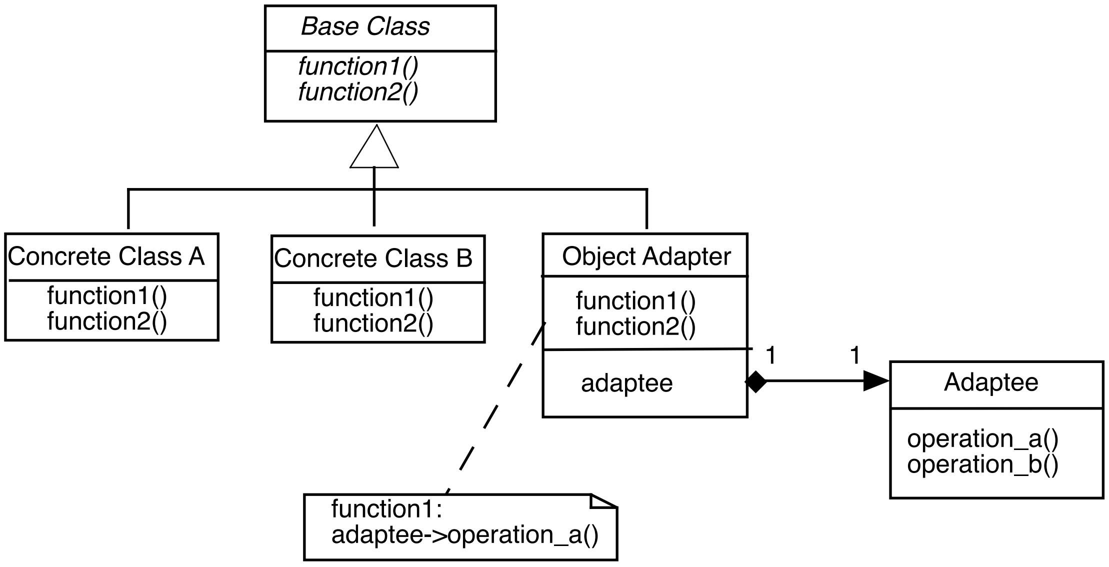
- **Idioms & Design Patterns Structural**
- **Introduction to Design Patterns**
  - Patterns and idioms can be grouped roughly into:
    - Creational Patterns and idioms
      - Singleton, Factory Method, Abstract Factory, Named Constructor
    - Structural patterns and idioms
      - Composite, Facade, 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.*

- **Wrapper Idiom**
  - Hide the details of a pre-existing facility inside a class, provide for initialization and deinitialization with constructors and destructors, and a better interface.
  - Example: wrap C file I/O in a class:

    -
      ```
      class Output_file {
         public:
            Output_file(const char*  filename, const char* mode); // use fopen()
            ~Output_file();   // close the file with fclose
             operator bool(); // test the state of the file stream
            Output_file& operator<< (int);     // output an integer
            Output_file& operator<< (const string&)     // output a string
         private:
            FILE * file_ptr;
         };
      ```
    - in fact, early implementations of iostream and fstream were just wrappers around the C's stdio

- **Facade Pattern**
  - A larger-scale version of the wrapper idea.
  - Wrap an entire subsystem in a class, hiding the whole subsystem, and providing a simpler interface to the subsystem and hide all of its details. The Facade class constructor might build and configure the subsystem. The public interface might do additional computation and multiple calls into the subsystem components.
  - My production-system example.
    - Public interface is load, reset, and run, plus various mode/output settings.
    - Internals involve 22 separate modules: several dozen classes, a whole collection of templates, very complicated processing - all hidden as private member variables and functions.

- **Adapter Pattern**
  - A wrapper that mates an odd-ball class or facility to a pre-defined interface.
  - Used when you need everything to fit into a polymorphic class hierarchy.
  - 

  - This is an object Adapter - works by delegating to a subobject (via composition).
  - There is also a "class Adapter" - works by inheriting from an interface.
    - e.g. see MVC example where a View base class is used to provide an interface for classes inheriting from GUI library, thereby allowing model to talk to GUI subclasses without becoming dependent on the GUI library details.

