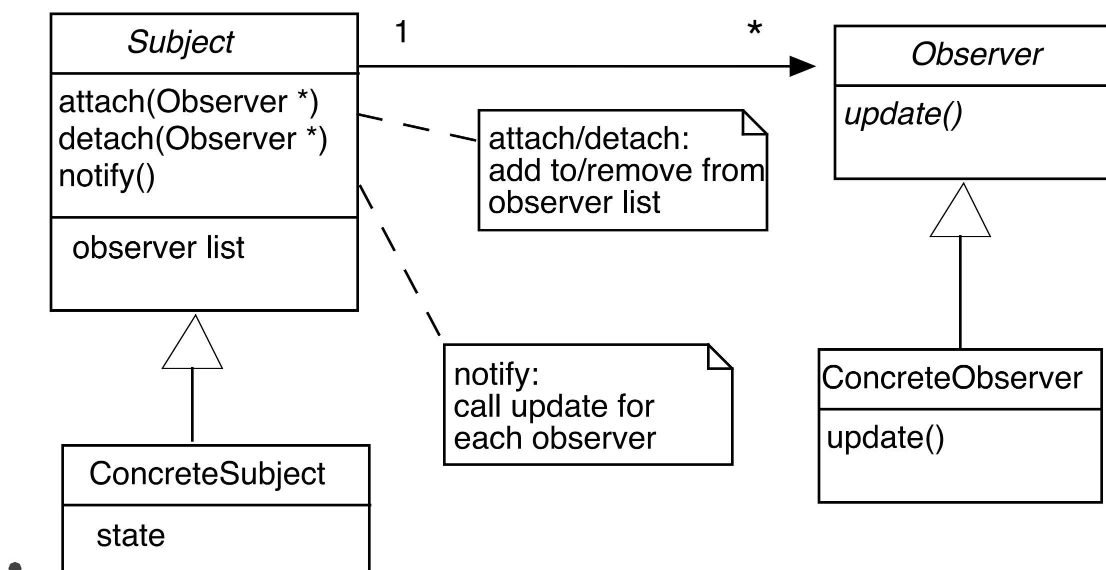
- **Idioms & Design Patterns Behavioral**
- **Introduction to Design Patterns**
  - Patterns and idioms can be grouped roughly into:
    - Creational Patterns and idioms
      - Singleton, Factory Method, Abstract Factory, Named Constructor
    - Structural patterns and idioms
      - Composite, Facade, Adapter, Compiler Firewall
    - Behavioral patterns and idioms
      - Observer, MVC, Double-dispatch
    - Idioms are small-scale patterns, in this context
    - Design patterns come in two basic flavors:
      - Using one or more base classes to hide details from the client
        - One thing about most of the design patterns: presented in a very general form
          - E.g. normally everything has an abstract base class that defines the working interfaces between the parts of the pattern
          - But an actuall implementation might not need this - the abstract base can be collapsed into a single concrete class
      - Other clever ideas using encapsulation, interfaces, class responsibilities to hide details from the client.
        - e.g. Singleton
  - Key concepts for using design patterns:
    - Take the class relationships and the details seriously!
      - You aren't taking advantage of the design pattern just by having some classes with the "buzzword" names organized kinda like the pattern. The exact way in which the classes relate to each other, and how key details are handled, is where the real power of the pattern is!
    - The goal of many of the patterns is to achieve easy extensibility, at the expense of some verbosity and some run-time overhead.
      - This means that a special case solution to a design problem will take less code, and maybe run faster, but it will be much harder to generalize when new features and capabilities get added to the code.
      - The need to extend a program is very common, even if it wasn't planned for.

    - Either use a design pattern from the beginning, to allow for future extensibility, or be ready to refactor the special case solution to make use of a design pattern so that future extensions will go more smoothly.

- *So the goal of the patterns is not short code, or fast code, but easy-to-extend code.*

    - *Don't wreck the pattern to make the code shorter or more efficient - you're missing the point!*

- **Pattern: Observer**
  - Example usage: e.g. different windows that show different views of data for the same data set, stay up-to-date as data changes; add/remove window views at run time
  - Purpose: Allow an arbitrary collection of objects to depend on another so that when it changes state, all of the dependent objects are notified and updated automatically. You need to notify a varying list of objects that an event has occurred.
  - Solution: A Subject object keeps a list of Observer objects. Whenever the Subject changes state, it broadcasts an update event to all of its Observers. The list of Observers can change at run time.
  - Advantage: Decouples Observers and Subjects
    - The update and accessor interfaces define how they interact; - keeps e.g. display of data decoupled from generating the data
    - Which observers are observing which subjects can be defined or changed at run time, which also means additional observer types can be added without changing the Subject code.
  - Abstraction: Observers can be a base class so that different Observer classes can share the same interface and do different things. Likewise for Subject.
    - Often, there is only one possible Subject, so the base class and the Concrete Subject are combined into a single concrete class.
  - Basic Pattern principle: Figure out what could vary, and hide it - usually behind a base class.
  - Also called publish/subscribe, closely related to Model-View-Controller
  - UML picture
    - 

