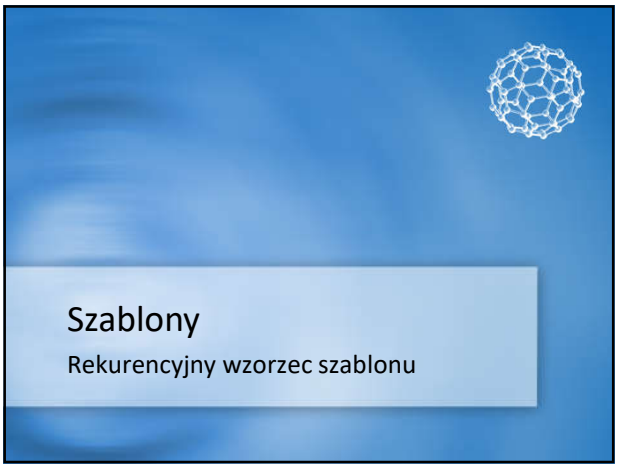


257

Szablony – rozszerzenie

Rekurencyjny wzorzec szablonu – przykład: licznik

reprezentuje idiom służący do definiowania dowolnych klas zawierających licznik utworzonych instancji tych klas

Pokazane na wcześniejszych wykładach podstawowe rozwiązanie polega na tym, że do klasy dodajemy statyczne pole które będzie reprezentować licznik. Przy każdym wywołaniu konstruktora licznik ten jest inkrementowany a przy wywołaniu destruktora – dekrementowany

Jest to rozwiązanie skuteczne ale siermiężne: tworząc każdą kolejną nową klasę należy pamiętać o doimplementowaniu obsługi tej składowej statycznej i jeszcze się przy tym nie wolno pomylić

Korzystniej byłoby stworzyć klasę bazową zawierającą taki licznik oraz odpowiedni konstruktor i destruktor, a następnie tworząc nowe klasy zawsze deklarować, że dziedziczą po tej jednej, bazowej, np. ..

258

Szablony – rozszerzenie

```
class Policzalny {
   static int count;
public:
   Policzalny() { ++count; }
   Policzalny(const Policzalny&) { ++count; }
   ~Policzalny() { --count; }
   static int getCount() { return count; }
};
int Policzalny::count = 0;
class PoliczalnyC1 : public Policzalny {};
class PoliczalnyC2 : public Policzalny {};

int main() {
   PoliczalnyC1 a;
   cout << PoliczalnyC1::getCount() << endl; // 1
   PoliczalnyC1 b;
   cout << PoliczalnyC1::getCount() << endl; // 2
   PoliczalnyC2 c;
   cout << PoliczalnyC2::getCount() << endl; // 3 (Błąd! – wcale nie 3..)
}
```


259

Szablony – rozszerzenie

Rekurencyjny wzorzec szablonu

Dziedziczenie po wspólnej klasie bazowej jest błędne.

Potrzebna jest raczej metoda automatycznego generowania różnych klas bazowych dla każdej klasy pochodnej.

Służy do tego konstrukcja szablonu pokazana na następnym slajdzie.

260

Szablony – rozszerzenie

```
template<typename T>
class Policzalny {
   static int count;
public:
   Policzalny() { ++count; }
   Policzalny(const Policzalny<T>&) { ++count; }
   ~Policzalny() { --count; }
   static int getCount() { return count; }
};

template<typename T>
int Policzalny<T>::count = 0;
```

Do czego przyda się parametr T?..

261

Szablony – rozszerzenie

```
template<typename T>
class Policzalny;

template<typename T>
int Policzalny<T>::count = 0;

class PoliczalnyC1 : public Policzalny<PoliczalnyC1> {};    //(!)
class PoliczalnyC2 : public Policzalny<PoliczalnyC2> {};    //(!)

int main() {
   PoliczalnyC1 a;
   cout << PoliczalnyC1::getCount() << endl; // 1
   PoliczalnyC1 b;
   cout << PoliczalnyC1::getCount() << endl; // 2
   PoliczalnyC2 c;
   cout << PoliczalnyC2::getCount() << endl; // 1 (!)
}
```


