The goal of ADT design and implementation is to build a wall between the ADT and those program parts that use the ADT, which are called its clients. The ADT’s interface provides the gateway through which the clients and the ADT can interact. Contact through any means other than the interface is forbidden.

Effective data abstraction and encapsulation require more effort than simply creating a class with private fields.

**Steps of ADT Design and Implementation**

1. Study the properties of the data abstraction and develop a list of operations that it should provide.
2. For each operation in the list: (a) Derive the client/object interaction that it requires and (b) write a method specification that supports this interaction; use comments to express any properties of the operation that you cannot represent in code.
3. Code each method so that the properties of the data abstraction are maintained.
4. Test the interface and implementation of each method.

If a team of more than one programmer is developing the program then steps 1 and 2 must be done as a group with consensus on the interface design. Once the interface is designed, each programmer can complete his or her separate part of the program (i.e. client vs. ADT) without interfering with the others, so long as the wall of separation is maintained.
The Fundamental Principle of ADT Design and Implementation

*The ADT’s interface must be general enough to encourage its use and restrictive enough to protect the integrity of its data abstraction.*

The key to satisfying the encouragement clause is to make the ADT’s interface easy to use and versatile enough to be widely applicable.

**Example**

The Java API class `javax.swing.BoxLayout` violates the fundamental principle of ADT design because it is not easy to use.

The data abstraction is simple enough. In Java, a layout manager controls how GUI components are organized within a window. The `BoxLayout` manager organizes components either vertically or horizontally with no wrap.

Using the class is not so simple, as illustrated in the code below, which contrasts the code needed to use the `BoxLayout` manager versus the `FlowLayout` manager.

**Using Flow Layout**

```java
JFrame win = new JFrame();
win.setLayout( new FlowLayout() );
```

**Using Box Layout**

```java
JFrame win = new JFrame();
win.getContentPane().setLayout( new BoxLayout( box, BoxLayout.X_AXIS ) );
```
Example
The Java API class `java.util.Date` violates the fundamental principle of ADT design because it is not versatile. Using this class, a specific instant of time can be interpreted as a year, month, day, hour, minute and second within the *Gregorian calendar* but not in any of the world’s other calendars. Thus, the class is not appropriate for international use.

One key to satisfying the integrity clause is to carefully select the object’s *mutators*, which are the public instance methods that can change the object’s state.

Example
Class *Fraction* provides a data abstraction for a *rational number*, which is the quotient of two integers where the divisor is not zero. In other words, a fraction \( \frac{a}{b} \) where \( a \) and \( b \) are both integers and \( b \neq 0 \).

```java
public class Fraction
{
    private int numerator;
    private int denominator;
    . . .
}
```

Here are three possible public set methods, any or all of which could be included in the class’s interface:

```java
public void setNumerator( int n )
    // Set this fraction's numerator to n.

public void setDenominator( int d )
    // Set this fraction's numerator to d.

public void setFraction( int n, int d )
    // Set this fraction to n / d.
```

One could argue that *setFraction* is the best choice because it treats the numerator and denominator as a collective whole. Thus, it is more consistent with the data abstraction that the *Fraction* class is trying to create.
The most crucial requirement for satisfying integrity clause is to implement the public mutators so that they cannot leave an object in an inconsistent state or violate its data abstraction.

**Example**

Two possible implementations of a `setFraction` method for the `Fraction` class (see the previous example) are shown below. The one on the left allows programmers to create a fraction having a zero denominator:

```java
Fraction f;
f.setFraction( 1, 0 );
```

The `setFraction` method shown on the right prevents this by throwing an exception if the second argument is zero.

```java
public class Fraction {
    private int numerator;
    private int denominator;

    public void setFraction(int n, int d) {
        numerator = n;
        denominator = d;
    }
}
```

```java
public class Fraction {
    private int numerator;
    private int denominator;

    public void setFraction(int n, int d) throws IllegalArgumentException {
        if (d == 0)
            throw new IllegalArgumentException();
        else {
            numerator = n;
            denominator = d;
        }
    }
}
```

In so far as he or she maintains the integrity of the data abstraction, the programmer writing the code for the ADT has free rein to do whatever. There is good advice, however, on how to save time and effort on the implementation.

**The “Don’t Reinvent the ADT Wheel” Implementation Rule**

Implement a core set of methods that directly manipulate the private fields of the abstract data type. Use these core methods to implement the other, more complex, methods.  

---

Example
Once you’ve implemented the `setFraction` method of class `Fraction`, you can implement the class constructor by simply calling it (see line 9).

```java
public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction( int n, int d )
        throws IllegalArgumentException
    // Fraction constructor.
    {
        setFraction( n, d );
    }

    public void setFraction( int n, int d )
        throws IllegalArgumentException
    // Set this fraction to n / d.
    {
        if ( d == 0 )
            throw new IllegalArgumentException( );
        else
        {
            numerator = n;
            denominator = d;
        }
    }
}
```