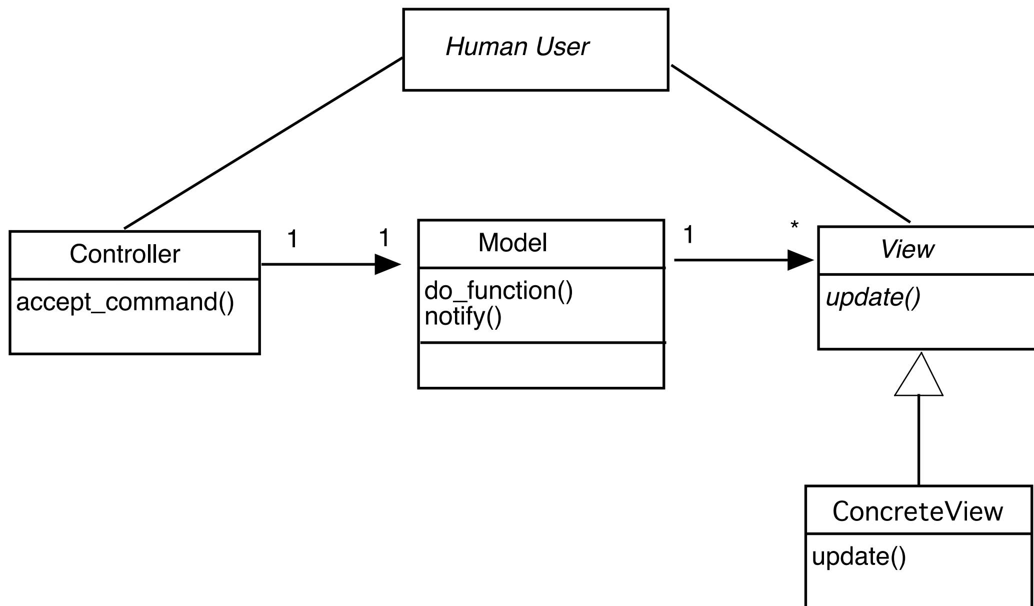
  - Subject:
    -

- Subject optionally is a base class; could be a concrete class. Maintains the state of the data with other member functions (not shown).
- Subject has container holding Observer base class pointers and attach/detach methods to add/remove Observer pointers.
- Subject has notify function(s) that broadcast updates to all current Observers
    - often one notify function for each kind of update.
- Subject knows nothing about what kind of Observers are involved and treats them all equally: if they are currently attached, they get all update broadcasts.
- Subject is not responsible for what Observers are created, attached, or detached. The client code is responsible for managing the Observers.

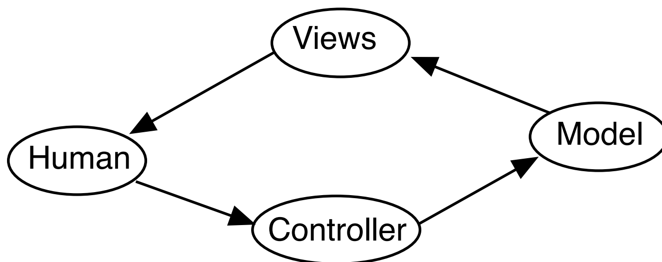
- Observer:
    - Observer is a base class for specific kinds of Observers, with virtual update functions typically declared as empty implementations in the base class.
    - There may be multiple update functions depending on the kind of data being broadcast.
    - Different Observer classes will do different things with the update data.
    - If an Observer is not interesting in a particular kind of update, it won't override that function; if it is, then its override does whatever that class is interested in.
    - How is data passed from Subject to Observer?
        - Affects coupling between Observer and Subject - does Observer have to know about Subject:
        - Pure Push - Subject pushes information to Observer by including all relevant information as parameters of Observer update function(s).
            - Pro: Often, Observer can be completely decoupled from Subject:
                - Observer code does not depend on Subject in any way.
            - Con: Observer interface more complicated - typically one update function for each kind of data or event, possible multiple parameters.
                - Important: Usually best to have multiple update functions, one for each kind of data that changes separately from others.
                - No advantage of minimizing number of update functions by “bundling” data parameters that do not change simultaneously.
        - Pure Pull - Observer pulls information from the Subject; Subject has appropriate public accessors. The update from Subject is only a signal that something has changed; each Observer decides what information to get from the Subject.
            - Pro: Obsever interface can be very simple.

        - Con: Often needs complex accessors to Subject's data, and typically end up with some dependence on internals of Subject. - conflicts with major advantage of the pattern.
    - Recommended: Try to do push as much as possible - usually works better.
- Client code: creates/interacts with the Subject, creates/destroys Observers and attach/detach them from Subject

