Design Patterns Summary

The editor example in your text gives a good summary of several patterns along with illustrations of how they are applied.

Here is another summary of patterns, organized into several categories, with the context and structure of each each briefly explained.

Factory Patterns

Solutions to the problem that when you define an abstract interface and then subclass it with concrete classes, there are no constructors for the concrete classes that can be invoked from the abstract interface.

- **Builder**
  Separate factory class, typically named after the interface class, containing one or more methods eahc of which returns an instance of a concrete class that implements the interface.
- **Abstract Factory**
  Multiple factories (builders) to create different categories of objects. Make an abstract class to provide general abstract interface for several related factories. Provide concrete subclasses for the different builders.
- **Factory Method**
  Implement a factory without making a separate class... merge the factory class into an existing class. Do this when factory class flexibility is not needed, when an appropriate class to merge with is obvious, and when factory methods are few. This pattern allows the factory method(s) to be inherited.
- **Singleton**
  Used to create only one object instance of a class. Keeps track of whether the constructor has been invoked, and once invoked, subsequent constructor invocations are trapped and the existing pointer returned for the singleton object. Often used to create factory classes... since there needs be only one builder of each type. Also can be used to create global data as object storage.
- **Flyweight**
  Useful when application needs to share objects; helps prevent big proliferation of small objects, like nodes in an expression tree. Implemented as hash table (for example) of pointers to singleton objects. Also used for character (glyph) objects in text editors, etc.
- **Prototype**
  Does not use classes... instead uses prototype objects and clones them. Ask existing prototype object to close itself when a new object of that class is needed.

Delegating Responsibility

Patterns for effectively using delegation to replace inheritance. Inheritance is static and hence lacks flexibility; delegation is dynamic and allows alterations at run-time.

Delegation is passing responsibility for an activity from one object to another. The object doing to delegation must maintain a pointer to the object receiving the delegation. Each operation being delegated must be correctly coded to invoke a corresponding operation in the object receiving delegation. C++ does not provide explicit support for all these mechanics as some newer OO languages do.

- **Adapter**
  Use when the functionality needed for some class is already provided in another existing class. The interfaces need to be brought into synch. Often used to have a windowing class written specifically in application terms, but then to be implemented by delegation to some common windowing package (X, Motif, etc.).
- **Bridge**
  For separating an interface from its implementation. A bridge is a wrapper, a class that completely contains the interface to another class. There is no way to get to the wrapped class except "across the bridge" class. Continuing the analogy, the wrapped class is an island, unreachable except by the bridge.

- **Decorator**
  Chain of bridges... components of the chain can be altered during execution to respond to user functionality selections. Each object in the decorator chain must have the same interface as the object being decorated. Each object adds some functionality or "decoration" to the base object, and this decoration can be dynamically added or subtracted (unlike if this were done via inheritance).

- **Facade**
  Combination of delegations. A wrapper object that maintains multiple pointers to other objects, the combination of which provide the functionality of the wrapper. So the facade has an interface that is partially provided by each object to which it delegates method calls.

- **Proxy**
  An object that can stand in the place of another object and field requests for the methods of the wrapped object; a proxy can even stand in for an object that does not (yet) exist at the time its methods are invoked. Allows creation of the wrapped object at some future time, when it becomes truly necessary.

  For example, in an editor, there may be numerous graphic objects for the drawing, pictures, etc. in a document. There is no need to incur the run-time expense of creating these graphic objects (and loading them with extensive data from disk) if they are not visible on the screen. So cheaper proxys can be generated for the graphics to hold open a place in the document until one needs to be displayed. Then the proxy can trigger creation and loading of the object for which it is fronting.

  Proxies are also commonly used in distributed systems to encapsulate the behavior and interface of a remote server. The proxy in this case differs from a "standard" wrapper in that it will be responsible for the extra detail of creating remote network calls... and the client can talk to the proxy as if it were a local object, simplifying the client code by the labor division.

Control Patterns

- **Composite**
  For creating well organized hiearchies of related objects, in such a way that atomic objects and composites of those atomics can be treated uniformly by a client. For example, in a graphics program, a grouping of shapes should be able to be an element in a drawing just as a single shape... and hence, a grouping could be an element in another grouping. So a composite pattern allows an object that can represent a shape or a grouping uniformly.

- **Interpreter**
  Somewhat specialized pattern. Encapsulation of an algorithm for processing a specialized data structure, usually a grammar for a language. For example, in a compiler, a syntax tree might be processed by an interpreter pattern that knows how to manipulate that tree to determine the code to be generated from the syntax.

- **Iterator**
  Iterator is a specialized form of Interpreter. Its purpose is to traverse a data structure and produce in some sequence all the elements of the structure. The pattern allows this traversal by clients without them having to know the internal structure of the data.

- **Command**
  For encapsulating a user interface so that user sessions can be "traversed" (i.e., undo, redo). User commands are made concrete as data structures.

- **Strategy**
  Allows the substitution of alternate forms of an algorithm, dynamically selected for processing different forms of related data. For example, in a text editor, when text needs formatting the user might select pagenated, or hypehnated, or neither... requiring three slightly different algorithms. Instead of having lots of conditionals in one big formatting algorithm, three slightly different ones can be written and encapsuated in objects that are delegated to at run-time.

- **Template**
  Uses inheritance to replace parameterized portions of an algorithm at compile time... in contrast to Strategy, which uses dynamic delegation to replace entire algorithms at run-time. The template is a concrete method in an abstract class; it invokes abstract methods that are given different concrete realizations by subclassing, thus fleshing out the algorithm template in different ways for different subclasses.

- **Visitor**
  Often used with an Iterator. Visitor object encodes the logic for handling different kids of nodes in a data structure... as Iterator walks a structure, each node the visitor "calls on" the node and processes the data.

Algorithmic Patterns

- **Mediator**
  Manages interaction of collection of objects. Promotes loose coupling by keeping objects that need to interact from referring to each other directly by pointers. Mediator object encodes the interactions that are allowed, manages object references. Could be useful for collaborative distributed systems, where various interaction rules control how users may collaborate.

  One use is for dialog boxes in user interfaces. Mediator object keeps track of all the buttons and widgets in the box, and directs data from one to the other. That way, the widgets do not need to be aware of each other, and they can come and go from the interface dynamically.

- **Mememto**
  An object that can snapshot the internal state of another object (makes a mememto of that object); memento object understands the internal structure, but the various clients that need the snapshots do not have to know the internal structure of the object.

- **Observer**
  Defines a one-to-many dependencies structure among objects. Allow changes in one object to cause propogated changes in the related objects. Used in model-view-controller structures to manages different simultaneous views of data. For example, in a spreadsheet you might have a barchart representation of data in cells, a piechart, and the numerical cells. All need to be updated as the numbers change. The observer will watch for changes in one and trigger corresponding actions in the related objects.