262

Szablony – rozszerzenie

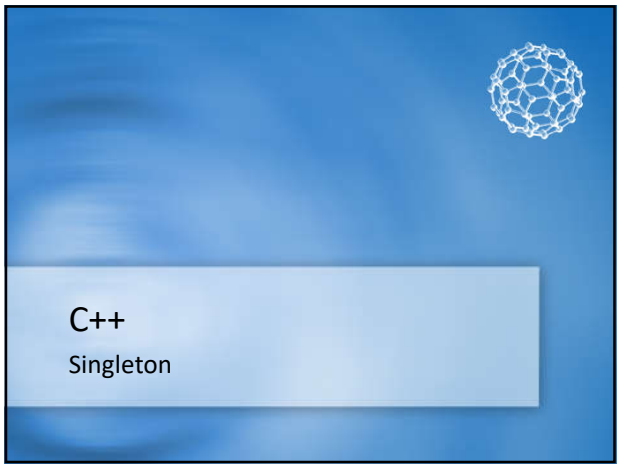
Rekurencyjny wzorzec szablonu

Wydawać się może, że jest to definicja cykliczna …

> byłaby taką, gdyby jakiekolwiek pole klasy bazowej użyło w obliczeniach argumentu szablonu. Jednak żadne pola **`Policzalny`** nie zależą od T.
>
> Wielkość **`Policzalny::count`** wynosząca zero jest znana już podczas analizowania szablonu. Wobec tego nie ma znaczenia, który argument zostanie użyty do ukonkretnienia **`Policzalny`**, bo jej wielkość będzie zawsze taka sama.

Wszelkie dziedziczenie po **`Policzalny`** może mieć miejsce zaraz po jej analizie, ponieważ nie występuje w ogóle rekurencja.

Każda klasa bazowa jest niepowtarzalna, dlatego ma własne dane statyczne.



Singleton

Wprowadzenie

W programach C++ wielokroć powtarza się potrzeba utworzenia obiektu, który będzie istniał przez cały czas funkcjonowania programu w tylko jednym egzemplarzu.

Dlatego przyjęło się powszechnie jedno rozwiązanie tego problemu, stanowiące rodzaj wzorca, wg którego należy taką potrzebę realizować – tzw. Singleton.

Kluczem do utworzenia prawidłowego Singletona jest zablokowanie przejęcia kontroli nad czasem życia obiektu przez programistę. W tym celu należy zadeklarować wszystkie konstruktory klasy jako prywatne i uniemożliwić kompilatorowi niejawne generowanie konstruktorów domyślnych.

Singleton

```
class Singleton {
  static Singleton s;
  int i;
  Singleton(int x=0) : i(x) { }            // konstr. domyślny - prywatny
  Singleton(const Singleton&);             // konstr. kopiujący - prywatny
  Singleton& operator=(Singleton&);        // operator przypisania - prywatny
public:
  static Singleton& instance() { return s; }
  int getValue() { return i; }
  void setValue(int x) { i = x; }
};
Singleton Singleton::s(47);                // inicjalizacja składowej statycznej
int main() {
  Singleton& s = Singleton::instance();
  cout << s.getValue() << endl;
  Singleton& s2 = Singleton::instance();
  s2.setValue(9);
  cout << s.getValue() << endl;
}
```


Singleton

- Prywatne konstruktory i operator przypisania – to zapobiega korzystaniu z nich oraz generowaniu przez kompilator domyślnych wersji tych mechanizmów.
- Jeden obiekt (singleton) tworzony jest statycznie w momencie uruchomienia programu – pole:

```
static Singleton s;
```

- Inna wersja – tworzy obiekt na żądanie, tj. w momencie, kiedy programista po raz pierwszy odwoła się do obiektu.
- Taka inicjalizacja ma sens, kiedy jest ona kosztowna, a może okazać się niekonieczną, jeżeli akurat w danym uruchomieniu programu odwołanie się do tego Singletona nie było wymagane.