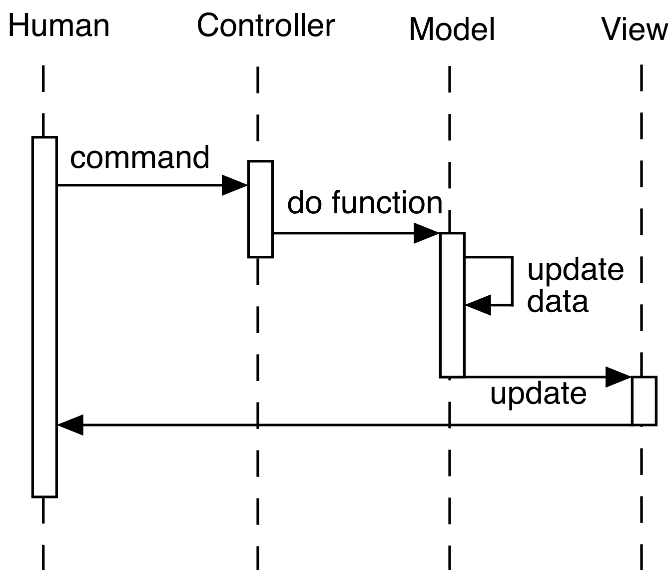
- **Pattern: Model-View-Controller**
  - Elaboration of Observer pattern for user interfaces
  - A good approach to the problem of how to separate the functionality "engine" underlying an application from the User Interface code.
    - Important because there is a tendency for the functionality code to get intertwined with the UI code, making it difficult to change either one - either for revisions/modifications, or to port to a different UI library or platform
  - Based on Observer:
    - The "Model" is the Subject - contains the application's data and the functionality engine.
      - Has interface to access and control state
    - The "Views" are the different windows, etc, that function as "Observers" - dynamically changeable displays of the state of the application
      - are driven by update notifications from the model
      - either "pull" the data out of the model when they need to draw
      - or the model "pushes" the data into the View, which remembers the relevant data and draws it when told to draw - common GUI pattern
        - See Observer pattern discussion on advantages of “pure push” design, and choice of notify/update functions and parameters.
    - The "Controller" is the module that the human user uses to control the application.
      - controls the Model
      - creates/destroys views, and attaches/detaches them from the model
      - in command-language interfaces, usually only one such object
    - UML diagram



- Flow of control:
  - human user operates the Controller to tell the Model what to do. Model tells the Views what has changed. Human looks at the Views to decide what to do next.

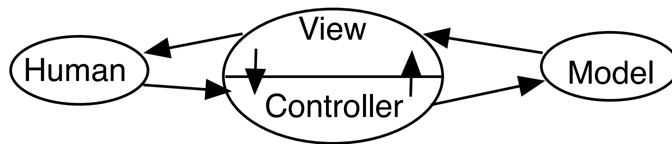


  - UML sequence diagram



- In GUIs, common for the View and the Controller to be tightly coupled, Model-(ViewController)

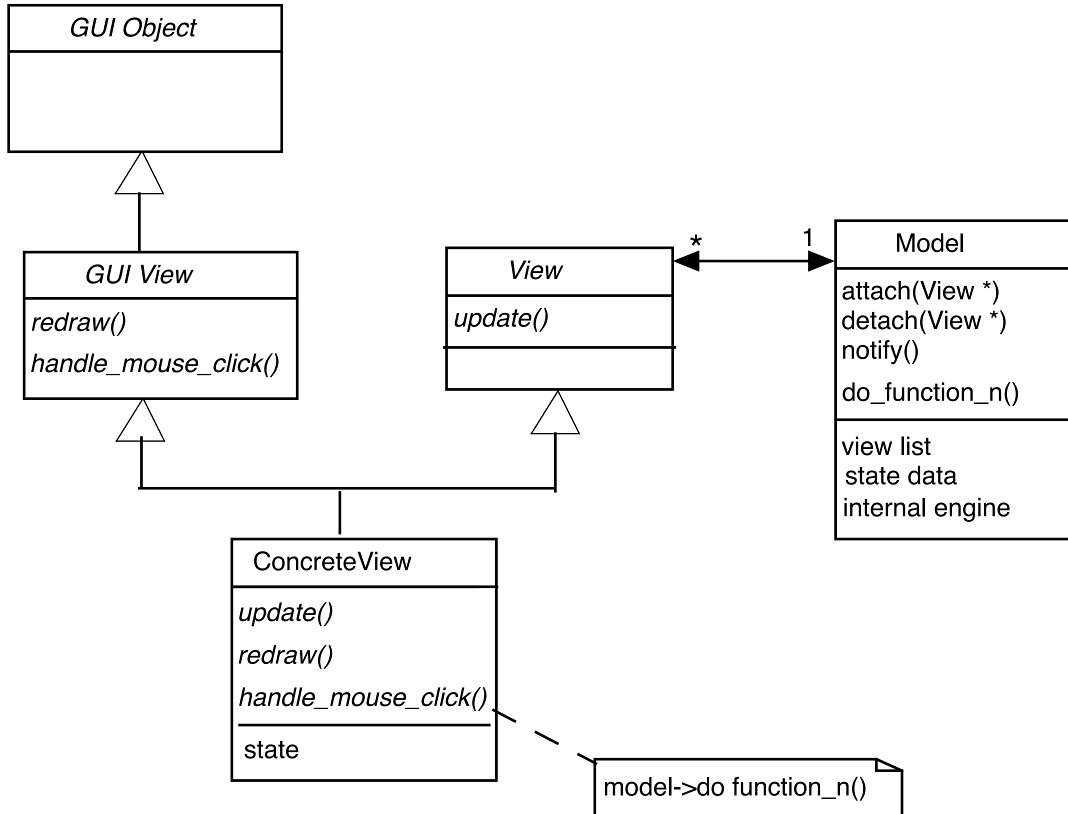
- e.g. user tells Model what to do by using Controller functions tor selecting or manipulating objects on the View.



