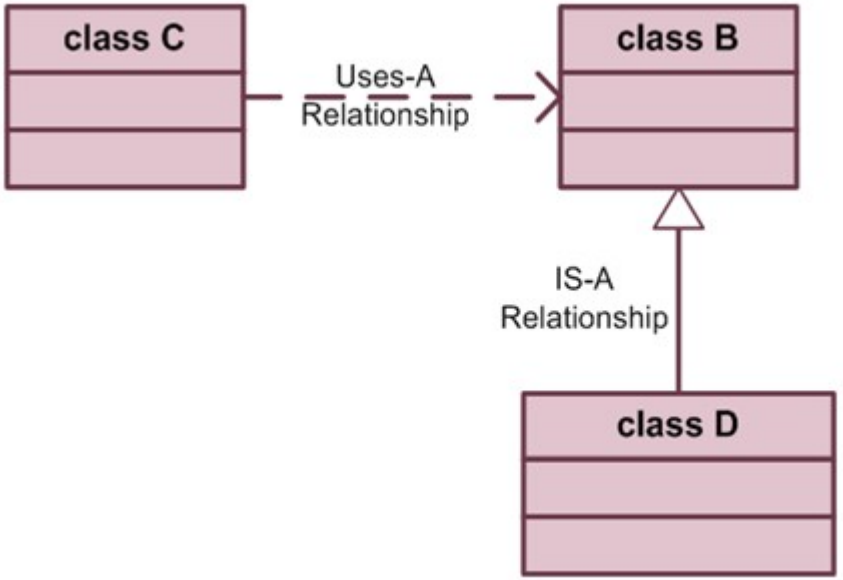
КРОСС-ПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

ПРОЕКТИРОВАНИЕ ПО КОНТРАКТУ

ПРОЕКТИРОВАНИЕ ПО КОНТРАКТУ

Наследование



Использование утверждений совместно с наследованием упрощает создание производных классов и гарантирует клиентам базового класса согласованное поведение при использовании динамического связывания и полиморфизма.

Правило родительских инвариантов: инварианты всех родителей применимы и к самому классу

Предусловия и постусловия

Полиморфизм и динамическое связывание добавляет некоторые особенности при работе с предусловиями и постусловиями.

```
class B {
  public virtual int Foo(int x) {
      Contract.Requires(x > 5, "x > 5");
      Contract.Ensures(Contract.Result<int>() > 0,
                       "Result > 0");
      // Реализация метода
  }
}
```

Использование объекта B внутри класса C:

```
class C {
  //...
  B b = GetFromSomewhere();
  int x = GetX();
  if (x > 5) { //Проверяем предусловие pre_b
    int result = b.Foo(x);
    Contract.Assert(result > 0); // Проверяем постусловие
  post_b
  }
}
```

