Alternator

An Object Behavioral

Design Pattern

John Liebenau

lieb@itginc.com

1. Intent

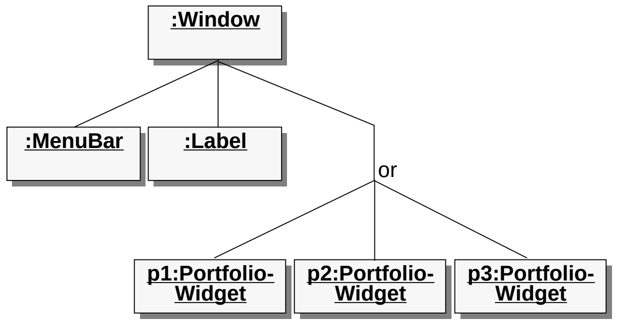
Allow multiple alternative subtrees in a hierarchical structure, and make them transparent to clients of the structure.

2. Motivation

Suppose we are developing a stock trading application. The main purpose of our application is to provide a trader with visual access to groups of stocks, called portfolios, which can be bought and sold by the trader. We will use an object-oriented GUI framework to build our application's graphical user interface. Each window will contain a hierarchy of visual components. A simple way of organizing the application is to display each portfolio in its own window. The trader can choose a portfolio by selecting a window to work with. In general, mapping portfolios to windows seems to work well but the trader's work environment imposes additional requirements for our application.

The trader's work environment already includes a variety of applications for displaying and analyzing stock related information. A typical trader may require multiple workstations or dedicated news displays in order to do his/her job. This proliferation of video displays and application windows inevitably leads to information overload. Traders prefer using applications that provide the right information and minimize the amount of screen real estate needed for display in order to reduce the effects of information overload.

Our stock trading application needs to provide a mechanism for a single window to select and display the portfolios one at a time. In essence, the window has to present several alternative views to the trader based on the trader's selections. The diagram below shows the component hierarchy for our stock trading application with three portfolios.

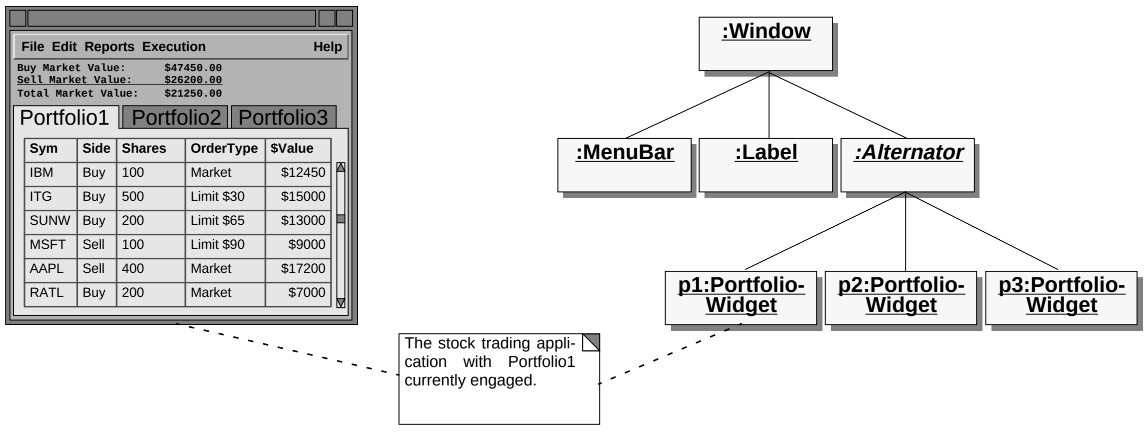


Our stock trading application's GUI consists of a single window containing a menu bar, labels that display summary information about the current portfolio, and several widgets that display portfolio information. We still need a mechanism for:

- managing the placement and sizing of the portfolio widgets,
- managing the visibility of the portfolio widgets, and
- managing the selection of a portfolio widget.

We could implement this mechanism in our window object by creating a subclass of Window that provides the customized algorithms needed handle our special situation. However, this approach has several drawbacks. It forces us to write a new subclass of Window for each situation where we require alternative views. It prevents us from taking advantage of the algorithms that are already provided by the object-oriented GUI framework.