- important distinction: two levels of MVC logic in a typical GUI
    - GUI "widgets" - e.g. buttons, menus
        - typically predefined customizable classes
    - application window display - e.g. drag operations on application-specific contents
        - typically completely application-specific
- Typically each manipulatable object on the display is a distinct ViewController object
    - so there are many possible ViewControllers

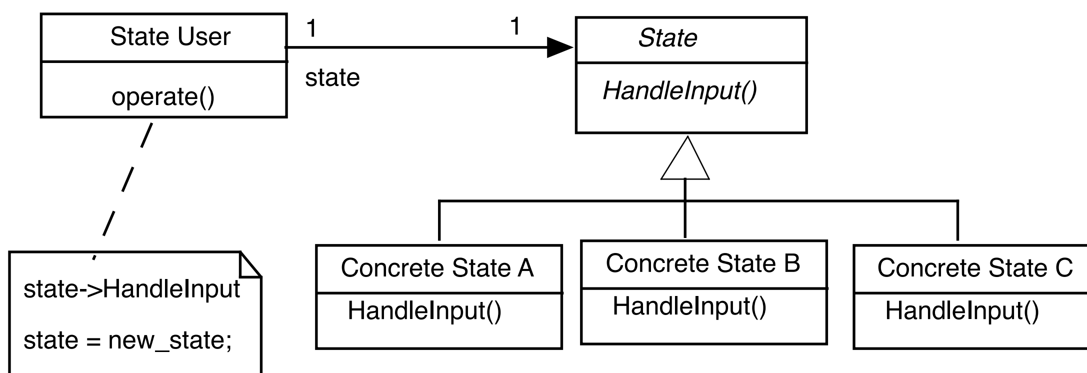
- Key to success is enforcing the strict flow of control, strict division of responsibilities, and decoupling
    - Controller controls the model which updates the View
        - Mischief, difficulty happens when updating is not triggered by the Model
    - See Observer discussion for pros and cons of “push” versus “pull” concepts of how Views get information about data held by Model.
    - Model has absolutely no responsibilities related to creating, destroying, or controlling the Views, or knowing what kind of views there are. Model has nothing but a container of View base class pointers, and the attach/detach functions must work strictly in terms of the pointer values, not member functions or member variables of the Views.
        - If Model is relied upon to look up individual Views, or has any knowledge of View types, or calls anything except the update functions, the responsibilities have been muddled, and the pattern broken.
        - Model just provides updates to Views, absolutely nothing more. In a typical GUI application, either Controller or the OS window manager will invoke the View draw function to update the display. Likewise Views know when their data has been changed and the display is no longer valid. Controlling or dealing with when the display gets drawn is absolutely not a Model responsibility.
- in typical GUI library, there is a class hierarchy of GUI objects
    - Do not want to couple Model views with the GUI views
        - Either have to modifiy GUI library, or else Model has to couple to specific GUI library
    - Use adapter pattern to adapt Model View class to GUI Class hierarchy - a "class adapter" works especially well.

- Can use Multiple Inheritance to attach the Model View's interface to the appropriate class in the GUI library



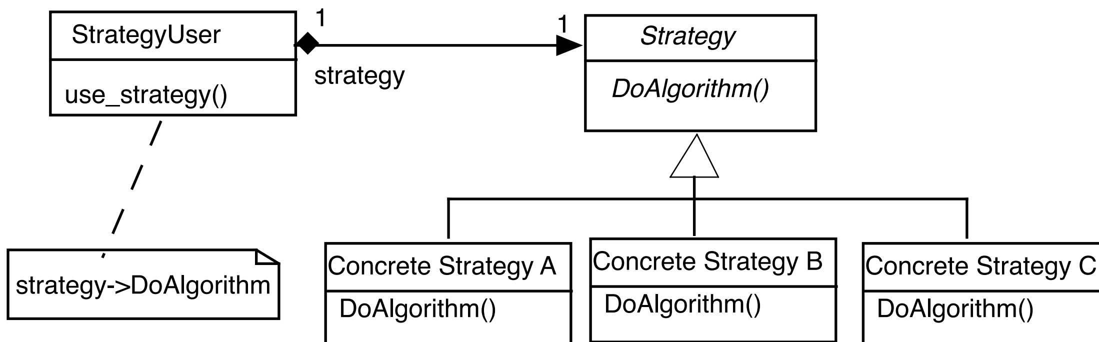
-

State Pattern

- Problem: You have a system consisting of a bunch of functions that change their behavior depending on a relatively small set of state changes. Traditional solution is to have lots of tests or switches on the state in the functions.
- Instead, have a class for each state, whose functions behavie appropriately for that state. Use a base class to define the common interface for all of the functions. Change state by changing a pointer to point to an object of the appropriate type. Call the functions through this pointer.
  - Note that you only need one object for each state.
  - Note that its fine if the state objects have no member variables.
- 

- What varies: The system state. Hidden under a base class.
- Warning: Use with care: This pattern can easily turn into overengineered complexity for simple state machines.