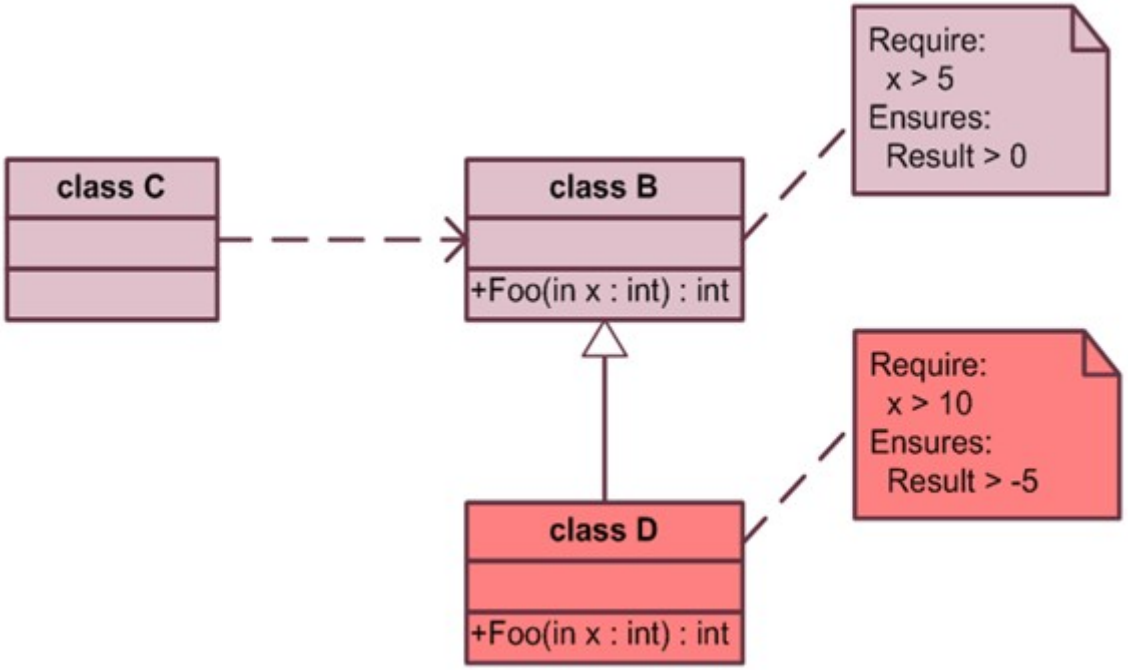
Предусловия и постусловия

Предположим, что при переопределении функции Foo один из наследников класса B захочет изменить предусловия и постусловия:

```
class B {
 public virtual int Foo(int x) {
   Contract.Requires(x > 5, "x > 5");
   Contract.Ensures(Contract.Result<int>() > 0,
           "Result > 0");
   // Реализация метода
 }
}
```

Предусловия и постусловия

Функция int Foo(int x) в классе D начинает требовать больше (содержит более сильное предусловие вида: x > 10), и гарантировать меньше (содержит более слабое постусловие вида: x > -5):



```
class D : B {
    public override int Foo(int x) {
        Contract.Requires(x > 10, "x > 10");
        Contract.Ensures(Contract.Result<int>() > -5,
                         "Result > -5");
        return -1;
    }
}
```

Хотя клиент класса B полностью выполняет свою часть контракта и предоставляет входное значение функции Foo, удовлетворяющее предусловию он может не получить требуемого результата.

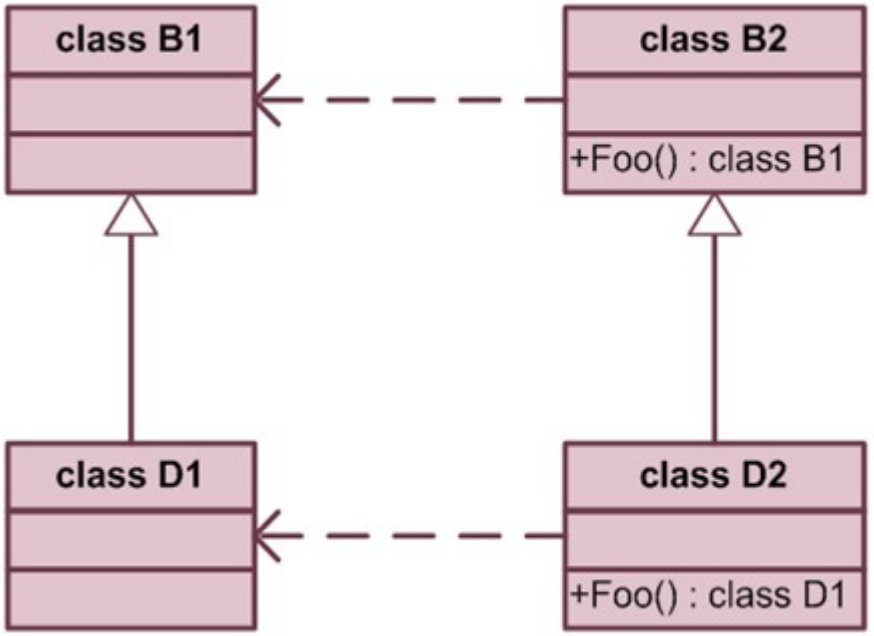
Вывод: при переопределении методов предусловие может заменяться лишь равным ему или более слабым (требовать меньше), а постусловие – лишь равным ему или более сильным:

```
class D : B {
    public override int Foo(int x) {
        Contract.Requires(x > 0, "x > 0");
        Contract.Ensures(Contract.Result<int>() > 10,
                         "Result > 10");
        return 25;
    }
}
```