Singleton

```
class GlobalClass {
   int    m_value;
   static GlobalClass* s_instance;
   GlobalClass( int v=0 ) {
     m_value = v;
   }
public:
   int  get_value(){ return m_value; }
   void set_value( int v ) {
     m_value = v;
   }
   static GlobalClass* instance() {
      if ( ! s_instance )
         s_instance = new GlobalClass;
      return s_instance;
   }
};

GlobalClass* GlobalClass::s_instance = 0;

void foo( void ) {
   GlobalClass::instance()->set_value(1);
   cout << "foo: global_ptr is "
   << GlobalClass::instance()->get_value()
   << '\n';
}

int main( void ) {
   cout << "main: global_ptr is "
   << GlobalClass::instance()->get_value()
   << '\n';
   foo();
}
```

Singleton

Podsumowanie:

- Singleton to obiekt, który istnieje w tylko jednym egzemplarzu dla danej klasy.
- Singleton przypomina zmienną globalną, przewaga Singletona polega na tym, że jesteśmy absolutnie pewni liczby kopii danego typu.
- Nadużywanie Singletonów w miejsce zmiennych globalnych jest błędem.
- Używanie zmiennych globalnych to zły styl programowania; powinny być używane tylko wtedy, kiedy jest to absolutnie konieczne.
- Zamienianie zmiennych globalnych na Singletony, w celu zmniejszenia liczby zmiennych globalnych to kompletna pomyłka: jeden rodzaj obiektów globalnych jest po prostu zamieniany na inny (*zamiast tego należy tak zmienić sposób przekazywania informacji w programie, aby jakiekolwiek obiekty globalne były zbędne, lub konieczne tylko w minimalnej ilości*).
- Dyskusja w sieci: czy singleton to *antywzorzec*?

269

270

Wielodziedziczenie

Wprowadzenie

- Wielodziedziczenie – wskazanie kilku klas jako bazowych dla klasy pochodnej.
- Wielodziedziczenie wiąże się z problemami związanymi z:
  - duplikowaniem się składowych, ponieważ w wielodziedziczeniu nie ma możliwości ograniczenia zbioru dziedziczonych składowych.
  - Położeniem fizycznych obiektów w pamięci (to problem dla kompilatora, nie programisty)
- Klasyczne rozwiązania korzystające w z wielodziedziczenia:
  - klasy interfejsu
  - Klasy domieszek

Poprawność implementacji obydwu paradygmatów programowania nie jest gwarantowana przez kompilator, a jedynie przez autora kodu

271

Wielodziedziczenie

Klasy interfejsu

- Dziedziczenie samego interfejsu symuluje się poprzez dziedziczenie po *klasie interfejsu*, która to klasa składa się wyłącznie z deklaracji i jest pozbawiona pól i kodu ciał metod. Deklaracje takie są metodami czysto wirtualnymi.

272

Wielodziedziczenie

```
class Printable {
public:
  virtual ~Printable() {}
  virtual void print(ostream&) const=0;
};
class Intable {
public:
  virtual ~Intable() {}
  virtual int toInt() const=0;
};
class Stringable {
public:
  virtual ~Stringable() {}
  virtual string toString() const=0;
};
```

```
class Able : public Printable,
             public Intable,
             public Stringable {
  int myData;
public:
  Able(int x) { myData = x; }
  void print(ostream& os) const {
    os << myData;
  };
  int toInt() const {
    return myData;
  };
  string toString() const {
    ostringstream os;
    os << myData;
    return os.str();
  };
};
```


273

Wielodziedziczenie

Klasy domieszek

Dziedziczenie implementacji – jest korzystne, bo eliminuje potrzebę implementowania od nowa całego zachowania klasy bazowej w klasie pochodnej

Do tego celu powstał drugi paradygmat – obok klas interfejsu – wykorzystywany w wielodziedziczeniu: *klasy domieszek* (*mixin classes*).

Są to klasy projektowane jako nieprzeznaczone do samodzielnej instancjalizacji, a wykorzystywane jedynie w celu dodania nowej funkcjonalności do innych klas poprzez dziedziczenie. Mają prywatny konstruktor, aby zapobiec tworzeniu ewentualnej pomyłkowej instancji.