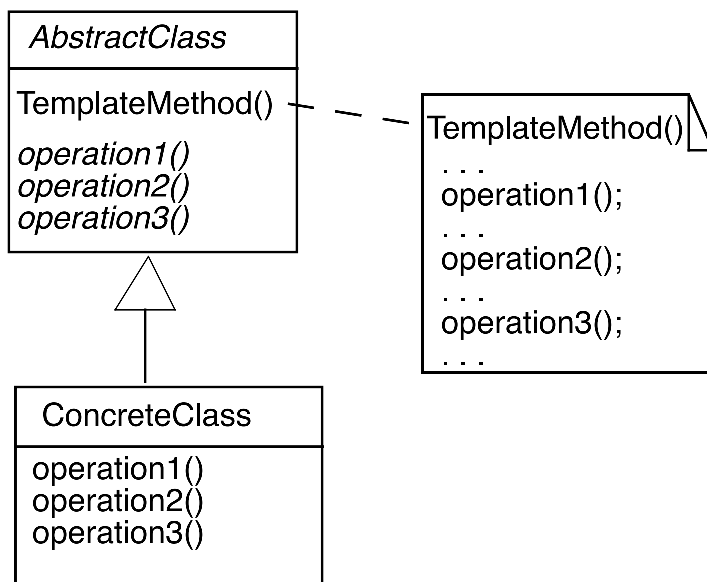
Strategy Pattern

- Similar to state:
- Problem: Allow for changes in the algorithm or strategy used to do something, both for future extensions, and run time changes.
- Solution: Represent the common interface for all of the algorithms with a base class; instantiate the concrete class for the required strategy, and call it. The strategy can be changed at any time by using a different concrete class object.
- 

- What can vary: The specific algorithm; Hidden under a base class.

"Template" Method Pattern

- Problem: You have a lot of different ways of doing a basic process, but some of the individual steps are different depending on which particular way you are using. But the basic process is always the same.
- Solution: A base class has a "template" method that represents the basic way the process is done - the "template" for the process (this is NOT a C++ template). The steps in the process are done by calling virtual member functions in the same class. Concrete derived classes override these virtual functions to represent the specific way the steps should be done.
  - Note that the base class method does virtual calls down to virtual functions overridden by the derived class. Unlike the usual pattern where a the most derived virtual function is called directly by the client, and then might call base class functions. This base-to-derived virtual calling sequence is the "Hollywood principle." - Don't call us, we'll call you.



- What varies: the exact content of each step; hidden behind virtual functions, while base class template method defines what stays constant

Non-virtual Interface Pattern

- Resembles Template Method Pattern
- Problem: You have a polymorphic class hierarchy, but need to separate the implementation provided by each derived class from the base class interface. For example, suppose you want all the virtual functions to share some behavior, differing only in specifics associated with derived classes. You forsee a need to change this sort of thing often.
  - E.g. you want all of the functions to check preconditions, generate some logging information at beginning and end of execution.
- Note that conventional public virtual functions supply both interface and derived-class specific implementation.
- Solution: Make the base class public interface functions NON-VIRTUAL. This means they will always be called, not bypassed like virtual functions would be. They call the virtual functions to allow for different implementations. These virtual functions are protected, in the derived classes need access to them, or are private if not.
  - Like a template method setup, except for removal of the virtual functions.from the public interface.
- Allows the public interface into the class hierarchy to be stable and always involved because it is non-virtual.
- Allows the implementation to be done in terms of virtual functions that are not in public interface.
- NVI version - zap() is the public interface; behaves as if virtual but with common behavior
  - ```
    #include <iostream>
    using namespace std;
    class Base { public:
         void zap() { // the "template method"
              cout << "Start zapping!" << endl;
              defrangulate();
              degauss();
              transmogrify();
              cout << "Done zapping!" << endl;
              }
    protected:
         virtual void degauss()
              {cout << "Base deguass" << endl;}
    private:
         virtual void defrangulate()
              {cout << "Base defrangulate" << endl;}
         virtual void transmogrify()
              {cout << "Base transmogrify" << endl;}
    };

    class Derived0 : public Base {
    private:
    };

    class Derived1 : public Base {
    private:
         virtual void defrangulate()
    ```
    ```

```
            {cout << "Derived1 defrangulate" << endl;}
};

class Derived2 : public Base {
private:
     virtual void transmogrify()
            {cout << "Derived2 transmogrify" << endl;}
     virtual void degauss()
            {cout << "Derived2 deguass" << endl;
            Base::degauss();// do Base's version also
            }
};

int main()
{
     Derived0 d0; Derived1 d1; Derived2 d2;
     (&d0)->zap();
     (&d1)->zap();
     (&d2)->zap();
}

