Traditionally, visitors are used to implement type-testing without sacrificing type-safety, so long as your types are well-defined up front and known in advance. Let's say we have a few classes as follows:

abstract class Fruit { }

class Orange : Fruit { }

class Apple : Fruit { }

class Banana : Fruit { }

And let's say we create a Fruit[]:

var fruits = new Fruit[]

{ new Orange(), new Apple(), new Banana(),

new Banana(), new Banana(), new Orange() };

I want to partition the list in to three lists, each containing oranges, apples, or bananas. How would you do it? Well, the *easy* solution would be a type-test:

List<Orange> oranges = new List<Orange>();

List<Apple> apples = new List<Apple>();

List<Banana> bananas = new List<Banana>();

foreach (Fruit fruit in fruits)

{

if (fruit is Orange)

oranges.Add((Orange)fruit);

else if (fruit is Apple)

apples.Add((Apple)fruit);

else if (fruit is Banana)

bananas.Add((Banana)fruit);

}

It works, but there are lots of problems with this code:

* For a start, its ugly.
* Its not type-safe, we won't catch type errors until runtime.
* Its not maintainable. If we add a new derived instance of Fruit, we need to do a global search for every place which performs a fruit type-test, otherwise we might miss types.

Visitor pattern solves the problem elegantly. Start by modifying our base Fruit class:

interface IFruitVisitor

{

void Visit(Orange fruit);

void Visit(Apple fruit);

void Visit(Banana fruit);

}

abstract class Fruit { public abstract void Accept(IFruitVisitor visitor); }

class Orange : Fruit { public override void Accept(IFruitVisitor visitor) { visitor.Visit(this); } }

class Apple : Fruit { public override void Accept(IFruitVisitor visitor) { visitor.Visit(this); } }

class Banana : Fruit { public override void Accept(IFruitVisitor visitor) { visitor.Visit(this); } }

It looks like we're copy pasting code, but note the derived classes are all calling different overloads (the Apple calls Visit(Apple), the Banana calls Visit(Banana), and so on).

Implement the visitor:

class FruitPartitioner : IFruitVisitor

{

public List<Orange> Oranges { get; private set; }

public List<Apple> Apples { get; private set; }

public List<Banana> Bananas { get; private set; }

public FruitPartitioner()

{

Oranges = new List<Orange>();

Apples = new List<Apple>();

Bananas = new List<Banana>();

}

public void Visit(Orange fruit) { Oranges.Add(fruit); }

public void Visit(Apple fruit) { Apples.Add(fruit); }

public void Visit(Banana fruit) { Bananas.Add(fruit); }

}

Now you can partition your fruits without a type-test:

FruitPartitioner partitioner = new FruitPartitioner();

foreach (Fruit fruit in fruits)

{

fruit.Accept(partitioner);

}

Console.WriteLine("Oranges.Count: {0}", partitioner.Oranges.Count);

Console.WriteLine("Apples.Count: {0}", partitioner.Apples.Count);

Console.WriteLine("Bananas.Count: {0}", partitioner.Bananas.Count);

This has the advantages of:

* Being relatively clean, easy to read code.
* Type-safety, type errors are caught at compile time.
* Maintainability. If I add or remove a concrete Fruit class, I could modify my IFruitVisitor interface to handle the type accordingly, and the compiler will immediately find all places where we implement the interface so we can make the appropriate modifications.

With that said, visitors are usually overkill, and they have a tendency to grossly complicate APIs, and it can be very cumbersome to define a new visitor for every new kind of behavior.

Usually, simpler patterns like inheritance should be used in place of visitors. For example, in principle I could write a class like:

class FruitPricer : IFruitVisitor

{

public double Price { get; private set; }

public void Visit(Orange fruit) { Price = 0.69; }

public void Visit(Apple fruit) { Price = 0.89; }

public void Visit(Banana fruit) { Price = 1.11; }

}

It works, but what's the advantage over this trivial modification:

abstract class Fruit

{

public abstract void Accept(IFruitVisitor visitor);

public abstract double Price { get; }

}

So, you should use visitors when the following conditions hold:

* You have a well-defined, known set of classes which will be visited.
* Operations on said classes are not well-defined or known in advance. For example, if someone is consuming your API and you want to give consumers a way to add new ad-hoc functionality to objects. They're also a convenient way to extend sealed classes with ad-hoc functionaity.
* You perform operations of a class of objects and want to avoid run-time type testing. This is usually the case when you traverse a hierarchy of disparate objects having different properties.