274

Wielodziedziczenie

Klasy domieszek

Projektując strukturę klas, jednej z klas nadajemy rolę klasy głównej bazowej, a pozostałe pełnią role domieszek. Można je dołączać na liście klas bazowych i w ten sposób ubogacać właściwości klasy dziedziczącej

Problem:

Nie zawsze istnieje możliwość zrównoleglenia funkcjonalności domieszek, tak aby można było je łatwo dodawać lub nie. Czasem w działanie kodu-domieszki musi być wbudowane działanie kodu głównego. Rozwiązanie tego problemu w przykładzie podanym na następnych slajdach.

275

Wielodziedziczenie

Przykład:

Przyjmijmy, że mamy do napisania moduł Menedżera Zadań zarządzającego zadaniami do asynchronicznego wykonania.

Interfejs klasy reprezentującej zadanie:

```
struct ITask
{
  virtual std::string GetName() = 0;
  virtual void Execute() = 0;      // główna metoda realizująca zadanie
};
```

Zakładamy, że ten interfejs został zaimplementowany przez wszystkie zadania, jakie będą wykonywane przez Menedżera.

Przykład pochodzi z:

Daniel Paull: C++ Mixins - Reuse through inheritance is good... when done the right way

http://www.thinkbottomup.com.au/site/blog/C%20%20_Mixins_-_Reuse_through_inheritance_is_good

276

Wielodziedziczenie

Przykład:

Postawiono też wymaganie, aby następujące dodatkowe elementy funkcjonalności, były wspólne dla wszystkich możliwych implementacji klas reprezentujących zadanie:

1. Określenie i wypisanie czasu wykonania pojedynczego zadania,
2. zapisywanie w logu komunikatów o uruchomieniu oraz o zakończeniu pojedynczego zadania.

277

Wielodziedziczenie

Przykład:

Podsumowując, na wstępie mamy zdefiniowane elementy:

1. Menedżer zadań (opis funkcjonalny),
2. Interfejs klasy reprezentującej zadanie – **`struct Itask`**
3. Wymagane dodatkowe elementy funkcjonalności wspólne dla implementacji klas reprezentujących różne zadania.

Cel: Należy zaimplementować klasy, w których będą implementowane właściwe czynności realizowane przez zadania, oraz klasy domieszek wprowadzające do zadań wymagane dodatkowe elementy funkcjonalności.

W pierwszym podejściu dodatkowe funkcjonalności domieszek zostaną zaimplementowane w klasie bazowej dla zadania reprezentującego właściwe zadanie i dziedziczącej po **`struct ITask`**.

278

Wielodziedziczenie

Podejście #1:

Zapisywanie w logu - klasa bazowa abstrakcyjna

implementuje metodę **`Execute`** oraz deklaruje metodę czysto wirtualną **`OnExecute`**,

1. która musi być implementowana w klasach pochodnych,
2. w której będzie implementowana właściwa czynność realizowana przez zadanie

```
class BaseLoggingTask : public ITask {
public:
 virtual void Execute() {
  std::cout << "LOG: The task is starting: " << GetName().c_str() << std::endl;
  OnExecute();
  std::cout << "LOG: The task has completed: " << GetName().c_str() << std::endl;
 }
 virtual void OnExecute() = 0;
};
```


279

Wielodziedziczenie

Podejście #1:

Korzystanie z abstrakcyjnej klasy bazowej – klasa pochodna

Klasa implementująca właściwą czynność realizowaną przez to zadanie:

```
class MyTask1 : public BaseLoggingTask
{
public:
 virtual void OnExecute() {
  std::cout << "...This is where the task is executed..." << std::endl;
 }
 virtual std::string GetName() {
  return "My task name";
 }
};
```

Na pierwszy rzut oka wygląda OK, kod jest spójny i łatwy do czytania. Podobnie możemy też zaimplementować klasę do określania czasu wykonania zadania..