/* Output
Start zapping!
Base defrangulate
Base deguass
Base transmogrify
Done zapping!
Start zapping!
Derived1 defrangulate
Base deguass
Base transmogrify
Done zapping!
Start zapping!
Base defrangulate
Derived2 deguass
Base deguass
Derived2 transmogrify
Done zapping! */
```

- For comparison, try writing out the conventional virtual public interface version: Getting the sequence of execution has to be duplicated in the derived classes.

- **Patterns for Reusing Base Class Functionality in Derived Classes**
  - Problem: You need to implement some classes that share substantial functionality in addition to interface. The shared interface will consist of functions that are virtual in the base class, and the shared functionality will be placed in member variables and member functions of the base class. Some of these will be public, others may be protected, depending on what they do.
    - Remember that having shared member variables in the base class by themselves does not provide shared functionality - shared functionality means both shared member variables and the member functions that operate on those shared member variables.
  - However, derived classes typically need to behave differently in some ways, meaning that often the base class functionality has to be *customized* in some way for each different derived class.
  - A good design puts as much shared code in the base class as possible while still permitting derived classes to customize the behavior in straightforward ways. This maximizes code reuse, giving single points of maintenance, and make it easier to ensure that Derived classes behave in a consistent way when appropriate.
  - Solution: Three basic techniques you can use, in whatever combination works well:
    - Base class functionality implemented with virtual functions that are overriden by Derived class functions to provide different behavior; Derived class virtual functions can call Base class functions in a customized pattern to produce the Derived class behavior.
      - A Derived class virtual function can call Base class version of the same virtual function. This enables the Derived class to provide its functionality before or after the Base class functionality is used.
      - The Base class can have protected member functions that provide services that Derived classes can call, possibly with parameters, in a pattern that will produce customized behavior.
    - Base class functionality includes options or parameters for the Base class behavior. Derived classes set these options or parameters as needed to change behavior of Base class functions.
      - Options or parametes can be set either at construction time via Base class constructor parameters, or after construction, Derived class member functions calls Base class setter functions called by Derived class member functions.
    - Base class functionality implemented with a member function following a template method pattern or non-virtual interface pattern. This Base member function does all of the shared functionality, but does "Hollywood" calls to Derived class virtual functions to customize portions of that functionality. These Derived class functions typically do small amounts of customized work that fit into the overall pattern provided by the Base class template method.
      - When applicable, produces a very elegant design because the Base class function produces the overall pattern of the work, and typically most of the shared detailed work, with the Derived classes filling in their specific details in a few places.
      -

- Typically easy to refactor or extend if new Derived classes have different needs.
- Note that typically the Base class template method member function is not overriden by Derived classes (it might be marked “final”), and depending on the class structure, might not even be a virtual function.

Memento Pattern

- Problem: One component needs to save and restore the state of another component, whenever it wants, and without having to know anything about the internals of the component.
- This is not a solution to the problem of saving program state to a disk file - that's a different issue.
- Solution: The controlling component asks the component whose state is being saved/restored (the Originator) to create a Memento object whose contents are private for all other components. The Originator gives the Memento object to the controlling component (the Caretaker). When the Caretaker wants the Originator to return to a previous state, it gives the corresponding Memento object to the Originator, who then uses the information in the Memento to restore itself.
    - In C++, Memento should have all members private, but declare Originator as a friend; This encapsulates all of the state information.
- Sketch example:
    - multiple states stored, and a chosen one restored
    - ```
      class Memento {
      private:
          friend Originator;
          int i;
          int j;
          Memento();
      };

      class Originator {
      public:
          Memento * create_memento() {
              Memento * m = new Memento;
              m->i = i;
              m->j = j;
              return m;
              }
          void use_memento(Memento * m) {
              i = m->i;
              j = m->j;
              }
      private:
          int i;
          int j;
      };

      in Caretaker:
          Originator originator;
          vector<Memento *> saved_states;
      ```
      ```

```
// save the state
    Memento * m = originator.create_memento();
    saved_states.push_back(m);

// restore state n
    Memento * m = saved_states[n];
    originator.use_memento(m);
    // originator is now in state n
```

- What varies:
  - details about what might be part of the saved state - hidden inside the object;
  - what the caretaker might do with it - not part of originator's responsibility

- **Chain of Responsibility**
  - a series of class objects in a chain
    - first object in chain is given a request
    - each object either handles it or passes it to the next object
    - decouples originator of request from handler of request
  - Usually implemented with a base class that provides event interface and functionality for linking objects in the chain.
  - enables each object to respond to the commands it is interested in, with objects being given their chance to handle it in a specified order.
  - often used in GUIs -
    - The chain is the nested structure of widgets currently on the screen.
      - E.g. Window contains Dialog Box contains check boxes, text fields, etc.
      - Widgets inherit from a base class that provides the event interface and nesting structure functions.
    - A user generated event is handled by seeing if the lowest-level relevant widget can handle it.
      - E.g. event is left-click
        - If mouse pointer is on a text entry field, then select that field
      - Event is alphameric keystroke
        - If mouse pointer is on a check box, keystroke passed up to Dialog Box to handle
        - If Dialog Box doesn't handle it, pass to Window
        - If Window doesn't handle it, pass to Application
        - If Application doesn't handle it, give to Window Manager (the system) - e.g. "beep"

Command

- put all the information for an action in an object, using abstract interface for them, e.g. virtual void execute();
  - client has e.g. perform(AbstractCommand *), e.g. queue<AbstractCommand *>, etc.
- ```
  class AbstractCommand {
   public:
     virtual void execute() = 0;
     virtual void undo();
   };