To solve this problem, we can use an Alternator object to manage the selection, visibility, and arrangement of alternative component groups. Alternators act as both place holders and containers for their components. The diagram below expands upon the previous diagram by introducing the Alternator component to manage the alternate views.

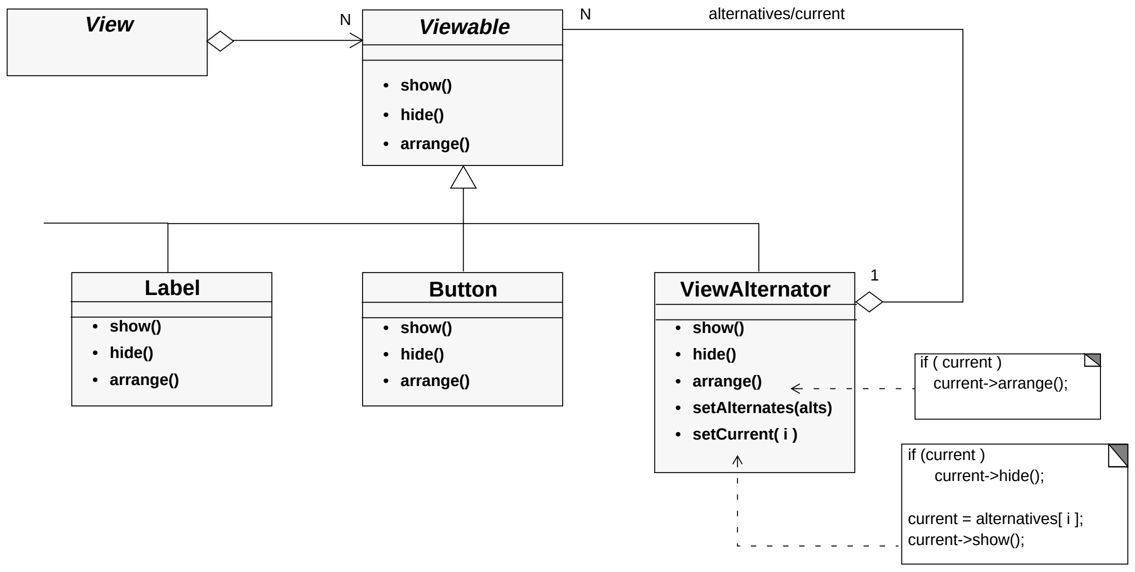


```
Buy Market Value:      $47450.00
Sell Market Value:     $26200.00
Total Market Value:    $21250.00
```

Sym	Side	Shares	OrderType	$Value
IBM	Buy	100	Market	$12450
ITG	Buy	500	Limit $30	$15000
SUNW	Buy	200	Limit $65	$13000
MSFT	Sell	100	Limit $90	$9000
AAPL	Sell	400	Market	$17200
RATL	Buy	200	Market	$7000

The stock trading application with Portfolio1 currently engaged.

The following class diagram shows a small piece of a graphical user interface framework. The framework provides a Viewable class which defines the common interface for graphical user interface objects. The View class maintains references to the Viewable objects that make up the View's contents. The Viewable class interface is extended by the Label, Button, and ViewAlternator classes. These classes have operations specific to them. The ViewAlternator class maintains references to its alternates and to the currently engaged alternate. The View class treats the ViewAlternator as just another Viewable object and has no knowledge that it is a place holder for a group of alternatives.



```
if ( current )
    current->arrange();
```