280

Wielodziedziczenie

Podejście #1:

Zapisywanie czasu wykonania - klasa bazowa abstrakcyjna

implementuje metodę **`Execute`** oraz deklaruje metodę czysto wirtualną **`OnExecute`**

```
class BaseTimingTask : public ITask {
 Timer timer_;
public:
 virtual void Execute() {
  timer_.Reset();
  OnExecute();
  double t = timer_.GetElapsedTimeSecs();
  std::cout << "Task Duration: " << t << " seconds" << std::endl;
 }
 virtual void OnExecute() = 0;
};
```


281

Wielodziedziczenie

Podejście #1:

Korzystanie z abstrakcyjnej klasy bazowej – klasa pochodna

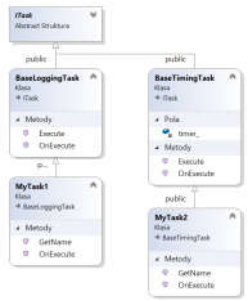
Klasa implementująca właściwą czynność realizowaną przez to zadanie:

```
class MyTask2 : public BaseTimingTask
{
public:
 virtual void OnExecute() {
  std::cout << "...This is where the task is executed..." << std::endl;
 }
 virtual std::string GetName() {
  return "My task name";
 }
};
```


282

Wielodziedziczenie

Podejście #1:



283

Wielodziedziczenie

No, ale..

co by było, gdybyśmy musieli napisać taką klasę, która dziedziczy na raz obydwie funkcjonalności - tego się nie da zrobić w obecnej wersji kodu. ☹

Druga kwestia..

Mamy w tym rozwiązaniu mnóstwo wirtualności – metody wirtualne wywołują inne metody wirtualne do zrobienia czegoś *de facto* bardzo prostego. Przy częstym wywoływaniu tych metod ujawni się nadmierny koszt czasu wykonania.

284

Wielodziedziczenie

Podejście #2 (1/3): zaczynamy kombinować..

A może by tak zastosować kompozycję zamiast dziedziczenia (*Oj!* ..)

```
class LoggingTask : public ITask {
 ITask* task_;
public:
 LoggingTask( ITask* task ) : task_( task ) { }
 ~LoggingTask() { delete task_; }
 virtual void Execute() {
  std::cout << "LOG: The task is starting - " << GetName().c_str() << std::endl;
  if( task_ ) task_->Execute();
  std::cout << "LOG: The task has completed - " << GetName().c_str() << std::endl;
 }
 virtual std::string GetName() {
  if( task_ ) return task_->GetName();
  else return "Unbound LoggingTask";
 }
};
```


285

Wielodziedziczenie

Podejście #2 (2/3):

Konstrukcja jest prosta:

`LoggingTask` implementuje **`Itask`**, a jednocześnie obiektowi tego typu przekazywany jest wskaźnik do obiektu **`ITask`** w wywołaniu konstruktora - nazwijmy ten obiekt zadaniem „dziecko”.

Zadanie „rodzic” **`LoggingTask`** deleguje swoją implementację metody **`GetName()`** do zadania „dziecko”. Deleguje do zadania „dziecko” również **`Execute()`**, ale dodatkowo wykonuje zapisy do pliku logowań przed i po tym delegowaniu.

Podobnie można wykonać drugą klasę odpowiedzialną za zapisanie czasu wykonania zadania.

286

Wielodziedziczenie

Podejście #2 (3/3):

Tworzymy jedną klasę **MyTask** implementującą właściwą czynność i dziedziczącą interfejs klasy reprezentującej zadanie tj. **Itask**:

```
class MyTask : public Itask {
public:
 virtual void Execute() {
  std::cout << "...This is where the task is executed..." << std::endl;
 }
 virtual std::string GetName() { return "My task name"; }
};
```

Dodawanie obydwu funkcjonalności przez kompozycję i delegację:

```
ITask* t = new LoggingTask(
                  new TimingTask(
                        new MyTask() ) );
t->Execute();
delete t;
```


Wielodziedziczenie

Rozwiązanie **#2** jest skuteczne, ale jakby nieco zawiłe.. ☺

1. Musimy pamiętać o czasie życia obiektu „dziecka"(w przedstawionym rozwiązaniu to klasy **LoggingTask** i **TimingTask** *adoptują* zadanie dziecko przekazane im w argumencie konstruktora i w swoich destruktorach usuwają to zadanie).
2. Musimy w kodzie uwzględniać ewentualność, że do konstruktora przekazany zostanie wskaźnik NULL
3. W razie przekazania wskaźnika NULL, metoda **GetName()** implementowana w klasach **LoggingTask** i **TimingTask** również musi coś zwrócić (obecnie zwraca napis **"Unbound TimingTask"**), ale tak właściwie to nie powinno być jej zadaniem.