  class ConcreteCommand1 : public AbstractCommand {
  public:
     virtual void execute();
     virtual void undo();
  private:
     // member variables
   };
  ```
- objects can be kept in a queue, passed from one place to another, created by different parts of the program
  - e.g. command line, menu item, button could all create the same "save" command object, with its data (e.g. file name).
- when it is time to do the action, call its execute method - does whatever needs to be done.
  - necessary information is stored internally
- To undo the action, call its undo() method - the object has all of the necessary information stored internally, and the method knows how to reverse the operation.
- An example
  - my action processor - a combination of template method and command objects
    - the action processor has a template method
      - each action is first prepared, then executed, then finish step done.
      - can be processing an action in each stage, but each one can be entered only if previous action has completed it.
    - action objects have a common prepare, execute, finish abstract base class
      - each specific action class overrides these functions according to the kind of action it implements.
    - elsewhere in the system, a series of action objects are created and then sent to an action processor
  ```

- **Double dispatch, multimethods - function called depends on the type of more than one object**
  - *Stroustrup 22.3 has a discussion on this, which includes some of the techniques presented here, but it uses the "Visitor pattern" as a label for the dispatching technique, which misrepresents the purpose of Visitor. In general, Stroustrup is not very good on larger scale OO design concepts like patterns.*
  - In C++, run-time polymorphism depends on the type of one object
    - if f is a virtual function defined in a base class, and p is Base *, then p->f() will call the version of f that is defined for whatever type of object p is actually pointed to.
  - But what if we want to execute a function that depends on the run-time type of more than one object?
    - trivial at compile time and if object types are known - just use function overloading: f(Derived1 *, Derived2 *)
    - But at run time, we normally have base type pointers:
    - Base * p1 = new Derived1;
    - Base * p2 = new Derived2;
    - call a function with p1 and p2 involved, run the version of it that corresponds to Derived1 and Derived1, Derived1 and Derived2, etc.
  - Some examples:
    - Main dimension: are we double-dispatching on two objects from the *same* class hierarchy, or from *different* class hierarchy?
    - Same tree cases - objects are from possibly different classes from the same class hierarchy
      - Collisions in a space game
        - class hierarchy of space objects
          - torpedo, spaceship, spacestation, asteroid
        - collide between torpedo1, torpedo2, between torpedo1, spaceship1, etc
      - Intersecting shapes
        - circle, rectangle, ellipse
        - intersect between circle1, rectangle2, etc.
    - Different hierarchies cases - more common
      - GUI
        - class hierarchy of widget types - windows, buttons, checkboxes, text field, etc

- class hierarchy of kinds of input events - mouse cursor entering, left button click, keystroke on keyboard, etc.
- top level of code is a loop around handling events
  - get an event - e.g. left mouse button down
  - determine which widget it is happening in - e.g. window
  - execute a piece of code that depends on the kind of widget and the kind of event
- Event-driven simulators
  - events happen in simulated time
    - class hierarchy of event types
  - processors do things in response to the events
    - class hierarchy of processors
  - what happens depends on the kind of processor and the kind of event
- In general, dispatching code based on two types is pretty common and important - double dispatch - in general, multiple dispatch, multimethods
  - C++ doesn't directly support it at all!!
  - Other languages do - e.g. CLOS
    - call a function foo with two arguments: (foo ob1 ob2)
    - defmethod foo(Derived1 o1, Derived1 o2)
      - {code for this combination}
    - defmethod foo(Derived1 o1, Derived2 o2)
      - {code for this combination}
    - defmethod foo(Derived1 o1, Derived3 o2)
      - {code for this combination}
- So how do you get the same result in C++? Relatively awkward because the compiler doesn't do very much of the work for you.
  - Might-have been -
    - intersect (virtual Shape * s1, virtual Shape * s2);
- Three general ways to get it - all with problems.
- 1. Combination of virtual functions and overloaded functions
  -