- **Idiom: Compiler Firewall: Separating interface from implementation**
  - interface is presented to
    - other classes (public interface)
    - derived classes (protected interface)
  - implementation is how it gets done -
    - private members, contents of fucntions
    - also by non-public inheritance
      - adds capabilities without changing the interface
  - separation of interface and implementation is critical
    - expose only the minimum in the public interface
    - do not provide direct access to private member data
    - C++ permits some of it directly
      - source files supply the implementation
      - header files provide access to the interface
        - other module just includes it to use the class without being exposed to most of implementation details
    - what has to be recompiled if what changes?
      - unfortunately, some of implementation detail is exposed in the class declaration - the private or protected members, what gets inherited.
        - These affect the size of the object. If object declared as local or member var, or allocatd with new, this must be known.
      - means changes to implementation affect other modules ... force a recompile, even if public interface does not change
    - Concept of **insulation** - *insulating part of a system to changes in another part*
  - compiler firewall Idiom
    - Provide insulation - set up classes so that change can't propagate past the "firewall".
    - file Gizmo.h

```
class Gizmo_Impl; // incomplete declaration
class Gizmo {
   public:
      Gizmo();
      ~Gizmo();
      void start();
      void stop();
   private:
       Gizmo_Impl * pimpl;
  };
```

      - This file never changes unless public interface changes
    - file Gizmo.cpp

```
#include "Gizmo.h"
#include "Gizmo_Impl.h"
Gizmo::Gizmo() : pimpl(new Gizmo_Impl) {}
Gizmo::~Gizmo()
```

```
   {delete pimpl;}
Gizmo::start()
   {pimpl->start();}
Gizmo::stop()
   {pimpl->stop();}
```

  - this file will have to be recompiled if Gizmo_Impl.h changes, but change will not propagate further
- file Gizmo_Impl.h

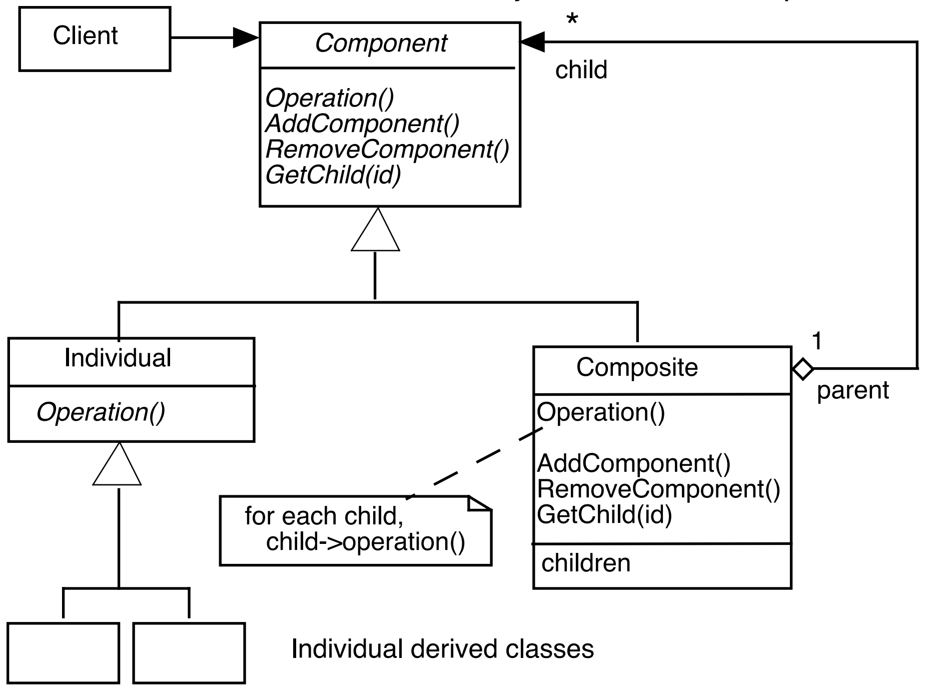
```
class Gizmo_Impl {
.....
```

- file Gizmo_Impl.cpp

```
.....
```

  - class implementer can change Gizmo_Impl freely, without having to worry about forcing users (= "client") of Gizmo to recompile for every little change in the private members of Gizmo's implementation

