Computer Science Design Patterns

In Computer Science and Computational Thinking there are a number of common tasks, strategies, or algorithm designs that are often called for. As you become increasingly familiar with these *design patterns*, these patterns can be applied more and more easily as required in other contexts.

1. **Counting Events**
   You'll often need to count the number of times an event occurs. To use a counter, you'll need to declare it and initialize it to some value (usually zero), add one to the counter when the event occurs, and clear the counter if it turns out that the count needs to be restarted for some reason.

   Example: Counting number of games played, and wins

```
/**
 * The win method updates current game statistics
 */
public void win()
{
    this.gamesPlayed++;
    this.wins++;
}
```

2. **Keeping a Running Total**
   In many situations, a record is kept of some value, with changes made as events occur. A BankAccount balance may receive deposits, or pay out withdrawals. A game character's "life points" may go down as she takes damage, and up as she is healed. A car's gas tank may have gas used while driving, and then be refilled.

   Example: Betting money in the game of Blackjack

```
/**
 * The win method updates current game statistics
 */
public void win(double betAmount)
{
    this.money += betAmount;
}
```

3. **Keeping Track of a Series of Values or Objects**
   If you need to keep track of a number of values or objects, an `Array` or `ArrayList` data structure is required. A list of `Words`, a collection of `BankAccounts`, a record of chess moves in a chess game, a list of items in a `ShoppingCart`, all can be handled by an `Array` or `ArrayList`.

   In Java, `Arrays` are simpler to work with: they can be declared to hold any type of primitive or object data, and individual elements are directly accessed via their index value. `ArrayLists` can be dynamically resized, however, and may be a better choice in some cases.

   Example: Managing a list of guests to a party

```
import java.util.ArrayList;

/**
 * The PartyPlanner class manages your social events
 */
public class PartyPlanner
{
    private ArrayList<Person> guests;
    private int maxGuests;

    /**
```

```
     * Initialize the Planner
     * @param maximumGuests the maximum number of people who can come
     */
    public PartyPlanner(int maximumGuests)
    {
        guests = new ArrayList<Person>();
        maxGuests = maximumGuests;
        .
        .
        .
    }

    /**
     * The addGuest method adds another person to invite list
     * @param newGuest The Person who is invited
     * @return 0 if person was added, -1 if it failed
     */
    public int addGuest(Person newGuest)
    {
        if (guests.size() < maxGuests)
        {
            guests.add(newGuest);
            return 0;
        }
        else
            return -1;
    }
```

4. **Managing Objects via Attributes and Methods**
Programming languages provide the ability to store data in a variable, or a collection of data in an array or list. In *object-oriented programming*, an *object* is another type of structure that is used to store information about data.

Objects, once constructed, usually store information in the form of *attributes*, and allow one to interact with that information using *methods*. Methods can be categorized as *accessor methods* ("getters"), which *get* information about the object, and *mutator methods* ("setters"), which *set*, or alter the state of the object.

Almost all objects, even simple ones, will include:
- a constructor to establish the initial state of the attributes
- accessor methods to retrieve the state of those attributes
- mutator methods to change the state of those attributes

Example: Creating a Bug class

```
/**
 * The Bug class manages a bug
 */
public class Bug
{
    // instance variables
    private String name;

    ////////////////////////////////////

    /**
     * Constructs a new bug
     * @param name The name of the bug
     */
    public Bug(String name)
    {
        this.name = name;
    }
```

```
        ////////////////////////////////////

        /**
         * The getName method identifies the name of the bug
         * @return the name of the bug
         */
        public String getName()
        {
            return name;
        }

        ////////////////////////////////////

        /**
         * The changeName method renames the bug
         * @param newName the new name of the bug
         */
        public void changeName(String newName)
        {
            name = newName;
        }
    }
```

5. **Describing the Position of an Object**
   Objects that move about in space usually need attributes to keep track of their location. `Stars` have a location in the sky, `Cars` have a location on the road, and `Pawns` have a location on a chessboard.

   For such objects, there is usually at least one mutator method that causes the location attributes to be updated.

```
    // instance variables
    private int xLoc = 0;
    private int yLoc = 0;
    .
    .

    /**
     * Changes the location of the chesspiece on the chessboard
     */
    public void move(int rowDelta, int columnDelta)
    {
        y = y + rowDelta;
        x = x + columnDelta;
    }
```

   In some cases, location data may be updated based on existing motion values and time:

```
    /**
     * Changes the horizontal and vertical positions of a projectile
     */
    public void move(int deltaTime)
    {
        // xf = xi + vt                          Physics!
        x = x + velX * deltaTime;

        // yf = yi + v(i)t + (1/2) a t^2         Physics!
        y = y + velY * deltaTime + (1/2) * accelY * deltaTime * deltaTime;

        // Update velY for the next iteration   More physics!
        velY = velY + accelY * deltaTime;
    }
```

6. **Objects with Distinct States**

An object's attributes are used to describe its state—`xLoc` and `yLoc` above, for example. Some attributes have a limited number of states, however. A simple two-state attribute might be managed with a `boolean` variable:

```
// instance variables for BankAccount
private double balance;
private boolean accountActive = true;
.
.

/**
 * Widthdraw money from active account
 * @return 0 if successful, -1 if failed
 */
public int withdraw(double amount)
{
    if (accountActive)
    {
        balance -= amount;
        return 0;
    }
    else return -1;
}
```

For multiple states, final variables can be set to an `int` value, and that value/variable used in the program.

```
// instance variables for Frog class
private int age;

public static final int EGG = 0;
public static final int TADPOLE = 1;
public static final int JUVENILE = 2;
public static final int ADULT = 3;
public static final int DEAD = 4;

/**
 * Advance the frog's age
 */
public void grow()
{
    if (age < DEAD)   { age++; }
}
```

Multiple states can also be identified using an *enumerator*, which creates a type with finite set of specific state values.

```
public class Person
{
    // Enumeration type for Person class
    public enum Stage { INFANT, CHILD, ADOLESCENT, ADULT };

    private Stage stage;

    public Person()
    {
        stage = Stage.INFANT;
    }

    public String getAgeRange()
    {
        if (stage == Stage.INFANT) { return "0 - 18 months"; }
        else if (stage == Stage.CHILD) { return "18 months - 12 years"; }
        else if (stage == Stage.ADOLESCENT) {return "13 years - 17 years"; }
        else return "18 years and up";
    }
}
```