To rozwiązanie choć realizuje zadaną funkcjonalność jest jeszcze gorsze od poprzedniego: oprócz kosztu wywołań metod pojawia się dodatkowe obciążenie zasobów – musimy robić alokacje na stercie (tworzenie i usuwanie procesów „dzieci") oraz sprawdzać, czy wskaźnik nie zawiera przypadkiem wartości NULL).

Wielodziedziczenie

Podejście #3 (1/3):

Wróćmy do kombinowania z dziedziczeniem..

Zacznijmy jeszcze raz, tym razem od klasy bazowej **MyTask**:

```
class MyTask : public ITask {
public:
 virtual void Execute() {
  std::cout << "...This is where the task is executed..." << std::endl;
 }
 virtual std::string GetName() {
  return "My task name";
 }
};
```


Wielodziedziczenie

Podejście #3 (2/3):

Funkcjonalność zapisania łącznego czasu wykonania zadania

```
class TimingTask : public MyTask {
protected:
 virtual void Execute() {
  timer_.Reset();
  MyTask::Execute();
  double t = timer_.GetElapsedTimeSecs();
  std::cout << "Task Duration: " << t << " seconds" << std::endl;
 }
};
```

Dlaczego metoda *Execute* jest zadeklarowana jako *protected?*

Aby tylko z poziomu wskaźnika do klasy bazowej *MyTask* można była ją wywołać, tj. poziomu, kiedy dysponujemy różnymi obiektami, o których wiemy tylko tyle, że reprezentują zadania – dziedziczą po *MyTask*.

Wielodziedziczenie

Podejście #3 (3/3):

Następnie – funkcjonalność zapisania w logu komunikatów o uruchomieniu oraz o zakończeniu zadania

```
class LoggingTask : public TimingTask {
protected:
 void Execute() {
  std::cout << "LOG: The task is starting: " << GetName().c_str() <<  std::endl;
  TimingTask::Execute();
  std::cout << "LOG: The task has completed: " << GetName().c_str() << std::endl;
 }
};
```

I gotowe. Czy jest lepiej?..

Wielodziedziczenie

Rozwiązanie daje pożądaną funkcjonalność, ale nie jest elastyczne.

Kod zapisujący czasy startu i zakończenia jest uzależniony od kodu zapisującego czas wykonania który jest uzależniony od **MyTask**, a żadna z klas nie jest szczególnie gotowa do wielokrotnego użytku.

Ma jednak swoje zalety:

- Nie ma konieczności robienia alokacji na stercie.
- Nie ma nadmiarowych sprawdzeń wskaźników.
- Nie ma potrzeby ustalania kto jest odpowiedzialny za czas życia obiektów.
- Nie ma (zbędnych) definicji metod wirtualnych.
- Nie ma (zbędnych) wywołań metod wirtualnych.
- Kompilator ma szansę na dokonanie optymalizacji kodu poprzez na przykład zamianę wywołań metody **Execute()** na wersję **inline**.

To rozwiązanie warto dalej rozwijać dążąc do wersji w pełni elastycznej.

Wielodziedziczenie

Podejście #4: finalna wersja kodu (wykorzystująca klasy-domieszki)

Klasy-domieszki bazują na następującym idiomie:

```
template< class T >
class MyMixin : public T
{
  // tutaj różne składowe klasy
  …
};
```

Jest to szablon klasy, w którym wartość parametru reprezentuje klasę bazową dla tej klasy.