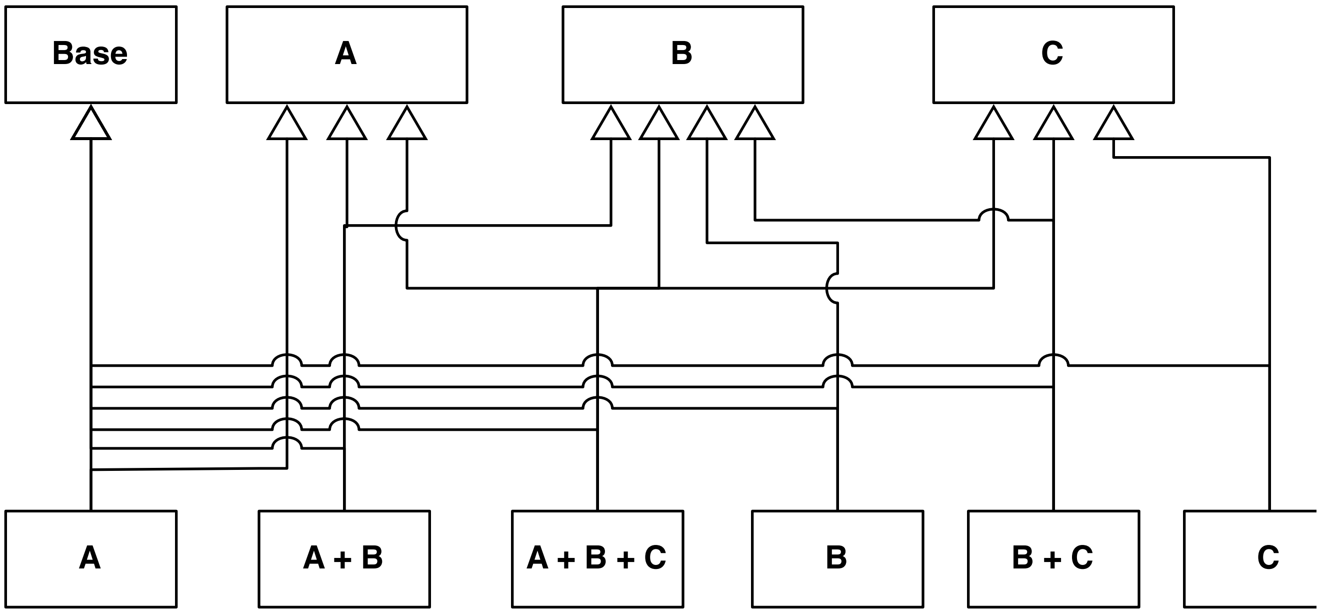
- **Composite Pattern**
  - Problem: You have a collection of basic objects that can be grouped in arbitrary ways. However, the client code needs to be able to operate on all of the objects the same way, even if they are grouped together.
    - Common example - a drawing program - can group shapes together, have them drawn, resized, moved, etc as a group.But individal objects still exist.
  - Solution: A base class represents a Component of the collection. A Component is either a basic Individual object, or is a Composite object. A Composite object contains a container of pointers to other Components (its children). The Composite class has functions to add, remove, or lookup members of the container. You can call an Operation function on a Component; if it is a basic or leaf Individual object, it does the operation. If it is a Composite object, it calls the function on each of its children.
    - Alternative names for the classes: Component is an Interface; Leaf is a class for Individual Objects; Composite represents groups of objects from the Component/ Interface class.
    - The Leaf/Individual class can be more than one class that inherits from Component/ Interface, or the root of a inheritance tree of classes; the concept is that the Leaf/ Individual classes are used to create individual objects rather than a Composite/Group object.
    - Note how the Component class has a fat interface that includes both the functions only needed in the Composite class (e.g. AddComponent) and those implemented in both the Leaf and Composite class (called Operation() in the diagram). It thus serves as the interface for both Leaf/Individual classes and the Composite/Group class.
    - In a good application of this pattern, the Client deals with both Leaf-Components and Composite-Components only through the Component fat interface (as shown in the diagram). The client rarely needs to distinguish the two types.
    - A typical structure of objects at run time will be a tree with a Composite at the root whose children are Individual objects or other Composites.