- Combination of virtual functions and overloaded functions - different hierarcies case
  - Example: Processor, Event base classes
  - Top level
    - Event * eptr = // next event to process
      Processor * pptr = // next relevant processor
      eptr->send_self(pptr); // execute the function for the combination of event and processor
  - class Event
    virtual void send_self(Processor * p) = 0;
  - class A_event : public Event
    virtual void send_self(Processor * p)
    {p->handle_event(this);}
  - class B_event : public Event
    virtual void send_self(Processor * p)
    {p->handle_event(this);}
  - class Processor
    virtual void handle_event(A_event * e) {}
    virtual void handle_event(B_event * e) {}
  - class Processor1 : public Processor
    virtual void handle_event(A_event * e) {// code for P1 A }
    virtual void handle_event(B_event * e) {// code for P1 B }
  - class Processor2 : public Processor
    virtual void handle_event(A_event * e) {// code for P2 A}
    virtual void handle_event(B_event * e) {// code for P2 B}
  - Pros
    - Slick, extremely fast, correctness enforced at compile/link time
  - Cons
    - Adding a new event or processor type requires modifications in base class
    - Unsuitable for a library that the user is not supposed to have to modify
- Combination of virtual functions and overloaded functions - same hierarchy case
  - Hierarchy of Shapes, compute intersections of them
  - top level
    Shape * s1 = new Rectangle, * s2 = new Circle;
    result = s1->intersect(s2);
  - class Shape {
    virtual bool intersect(Shape *) = 0;

```
    virtual bool intersect(Rectangle *) = 0;
    virtual bool intersect(Circle *) = 0;
};
```

- ```
  class Rectangle : public Shape {
      virtual bool intersect(Shape * s)
          {return s->intersect(this);}
      virtual bool intersect(Rectangle *) {//compute Rectangle/Rectangle}
      virtual bool intersect(Circle *) {//compute Rectangle/Circle}
  };
  ```
- ```
  class Circle : public Shape {
      virtual bool intersect(Shape * s)
          {return s->intersect(this);}
      virtual bool intersect(Rectangle *) {//compute Circle/Rectangle}
      virtual bool intersect(Circle *) {//compute Circle/Circle}
  };
  ```
- result = s1->intersect(s2);
    - s1->goes to rectangle::Intersect(Shape * s)
        - s-> goes to Circle::intersect(Rectangle) {// compute Circle/Rectangle
- Pros
    - very fast, compiler enforces correctness
- Cons
    - if you add another Shape type, base and all sibling classes have to be modified
    - unsuitable for a library that the user is not supposed to have to modify

- 2. Combination of virtual functions and switch on type
    - Commonly seen in GUI frameworks
    - user generates a stream of events - mouse moves, clicks, keys, etc.
        - events have different types - e.g. an enum Event_type.
        - example: mouse button is pressed down - what function should be run?
        - depends on where the mouse is pointing.
    - Widget class has a function
        - virtual void handle_event(Event_type e).
    - each widget (window, button, menu item, etc) on the screen is associated with a region of the screen.
        - base class is Widget
        -

- bool point_is_in_me(Mouse_point p);
- figure out which widget the mouse is pointing to, save as Widget * widget_ptr.
- call widget_ptr->handle_event(event_type);
  - automagically calls the event handler for the kind of widget -e.g. a button.
- Button::handle_event(Event_type e)

```
switch(e) {
   case BUTTON_DOWN:
      /* button has been pressed do button pressed stuff */
   case BUTTON_RELEASED:
      /* button has been released do button released stuff */
   etc
   default:
      /* can't handle this event ignore it  or call base
        class handle_event to deal with it*/
      }
```

- Pros
  - traditional, simple
- Cons
  - tedious code - GUI framework + "wizards" can automate a lot of this;
    - Clever ways also to delegate events so that the switch in a widget only has to switch on the cases it is interested in - using the Chain of Responsibility pattern
  - Can be serious maintenance problem - but not so bad if event types are relatively few and pretty stable
    - Which they tend to be in GUI frameworks - only a few kinds of basic user actions - determined by the standard hardware on the machine.
      - move the mouse, press/release buttons, hit a key on the keyboard
    - But tends to get complicated as the event handling concept is so flexible, used for more than just user-generated events, so opens maintenance problems.
- Slicker version - use a map from typeid to function pointers to replace switch on type
- 3. Look-up
  - Table Look-up
    - Alexandrescu, Andrei, Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.
    - use a 2D vector (of vectors) of function pointers - fill up at beginning, then access at run time void(*fp)(Base1*, Base2*);
    -

- write a registry function that gives each class a unique ID number, store those function pointers at those (i, j) cells
- to dispatch a function call, get Id number for each type, execute the function in that i, j cell
- Pro:
  - very fast, very general
- Con:
  - complex, requires code to ensure registration gets done, table populated.

- Map Look-up
  - Use map<std::type_index, map<std::type_index,, function_pointer_type>> dispatch_map
    - std::type_index is a C++11 wrapper class template for type_info that provides a value that can be ordered with comparison operators (e.g. operator< for a map key) or hashed (for an unordered_map).
    - declared in <typeindex>
    - Argument can be a type name, an object, or derefenced pointer; result is a value unique for the corresponding type
  - populate with function pointers for each combination of types
    - dispatch_map[type_index(Event_A)][type_index(Processor1)] = fp;
  - Retrieve function pointer from object pointers with fp = dispatch_map[type_index(*event_ptr)][type_index(*processor_ptr)];
  - Pro:
    - uses compiler-supported RTTI facility; simplifies code
    - set-up is simpler than the 2d table
  - Con:
    - not as fast as the 2d table - log time instead of small constant time
    - still more setup code than the virtual/overloaded function solution

Some others

- Iterator - support traversal of a data structure without having to expose internal details of the structure or how it is traversed. A fundamental component of the STL - nuff said.
- Visitor - involves virtual/overload application of double-dispatch in traversing a data structure and applying an operation in the form of a Vistor object to each node or the structure. Makes it easier to add additional operations without modifying the data structure classes, but requires that each node in the data structure make its state fully available to the Visitor.