- **Wizytator** – jego zadaniem jest "odwiedzenie" każdego elementu w danym kontenerze i wykonanie na nim konkretnych działań.

- Różne implementacje wizytatorów mogą wykonywać różne zadania

- Wizytatorzy są tworzeni na zewnątrz kontenera i następnie są przekazywani do kontenera, aby ten oprowadził ich po wszystkich wartościach w nim przechowywanych

**Klasa bazowa Wizytatora:**

```
template <typename T>
class Visitor
{
public:
    virtual void Visit (T & element) = 0;
    virtual bool IsDone () const
};
```

## Przeglądanie zawartości – wizytatorzy

```
class WizytatorDrukujacyInty : public Visitor<int>
{
public:
    virtual void Visit(int & value)
    {
        cout << value << endl;
    }
};

JakiśKontener<int> k;
WizytatorDrukujacyInty v;
k.Accept(v);
```

**Interakcja pomiędzy kontenerem a wizytorem:**

- wywołanie funkcji Accept. - kontener „akceptuje ” wizytora
- Kontener wywołuję metodę Visit przekazanego przez referencje wizytora dla każdego obiektu zawartego w kontenerze.

Przykład:

```
void SomeContainer::Accept (Visitor& visitor) const    for each
Object i in this container
                visitor.Visit (i);
```

- Funkcja Accept wywołuje funkcję Visit dla każdego obiektu *i* w kontenerze.
- Ponieważ klasa Visitor jest abstrakcyjną klasą bazową i nie dostarcza implementacji operacji Visit, od typu wizytora zależeć będzie, co będzie robić funkcja Visit.

## Przedterminowe kończenie wizytacji

Istnieją sytuacje, w których niepotrzebne jest wizytowanie wszystkich elementów. Wizytator sprawdzający, czy w kontenerze jest jakaś liczba nieparzysta, mógłby zakończyć pracę po znalezieniu pierwszej takiej liczby.

Dodamy wizytatorowi metodę `IsDone`. Kontener kontynuuje oprowadzanie do momentu aż `IsDone` zwróci `true`, albo aż wyczerpią się elementy.

```
template <typename T>
void Stack<T>::Accept(Visitor<T> & v)
{
    for (int i = top_idx; i >= 0 && ! v.IsDone(); --i)
        v.Visit(data[i]);
}
```

Domyślnie metoda IsDone zwraca false.

**Zad.1. Stworzyć klasę AddingVisitor – wizytatora, który oblicza sumę elementów zbioru.**

```
template <typename T>
class Visitor
{
public:
    virtual void Visit (T& element) = 0;
    virtual bool IsDone () const{ return false; }
};
```

**1.1.**

Tworzymy klasę AddingVisitor dziedziczącą z klasy Visitor. W klasie mamy pole(np. sum) w którym pamiętamy policzona sumę. Metoda AddingVisitor.visit(i) zwieksza wartość tego pola sum o **i**.

**1.2.** Implementujemy metodę Accept dla klasy SetAsArray (przegląda wszystkie elementy zbioru wywołując v.visit().(„skacze” po komórkach tablicy w których jest wartość true)

Zmiany w klasie Container:

```
template <typename T>
class Container
{
    Container (){};
 public:
    virtual int Count () const = 0;
    virtual bool IsEmpty () const {return Count()==0;};
    virtual bool IsFull () const = 0;
    virtual void MakeNull() = 0;
    virtual void Accept (Visitor<T> & v)=0;
 };
```

**Zad.2.**
**Proszę rozszerzyć test z poprzednich zajęć:**
**Stara część testu:**
Utwórz zbiory A,B,C i D (zbiór uniwersalny rozmiar 10)
Do zbioru A wstaw elementy parzyste
Do zbioru B wstaw elementy **nie**parzyste
C=A+B
D=C-B
A.Wypisz();
B.Wypisz();
C.Wypisz();
D.Wypisz();
sprawdź, czy:
"D==A"
"D<=A"
"C==B"
"B<=C"
Do zbioru A wstaw elementy wartość 1
sprawdź, czy:
"D==A"
"D<=A"
**Nowa część testu:**
Do zbioru A wstaw elementy wartość 5
Utwórz wizytatora dla zbioru A: v_A
A.Accept(v_A)
Wypisz sumę policzoną przez v_A
E=A *B
Utwórz wizytatora dla zbioru E: v_E
E.Accept(v_E)
Wypisz sumę policzoną przez v_E
Usuń wartość 1 z E i ponownie oblicz sumę elementów zbioru E korzystając z wizytatora