The following are a few design patterns which are not the regular object-oriented design patterns but are useful at some stage in software development:

Strategic programming Design Pattern

(1) Role Play:
The design pattern aims at defining a transformation that can be visualized as pipeline of steps, with roles already designated to each of the steps. Each step has a set of specific responsibilities, namely, analyses, guards, side effects, atomic transformations, etc.
Analysis: Type unifying. Input remains unchanged.
Guard: Condition checking for an input.
Side effect: Realized by access to an extended functionality interface for a state.
Atomic transformation: Type preserving steps.

Thus, each step has a specific delimited role to play in the complete transformation.
It becomes very easy to understand the creation, and modification for each step. The scheme transformation can be represented as a complete pipeline by using suitable composition operators among them.

Transformation = Role*1* (composition operation) Role*2* (composition operation) ... Role*n*

Concurrency Design Patterns

(2) Reactor pattern:
The reactor design pattern can be used as an event handling pattern. It can handle requests concurrently delivered to a service handler by a number of inputs. Once inputs are started to be received, the handler can then separate(demultiplex) them and dispatch them to the request handlers as required. These systems are single threaded but can exist in a multithreaded environment.

It consists of the following components:

- Resources: Anything that provides input or consumes output.
- Synchronous Event Demultiplexer: It starts sending the resources once it is possible to start a synchronous operation without blocking. It uses an event loop to keep a block on all the incoming resources.
- Dispatcher: Dispatches resources from demultiplexer to handlers.
- Request Handler: Application defined request handler and its resources.

Advantages: The application component can be separated into reusable, modular parts. This is made possible because the reactor pattern keeps the application code separate from the reactor implementation. It also allows simple concurrency due to synchronous calling of request handlers.

Disadvantages: It is difficult to debug compared to a procedural pattern. The scalability of this design pattern is limited by certain factors, namely, synchronous calling of request handlers and use of demultiplexer.

(3) Join Pattern:

Join Pattern is used with message passing to write concurrent, parallel and distributed computer programs. Unlike simpler implementation comprising of threads and locks, this is a high-level model. It abstracts the complexity of concurrent environment and focusses on the execution of a process between the messages being communicated among various channels.

This design pattern is based on Join calculus and makes use of pattern matching. It uses concurrent calls and message pattern to allow the join definition of channels communicating for the task execution. It allows for flexibility in communication between the participating entities.

It can be represented as follows with the use of keywords:

- What: Specify the start of communication expected.
- And: Join other channels.
- Do: Run some tasks with the various collected messages.

General form of a constructed join pattern:

j.When(action1).And(Action2)....And(ActionN).Do(Tasks)

Join pattern can also be defined by a set of pi-calculus channels. It supports two actions: sending and receiving, each needs a join calculus names for implementation, namely send and request. A call to x() returns a value that was sent on a channel x<>.

Join pattern can be used with different programming languages; some can use them as a base for their implementation or integrate the join pattern by using a library.