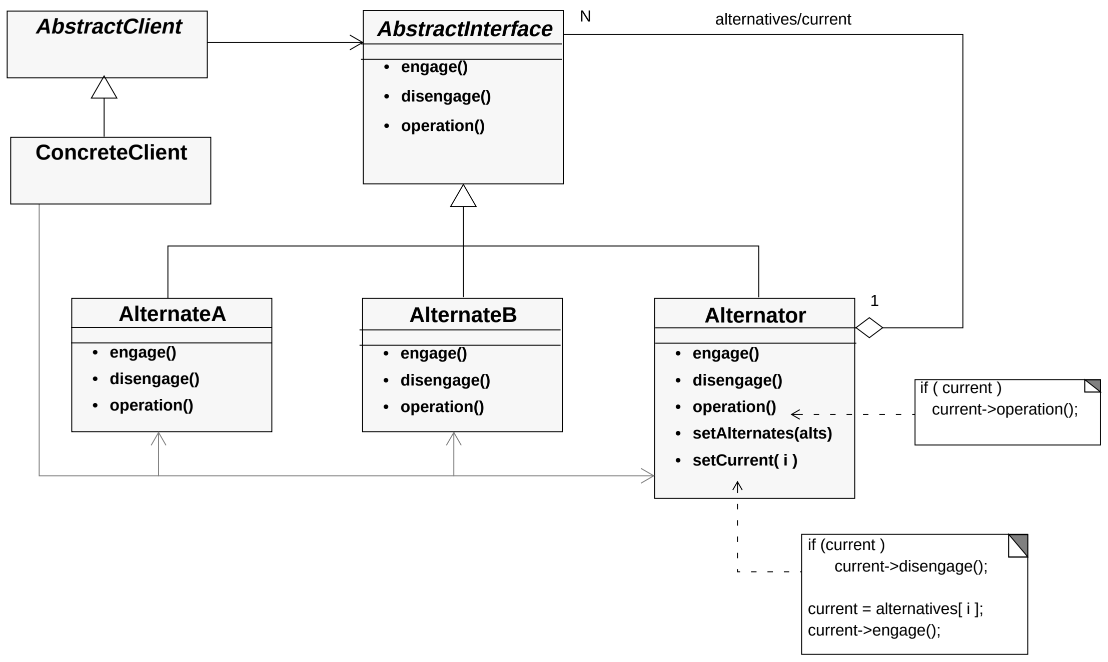
```
if (current )
      current->hide();

current = alternatives[ i ];
current->show();
```

3. Applicability

Use the Alternator pattern when:

- an object is part of a hierarchy
- an object is composed of several sub-objects with only one sub-object engaged at any given time
- you want multiple objects to exist in the logical space of a single object
- you want a way of selecting between alternatives
- alternative objects have some form of common interface

4. Structure



5. Participants

- AbstractInterface (Viewable)
  - provides the common interface between Alternators and Alternates
  - provides engage and disengage operations (e.g. show and hide) as part of the common interface
- Alternator (ViewAlternator)
  - provides an interface for setting alternatives and selecting a current alternative
  - provides a mechanism for transitioning from one alternative to another (i.e. disengaging one component and engaging another)
- Alternate (Button,CheckBox,...)
  - provides a concrete implementation of the AbstractInterface
  - may provide additional interfaces specific to each Alternate

- AbstractClient (View)
    - uses Alternators and Alternates through the AbstractInterface
- ConcreteClient (SpecificView)
    - uses Alternators and Alternates directly through their specific interfaces
    - may be responsible for creating and configuring Alternates and Alternators

6. Collaborations

- Clients use the AbstractInterface to manipulate objects. If the receiver object is an Alternate, the operation is handled directly by the Alternate object. If the receiver object is an Alternator, the operation is forwarded to the currently engaged Alternate object.
- Clients may use more specific interfaces of the Alternator and Alternate classes. In particular, clients may configure the Alternates using their specific interfaces and then configure the Alternator with a list of Alternates. The current Alternate is also set by Clients.

7. Consequences

The Alternator design pattern has the following benefits:

- *Allows several alternatives to appear as a single object*. This simplifies the structure of algorithms that use components in the object hierarchy because the algorithms can treat all of the components in a uniform manner.
- *Transparently manages the activation and deactivation of alternative objects*. These multiple objects occupy the same logical place in the object hierarchy. In a GUI context, this logical place is a component's arrangement on the screen. When a new alternate is selected to be currently engaged, the Alternator automatically disengages the previously engaged alternate.
- *Conserves resources*. In some domains such as user interfaces, the Alternator pattern can allow an application to conserve resources like screen real estate in an efficient and flexible manner.

Alternator has the following liabilities:

- *Could force alternates to be too general*. Alternates are manipulated through their common interface. This interface may be very general depending on the class hierarchy. Downcasting or maintaining additional reference may be required to access the specialized interfaces of alternates.
- *May prevent or hinder access to disengaged alternates*. Alternators are designed to appear like a single component in a hierarchy but some clients may need to have access to an alternator's disengaged components as well as its engaged one. This can break the Alternator's appearance as a single component, negating some of its benefits.