Wielodziedziczenie

Podejście #4: finalna wersja kodu

Domieszka reprezentująca zapisywanie w logu komunikatów o uruchomieniu oraz o zakończeniu zadania

```
template< class T >
class LoggingTask : public T {
public:
 void Execute() {
  std::cout << "LOG:The task is starting: "<< T::GetName().c_str() << std::endl;
  T::Execute();
  std::cout << "LOG:The task has completed: "<< T::GetName().c_str() << std::endl;
 }
};
```


Wielodziedziczenie

Podejście #4: finalna wersja kodu

Domieszka reprezentująca zapisywanie czasu wykonania zadania

```
template< class T >
class TimingTask : public T
{
 Timer timer_;
public:
 void Execute() {
  timer_.Reset();
  T::Execute();
  double t = timer_.GetElapsedTimeSecs();
  std::cout << "Task Duration: " << t << " seconds" << std::endl;
 }
};
```


Wielodziedziczenie

Podejście #4: finalna wersja kodu

Klasa implementująca właściwą czynność realizowaną przez to zadanie

(uwaga: klasa nie dziedziczy po niczym)

```
class MyTask {
public:
 virtual void Execute() {
  std::cout << "...This is where the task is executed..." << std::endl;
 }
 virtual std::string GetName() {
  return "My task name";
 }
};
```


Wielodziedziczenie

Podejście #4:

- Klasy **`TimingTask`** oraz **`LoggingTask`** są klasami-domieszkami.
- Klasa **`MyTask`** nie jest klasą-domieszką i tak miało być.
- Klasa **`MyTask`** reprezentuje jedno konkretne zadanie (w przeciwieństwie do domieszek, które mają charakter uniwersalny, ponieważ muszą znaleźć zastosowanie w wielu miejscach kodu)

Wielodziedziczenie

Podejście #4: finalna wersja kodu

```
MyTask t1; // Obiekt reprezentujący zadanie
t1.Execute();
TimingTask< MyTask > t2; // reprezentuje zadanie domieszkowane pomiarem czasu
t2.Execute();
LoggingTask< TimingTask< MyTask > > t3;    // reprezentuje zadanie domieszkowane
                                           // logowaniem i pomiarem czasu
t3.Execute();

// Obiekt reprezentujący zadanie domieszkowane pomiarem czasu i logowaniem
typedef TimingTask<
                LoggingTask<
                        MyTask > > TLTask;
TLTask t4;
t4.Execute();
```

Wielodziedziczenie

Gotowe.

Rozwiązanie jest elastyczne i efektywne, klasy domieszki są gotowe do wielokrotnego użytku w wielu miejscach kodu,

ale..

kodu nie można włączyć do Menedżera Zadań zarządzającego zadaniami do asynchronicznego wykonania ponieważ brak dziedziczenia po interfejsie **`ITask`**. ☹

Rozwiązanie:

Oprócz klasy **`MyTask`** reprezentującej zadanie asynchroniczne, należy też utworzyć uniwersalny adapter włączający **`ITask`** do hierarchii dziedziczenia.

299

Wielodziedziczenie

Podejście #4: ostatni element – adapter

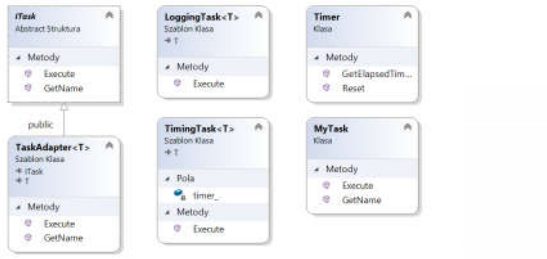
pozwala na wprowadzenie obiektu reprezentującego zadanie do asynchronicznego wykonania do Menedżera Zadań zarządzającego tymi zadaniami

```
template< class T >
class TaskAdapter : public ITask, public T {

public:
 virtual void Execute() {
  T::Execute();
 }

 virtual std::string GetName() {
  return T::GetName();
 }
};
```