- 

- What varies: Whether the child is a individual or a composite, hidden under the component interface.
- Warning: It can be tempting to try to collapse two of the three classes (Component, Leaf, Composite) together into a single class. This is NOT a good idea - serious violations of the basic principle of inheritance can result, causes many bizarre problems in the design.
  - Example: Let Individual be an Agent in a game/simulation, in which the Client code controls the Agents (tells them what to do). Then Component represents the "controllable" things - the things that the client can control, by virtue of the Component interface. Composite then represents a group of controllable things. The controllable things are either groups of controllable things or individual Agents.
    - Consider what happens if you combine some of these classes:
    - If you make Individual into the Component, you are saying that a group of Agents IS-A Agent. which can't be true. While a football team consists of football players, a team is not an individual person.
    - If you make Composite into the Component, You are saying that an individual Agent IS-A group of Agents, which can't be true. The person playing quarterback is not a football team.
  - Keep the classes distinct: Component provides the common interface to both individuals (leafs) and groups (composite), and allows the client to communicate with either individuals or groups in the same way. A group contains a collection of Components, which in turn are either individual or groups.
- Warning: Because of the fat interface in Component, it is almost always possible for the client code to refer to both Individuals and Composites through Component pointers only. If your client code for example has separate containers for Individuals and Composites, you are missing a key advantage of this pattern, and might even be misapplying it. On those probably rare occasions where the client might need to distinguish a Composite from Idividuals, note that a dynamic_cast<Composite*> will suffice.

- **Flyweight**
  - Use object-oriented techniques for a very large number of objects, without incurring storage overhead for a large number of objects.
    - Abstracting what to do from the specifics.
    - The Flyweight object is a package of functionality with no data.
  - Example: display complex formatted text on the screen by having an object for each character and telling it to draw itself - It draws the appropriate pixel pattern with the appropriate shape and location.
    - Drawing code doesn't have to know what character is being displayed.
    - But number of objects is huge - if they contain their own data (e.g. location), storage becomes ridiculous.
  - Pattern solution
    - Separate intrinsic from extrinsic state; put extrinsic state elsewhere
      - Instrinsic state can be shared by all objects of that type.
      - Extrinsic state can be stored elsewhere or computed as needed.
    - Create one shared object of each type needed, keep multiple pointers to multiple shared objects in appropriate containers.
    - To accomplish something, call a virtual function for each pointed-to object, and supply the extrinsic information. Virtual function for that type of object does the appropriate thing with the supplied information.
  - Example of complex text display:
    - Create one sharable object (the Flyweight) for each ASCII character code.
      - Maximum is total number of distinct character codes.
      - Use a factory:
        - If flyweight already exists, simply return pointer to it.
        - If not, create it and add it to a Pool of flyweights.
    - Each line of text represented by a container of pointers to shared objects.
    - To display the line, compute the position of each character, and give that to each objects Drawself function.
    - Fancy text editors have been built this way; worked well.
  - Other applications for graphical programs are possible
    - Especially in MVC -
    - Data about a bunch of objects is often held independently of how they need to be drawn on the screen - the Model vs. the View
    - Useful way to reduce the redundancy between the "real" objects in the Model data and the View objects.

- **"Forest of Trees"**
  - Problem: You have a base class and need to have many derived classes that combine a few kinds of functionality in a large number of ways. These functionalities do not overlap with each other, or with the base class functionality - they are "orthogonal".
    - We often want to implement functions that are virtual in Base with these functionality combinations.
    - Example: Base is the base class, and A, B, and C are the functionalities (groups of member variables and functions) that need to be combined in lot of different ways in classes that are derived from Base. We want to have leaf classes that have the functionality in A+B, A+C, B+C, just A, none of them, all three, etc.
  - Poor solutions:
    - Non-solution #1: Put all the functionalities into the Base class, abandon the derived classes altogether, to yield a single do-all class and let the client code only use the functionality it wants (somehow). Design fails to express the situation, and leads to clumsy code.
    - Non-solution #2: Have the single base class, with a derived class for each combination, and allow the functionality code to be duplicated as necessary across the derived classes. Design fails to express the situation, and causes a nightmare due to serious violation of single-point-of-maintenance rule.
    - Clumsy solution #1: Do the best you can with a class hierarchy that minimizes the amount of duplicated code. Not quite as bad as Non-solution #2.
    - Clumsy solution #2: Use a single base class with intermediate derived classes, one for each kind of functionality, and leaf classes inherit from multiple intermediate derived classes. Expresses the design concept, but at a severe price of complexity due to the multiple cases of diamond-shaped inheritance.
  - Better solution: Use a set of orthogonal base classes to provide mixin multiple inheritance for the functionalities. Each derived class inherits from whichever base classes it needs for its combination of functionality. There is no single root for the whole class structure, but rather multiple trees starting at each base - a "forest of trees".
    - Example: There is a Base class, and additional base classes for each functionality, A, B, and C. Derived classes all inherit from Base, but each also inherits from one or more of the functionality bases depending on the combination of functionality each one needs. With only the four base classes, we can represent all eight possible combinations of the A, B, and C functionalities. The diagram shows a subset of seven combinations. In effect, there is a tree for each of the four base classes, with the most derived classes being leaves on multiple trees (which of course doesn't happen with actual trees!).