8. Implementation

The following implementation issues should be considered when using the Alternator design pattern:

- *Creating alternates and configuring alternators*. Typically, alternate and alternator components are created directly by a client or on behalf of a client by an Abstract Factory [GHJV95]. The client will then configure an alternator with its alternate components. Some situations may require an alternator to create its own alternate components. These kinds of alternators will typically encapsulate their alternate components and prevent any direct access to them.
- *Consistent engage and disengage behavior*. The components that will act as alternates should have consistent or compatible behavior in their engage/disengage methods. Typically, components should have some form of “on” and “off” states. In the example described in the Motivation, GUI components are either visible on the screen or invisible. Visible components can accept input while invisible components can not.
- *Maintain additional references to the alternates*. In strongly typed languages (C++, Java), the Alternator will only provide an interface that is common to all of the components in the hierarchy (i.e. AbstractInterface). If you need to access methods that are specific to an alternate, you will either need an additional reference (of the specific type) to an alternate or you will need to downcast to the specific type.
- *Managing the state of alternates*. Sometimes alternates will be completely independent of each other. Their state will only be updated when they are engaged. In other scenarios, the alternates are dependent on each other. When the currently engaged alternate’s state is updated, the other alternates’ state must be updated as well. This can be handled by the Alternator component. All operations that modify the alternates’ state are forwarded to each of the alternates. For example, if the ViewAlternator class described in the Motivation implements a method for setting its x or y coordinates, the coordinates of all its alternates need to be updated:

```
void
ViewAlternator::setX(int x)
{
    for (
        vector<Viewable*>::iterator alternate( alternatives.begin() );
        alternate != alternatives.end();
        alternate++
        )
        (*alternate)->setX( x );
}
```

- *Alternator is implemented with Composite.* Alternator is used in hierarchical object structures. These object structures are implemented using the Composite design pattern [GHJV95]. Alternator is a kind of Composite but differs in that Alternator introduces the notion of selecting a child to be the currently active or engaged component while the other children are inactive.
- *Engaging multiple alternates simultaneously*. Some situations may call for more than one alternate to be engaged at the same time. This can be accomplished by maintaining a list of the engaged alternates instead of a single reference. An additional method for setting the currently engaged alternates should also be provided.

9. Sample Code.

The sample code is taken from the example given in the motivation section. The Viewable abstract class specifies the common interface of all GUI components.

```
class Viewable
{
public:
    //  ...

    // Modifiers
    virtual void  setX(int  x);
    virtual void  setY(int y);
    virtual void  setHeight(int  h);
    virtual void  setWidth(int  w);

    // Selectors
    virtual int   getX()const;
    virtual int   getY()const;
    virtual int   getHeight()const;
    virtual int   getWidth()const;

    // Actions
    virtual void  arrange()=0;
    virtual void  show()=0;
    virtual void  hide()=0;
};
```

The Button class extends the Viewable abstract class by providing methods specific to button components. The Button class also implements the pure virtual methods declared in Viewable.

```
class Button: public Viewable
{
public:
    //  ...

    // Modifiers
    void      setLabel(const string& lbl);
    void      setCommand(Command* cmd);

    // Action
    void      doCommand();
};
```

The ViewAlternator class is a GUI component that manages other GUI components (even other ViewAlternators). The ViewAlternator allows clients to select the alternate they wish to be visible and active. The other alternates are automatically hidden and deactivated.

```
class ViewAlternator : public Viewable
{
private:
    vector<Viewable*>   alternatives;
    Viewable*           current;
public:
    //  ...

    // Viewable: Actions
    void  arrange();

    // Modifiers
    void       setAlternatives(vector<Viewable*>& alt);
    void       setCurrent(Viewable* v);

    // Selectors
    Viewable*  getCurrent()const;
    int        getNumAlternatives()const;
};
```

The ViewAlternator delegates method calls to its currently engaged alternate.

```
void
ViewAlternator::arrange()
{
    if ( current )
        current->arrange();
}
```