Ковариантность и контрвариантность

Ковариантность по типу возвращаемого значения

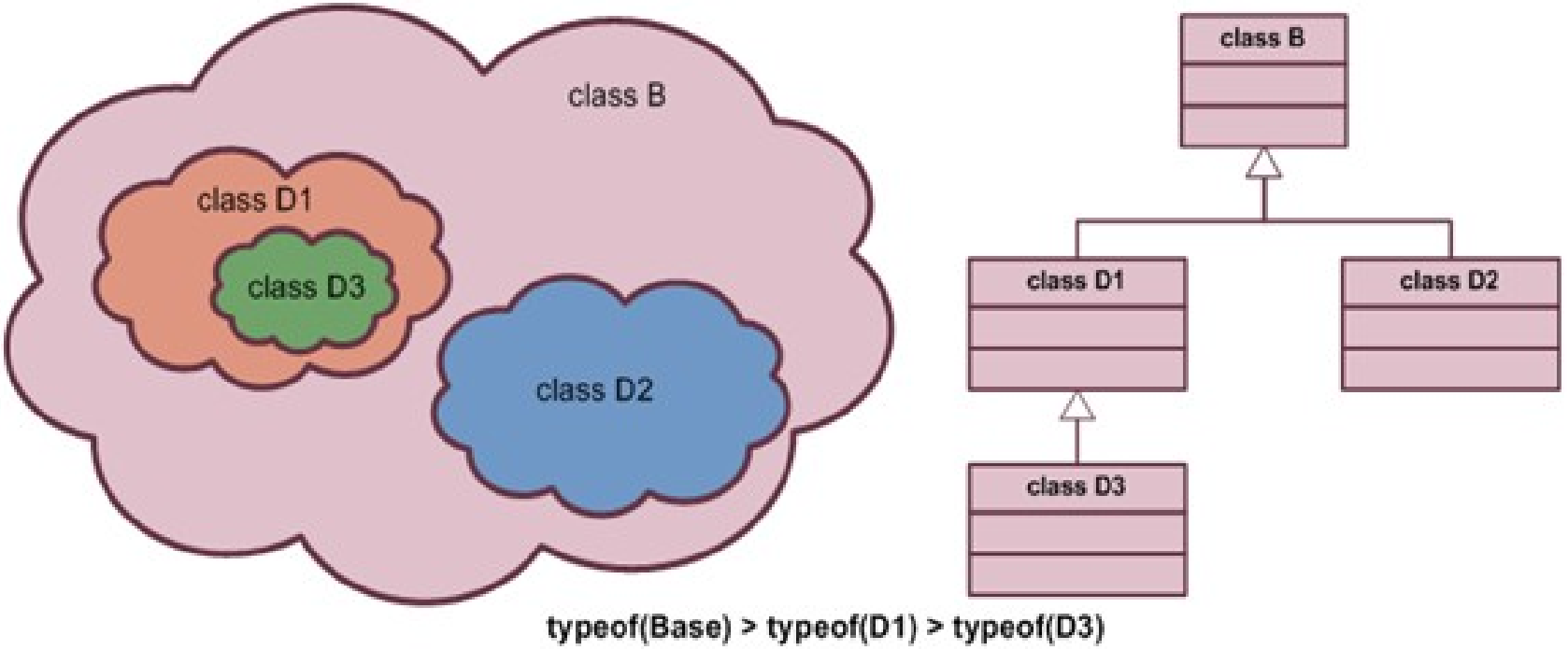
```
class B1 {};
class D1 : public B1 {};
class B2 {
public:
  virtual B1 Foo();
};
class D2 : public B2 {
public:
  virtual D1 Foo();
};
```



Можно считать, что для типов D1 и B1 выполняется соотношение:

`typeof(D1) < typeof(B1).`

Ковариантность и контрвариантность



Ковариантность и контрвариантность

Примером может служить ковариантность по типу возвращаемого значения и контрвариантность по типу принимаемого значения обобщенных интерфейсов и делегатов.

Делегату d1 с предусловием pre1 и постусловием post1 может быть присвоен делегат d2 с предусловием pre2, равном pre1 или более слабым, и постусловием post2, равным post1 или более сильным:

```
void Foo(object obj) {}
string Boo() {return “Boo”;}

// Контравариантность аргументов:
// предусловие делегата Action<object> и, соответственно
// метода Foo, слабее предусловия делегата
// Action<string>, поскольку typeof(object) < typeof(string)
Action<string> action = Foo;
// что аналогично следующему коду:
Action<string> action = new Action<object>(Foo);

// Ковариантность возвращаемого значения:
// постусловие делегата Func<string> и, соответственно
// метода Boo, сильнее постусловия делегата
// Func<object>, поскольку typeof(string) > typeof(object)
Func<object> func = Boo;
// что аналогично следующему коду:
Func<object> func = new Func<string>(Boo);
```