- 

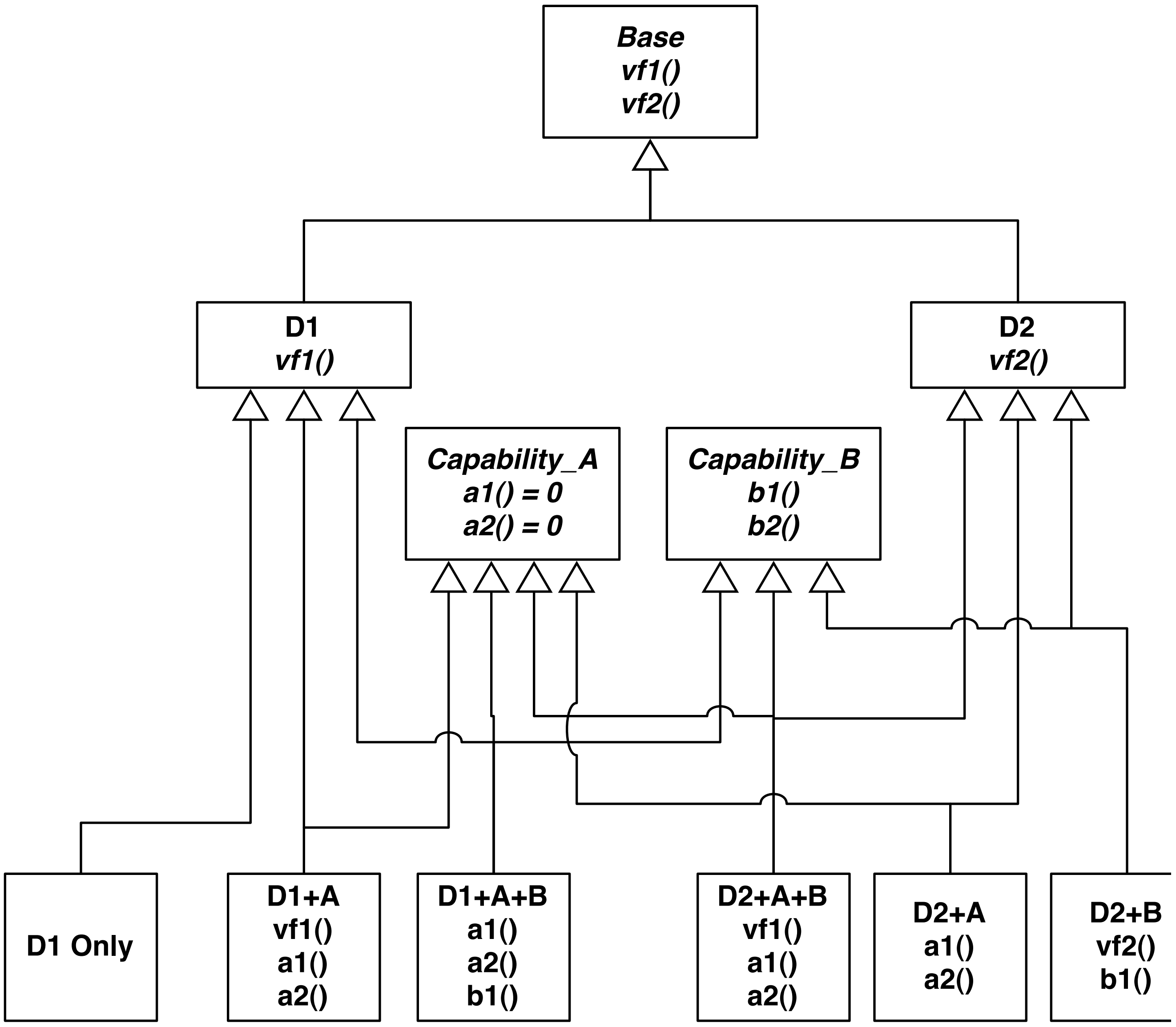
- This solution works well for representing a large number of different combinations of functionality groups. E.g. with only the three A, B, C base classes, we can have as many as 8 unique combinations, but we can use whichever subset of these we like. E.g. above diagram does not include the A+C combination, or the Base-only combination.
- Consistent with the guideline to avoid multiple inheritance when possible, do not adopt this solution if there is a way to formulate the problem equally well in a simple single-base single-inheritance form.
- If you are referring to leaf-class objects through a pointer to one of the base classes, you will have to do a cross-cast (with dynamic_cast) to access the functions provided in a different base class.