When setting the alternatives, the ViewAlternator engages (shows) one of the alternates and disengages (hides) all of the others.

```
void
ViewAlternator::setAlternatives(const vector<Viewable*>& alt)
{
    vector<Viewable*>::iterator alternate;

    alternatives = alt;
    alternate = alternatives.begin();

    if ( alternate != alternatives.end() )
    {
        current = *alternate;
        current->show();
        alternate++;
    }

    for (
        ;
        alternate != alternatives.end();
        alternate++
        )
        (*alternate)->hide();
}
```

When setting a new current alternate, the ViewAlternator hides the old current and makes the new current visible. If the old current and the new current are the same, the method returns without performing any changes.

```
void
ViewAlternator::setCurrent(Viewable* v)
{
    if ( current != v )
    {
        for (
            vector<Viewable*>::iterator alternate( alternatives.begin() );
            alternate != alternatives.end();
            alternate++
            )
        {
            if ( *alternate == v )
            {
                current->hide();
                current = *alternate;
                current->show();
                break;
            }
        }
    }
}
```

10. Known Uses

The Alternator pattern is used in several places in the GUI Framework 2.0, a graphical user interface framework developed for internal use at ITG. The GUI Framework provides a ViewAlternator class which manages alternate Viewable objects. A more specific implementation of the Alternator pattern is the Selector class which can present itself alternatively as a ScrollingList, CheckBox, OptionMenu, or MultipleChoice.

Java contains at least two examples of the Alternator pattern. The Java Swing Library has a class called JTabbedPane [Topley98], a component which lets the user switch between a group of components by clicking on a tab with a given title and/or icon. The Java AWT Library has a class called CardLayout [CL98], a container which shows only one of its components at a time; all other components in the container are hidden. The CardLayout provides methods to set the currently visible component.

The Microsoft Foundation Class Library provides a class called CPropertySheet [Microsoft95], a container of one or more CProperyPage components which in turn contain other widgets. The CPropertySheet lets the user select a CPropertyPage by clicking on its associated tab.

Motif provides a Notebook widget [OSF94] which contains multiple pages of other widgets. Each page can be selected by clicking on a tab.

11. Related Patterns

The Alternator is implemented using the Composite design pattern [GHJV95].

The Alternator pattern is similar to the Backup pattern [ST95] except it solves a more general problem. Backup is concerned with providing alternatives for a given function that can automatically be engaged if the primary function fails, while Alternator is concerned with the general task of providing a place holder for alternatives and a means of selecting an alternative to be engaged.

The Alternator is related to the Sponsor-Selector pattern [Wallingford98] in that both patterns handle the selection of alternate resources that can change dynamically. The difference between the two patterns is that Alternator focuses on managing the alternate components so that they appear as a single component of an object hierarchy in order to simplify algorithms that operate on the hierarchy. Sponsor-Selector focuses on selecting the best resource for a given task from a set of resources.

The Alternator pattern is similar to the State pattern [GHJV95] in that it the Alternator object appears to alter its class at runtime based on which Alternate is currently engaged. The difference is between Alternator and State is that transitions from one state to another in the State pattern have some regular well defined ordering while Alternator has no notion of transition but rather engages and disengages Alternates through its select operation.

References

CL98	Chan, Patrick and Rosanna Lee. The Java Class Libraries Second Edition, Volume 2. pp 208-220, Addison-Wesly Longman Inc. 1998.
GHJV95	Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesly Publishing Co. 1995.
Microsoft95	Microsoft Foundation Class Library Reference: Part Two, Volume Four. pp 1350-1368, Microsoft Press. 1995.
OSF94	OSF/Motif User's Guide: Revision 2.0. pp 3.28-3.29, Open Software Foundation, Inc. 1994.
ST95	Subramanian, Satish, and Wei-Tek Tsai. Backup Pattern: Designing Redundancy in Object-Oriented Software. Pattern Languages of Program Design 2, Addison-Wesly Longman Inc. 1996.
Topley98	Topley, Kim. Core Java Foundation Classes. pp 566-582. Prentice Hall Inc. 1998.
Wallingford98	Wallingford, Eugene. Sponsor-Selector. Pattern Languages of Program Design 3, Addison-Wesly Longman Inc. 1998.