly 1:1. In other words, TASKMASTER represents a superstructure upon which large applications can be built and maintained. As more and more functionality gets added into that superstructure, the additional complexity of Taskmaster will dwindle in significance. And the ability of such programs to be easily changed, maintained, and reused will offset the extra complexity even more.

Remember, it takes complexity to manage complexity.

Conclusion

We, at Object Mentor Inc., have had a great deal of success with the TASKMASTER architecture. We have used it in over a dozen GUI applications of significant size; and have found that the reduced coupling between tasks, views and documents allows us the ability to make many changes to the software without incurring massive recompiles. This is a large benefit when working on a 70,000 line application. We have also been able to put much of the taskmaster software into DLLs and then reuse it in different applications.

TASKMASTER is not for everyone, or every application. But in those cases where the applications are large, variable, and present a potential for reuse; TASKMASTER is an option to consider.