- **Capability Classes and Queries**
  - Problem: You have a class hierarchy with an interface defined with virtual functions declared in the base class, and derived classes override these to provide customized behavior. However, some derived classes have additional behaviors that are not included in the base class. A fat interface in the base class is not desired, but the client code needs to be able to call these functions if the object is from a class that has them.

    These additional behaviors come in groups, which are non-overlapping (orthogonal), and more than one derived class might need to include functions from one or more groups. In fact, there might be different combination of these non-base-class functions used in derived classes, and more might be possible in the future. A single inheritance tree cannot represent these combinations in a simple way without duplication or diamond-shaped inheritance emerging.
  - Solution: A variation on the Forest of Trees. In addition to the shared Base class, the mixin classes also serve as interfaces to the derived classes that inherit from them.

    In the example shown in the diagram, the Base class declares a virtual function interface that applies to all of the classes, but the intermediate and leaf derived classes may override these functions as desired. The Capability Class A declares an additional virtual function interface that is inherited by some of the derived classes, and whose functions might be overridden as needed. Similarly for Capability Class B. The capability classes are often pure interface classes (as shown for Capability A), but could also have functionality that could be overriden in derived classes (as shown for Capability B). Each leaf class object thus supports the Base class interface and zero, one, or two of the Capability Class interfaces.
    Different combinations of Base class virtual function overrides, Capability Class inheritance, and Capability Class virtual function overrides provides extreme flexibilty in customizing the leaf class behavior.

•



- All objects share the Base class interface. For example, calling one of the Base class functions through a Base pointer results in a virtual call to the implementation of that function corresponding to the exact type of the object:

  -
    ```
    Base * p = address of some leaf-class object;
    p->vf1(); // calls the appropriate version of vf1 depending on the object
    type.
    ```
- Since all objects share this interface, such a call is always valid. However, we can't call a Capability_A function for `p` for two reasons: These functions aren't declared in the Base interface, and the object pointed to by `p` might not be one that inherits from Capability_A. So even if we could do the call somehow (e.g. with a cast), the results would be undefined if `p` didn't point to an object from one of the classes like D2+A.

  -
    ```
    p->a1();  // illegal - no such function declared in Base
    ```
- We can call one of the Capability A functions only if the object is from a class that has the Capability A interface. We determine whether it does with a *capability query* - we ask the object: "Are you from a class that supports Capability A?" If so, we can treat the object as a Capability A object and call the function. This is done with dynamic_cast:

-
  ```
  Base * p = address of some leaf-class object;
  Capability_A * a = dynamic_cast<Capability_A *>(p);
  if(a)
       a->a1();  // p pointed to a Capability_A object, so call is valid
  else
       // p did not point to a Capability_A object, so we cannot use that
  capability.
  ```
- Note that the dynamic_cast tests for whether the object inherits from the Capability class, not whether it is an instance of a specific leaf class. The question is not for example, "Are you an object from class D2+A+B?" but rather "Do you have the Capability_A interface"?
- If the dynamic cast fails, it is an error of some type. We would not try to use a specific capability on an object that lacks it unless something was wrong (e.g. the user entered an incorrect command). This process is not switch-on-type because if the dynamic cast fails, we do not go on to query another type for the object.
- This can be an attractive solution if the situation is similar to the Forest of Trees problem - many different combinations of orthogonal Capabilities need to be provided for, and a single-inheritance tree cannot represent the possible subclasses without duplication or diamond-shaped inheritance.
- However, if the problem can be satisfactorily formulated as a single-inheritance tree with a (possibly fat) interface defined in the base class, then that simple solution should be preferred, because the virtual function mechanism will do all of the work - no need to query object capabilities in order to decide whether to call a function, just call the function. "Don't ask, just tell!"

- **Some others in Gang of Four book**
  - Decorator
  - Proxy
  - Bridge