300

Wielodziedziczenie

Podejście #4: ostatni element – adapter



301

Wielodziedziczenie

Podejście #4: adapter – wykorzystanie w kodzie

```
// typedef for our final class, inlcuding the TaskAdapter<> mixin
typedef public TaskAdapter<
                    LoggingTask<
                          TimingTask<
                                MyTask > > > task;

// instance of our task - note that we are not forced
// into any heap allocations!
task t;

// implicit conversion to ITask* thanks to the TaskAdapter<>
ITask* it = &t;
it->Execute();
```


302

Wielodziedziczenie

Podsumowanie podejścia #4: warsztat niezbędny do pisania kodu

.. jest gotowy:

1. Mamy klasy (szablony klas) domieszki wprowadzające odpowiednie cechy do tworzonego typu klasy reprezentującej zadanie asynchroniczne (**`TimingTask`** oraz **`LoggingTask`**).
2. Mamy przykład klasy reprezentującej zadanie (**`MyTask`**).
3. Mamy uniwersalny adapter, który łączy bazową klasę-interfejs **`ITask`** z klasą reprezentującą zadanie (**`MyTask`**) oraz z domieszkami (**`TimingTask`** oraz **`LoggingTask`**).
4. Wiemy, jak deklarować obiekty typu adapter i używać ich w kodzie programu (np. w kodzie Menedżera Zadań) za pomocą wskaźników do klasy-interfejsu **`ITask`** .

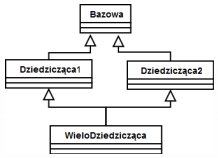
 Cel osiągnięty – koniec przykładu.

303

Wielodziedziczenie

W przypadkach hierarchii dziedziczenia typu romb w instancji obiektu **`WieloDziedzcząca`** mogą się pojawić dwie instancje klasy **`Bazowa`**.

Pożądane jest utworzenie rzeczywistego rombu dziedziczenia, tj. aby obiekt klasy '**Bazowa**' był jeden, współużytkowany przez znajdujące się wewnątrz obiektu klasy '**Wielodziedzicząca**' podobiekty '**Dziedzicząca1**' i '**Dziedzicząca2**'.



W tym celu deklarujemy klasę 'Bazowa' jako *wirtualną klasę bazową.*

304

Wielodziedziczenie

Konflikt nazw – wskazanie kompilatorowi właściwego wyboru:

```
class Bazowa
{
public:
  virtual ~Bazowa() {}
};

class Dziedziczaca1 : virtual public
    Bazowa
{
public:
  void f() {}
};

class Dziedziczaca2 : virtual public
    Bazowa {
public:
  void f() {}
};

class WieloDziedziczaca : public
    Dziedziczaca1, public
    Dziedziczaca2
{
public:
  using Dziedziczaca1::f;
};

int main() {
  WieloDziedziczaca b;
  b.f(); // wywołuje Dziedziczaca1::f()
}
```

Wirtualna klasa bazowa

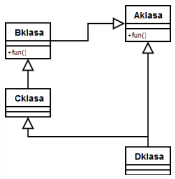
Wskazanie kompilatorowi właściwego wyboru

Wielodziedziczenie

W zależności od kształtu hierarchii dziedziczenia kompilator może sam próbować rozstrzygnąć której wersji zduplikowanych składowych używać.

`Aklasa` jest klasą bazową (bezpośrednio lub pośrednio) klasy **`Bklasa`**, dlatego **`Bklasa`** jest *niżej* w hierarchii dziedziczenia i wersja **`Bklasa::fun`** dominuje nas **`Aklasa::fun`**.

W sytuacji, gdy można określić kolejność dominowania, kompilator sam rozstrzyga, której wersji użyć.



Wielodziedziczenie

Podsumowanie:

- istnieją dwa podstawowe paradygmaty programowania wykorzystywane przy stosowaniu wielodziedziczenia:
  1. klasy interfejsu
  2. klasy domieszek
- Jeżeli w hierarchii klas nie występuje struktura typu *romb*, kompilator korzystając z reguł dominacji sam rozstrzyga, której wersji użyć.
  W przeciwnym przypadku należy mu to wskazać

Wielodziedziczenia należy unikać wszędzie tam, gdzie nie jest konieczne.