Wzorce projektowe dekorator i adapter

Przygotował Paweł Juszczyk

Kilka prawd o wzorcach projektowych

* Wzorce projektowe należy wykorzystywać tylko wtedy, gdy jest to konieczne (zasada **YAGNI***).
* Wzorce polepszają komunikację wewnątrz zespołu i jakość kodu, a przez to produkcyjność zespołu.
* Wzorce różnią się powodem wykorzystania, a nie implementacją. Wiele różnych wzorców może mieć taką samą implementację.
* Wzorce rzadko występują pojedynczo. Zwykle łączy się kilka różnych wzorców.

*YAGNI (en. You ain't gonna need it) - „I tak nie będziesz tego potrzebował". Inne podobne zasady do KISS (en. Keep it simple stupid) oraz DRY (en. Don't repeat yourself)

Dekorator

* Rozszerza funkcjonalność powstałej wcześniej klasy
* Rozróżniamy dekorator statyczny (klasowy) i dynamiczny (obiektowy)
* Zwykle wykorzystywany w strumieniach, systemach logowania, rozszerzaniu standardowych bibliotek

Dekorator statyczny implementacja (cz. I)

```
class Base {

      public:
            virtual void func() {
                  cout << "Base::func()" << endl;
            }

};
```

Dekorator statyczny implementacja (cz. II)

```
class Child : public Base {

    public:
        void log() {
            cout << "Child::log()" << endl;
        }

        void func() {
            Base::func();
            log();
        }

};
```

Dekorator dynamiczny implementacja (cz. I)

```
class AbstractChild {
      public:
            virtual void func() = 0;
            virtual ~AbstractChild() {}
};
```

Dekorator dynamiczny implementacja (cz. II)

```
class Child : public AbstractChild {
      public:
            void func() {
                  cout << "Child::func()" << endl;
            }
};


 class Child2 : public AbstractChild {
       public:
             void func() {
                   cout << "Child2::func()" << endl;
             }
 };
```

Dekorator dynamiczny implementacja (cz. III)

```
class Dekorator{

        private:
                AbstractChild* a;

        public:
                Dekorator(int i) {
                        if(i < 20) a = new Child;
                        else a = new Child2;
                }

                void log() {
                        cout << "Dekorator::log()" << endl;
                }

                void func() {
                        a->func();
                        log();
                }
};
```

Adapter

* Umożliwia wykorzystanie różnych niezależnych i niespójnych interfejsów w zależności od konfiguracji
* Bardzo często wykorzystywany w połączeniu ze wzorcami kreacyjnymi
* Częściowo realizuje dwie zasady SOLID

Zasady SOLID

* **S**ingle Responsibility Principle
* **O**pen-Close Principle
* **L**iskov Substitution Principle
* **Interface Segregation Principle** (zasada częściowo realizowana przez wzorzec adapter)
* **Dependency Inversion Principle** (zasada częściowo realizowana przez wzorzec adapter)

Adapter implementacja (cz. I)

```
class Abstract {

      public:
            virtual void insert() = 0;
            virtual void update() = 0;
            virtual void remove() = 0;
            virtual ~Abstract() {}

};
```

Adapter implementacja (cz. II)

```
class MySQL : public Abstract {

        public:
                void insert() {
                        cout << "MySQL::insert()" << endl;
                }
                void update() {
                        cout << "MySQL::update()" << endl;
                }
                void remove() {
                        cout << "MySQL::remove()" << endl;
                }

};
```

Adapter implementacja (cz. III)

```
class PostgreSQL : public Abstract {

public:
	void insert() {
		cout << "PostgreSQL::insert()" << endl;
	}
	void update() {
		cout << "PostgreSQL::update()" << endl;
	}
	void remove() {
		cout << "PostgreSQL::remove()" << endl;
	}

};
```

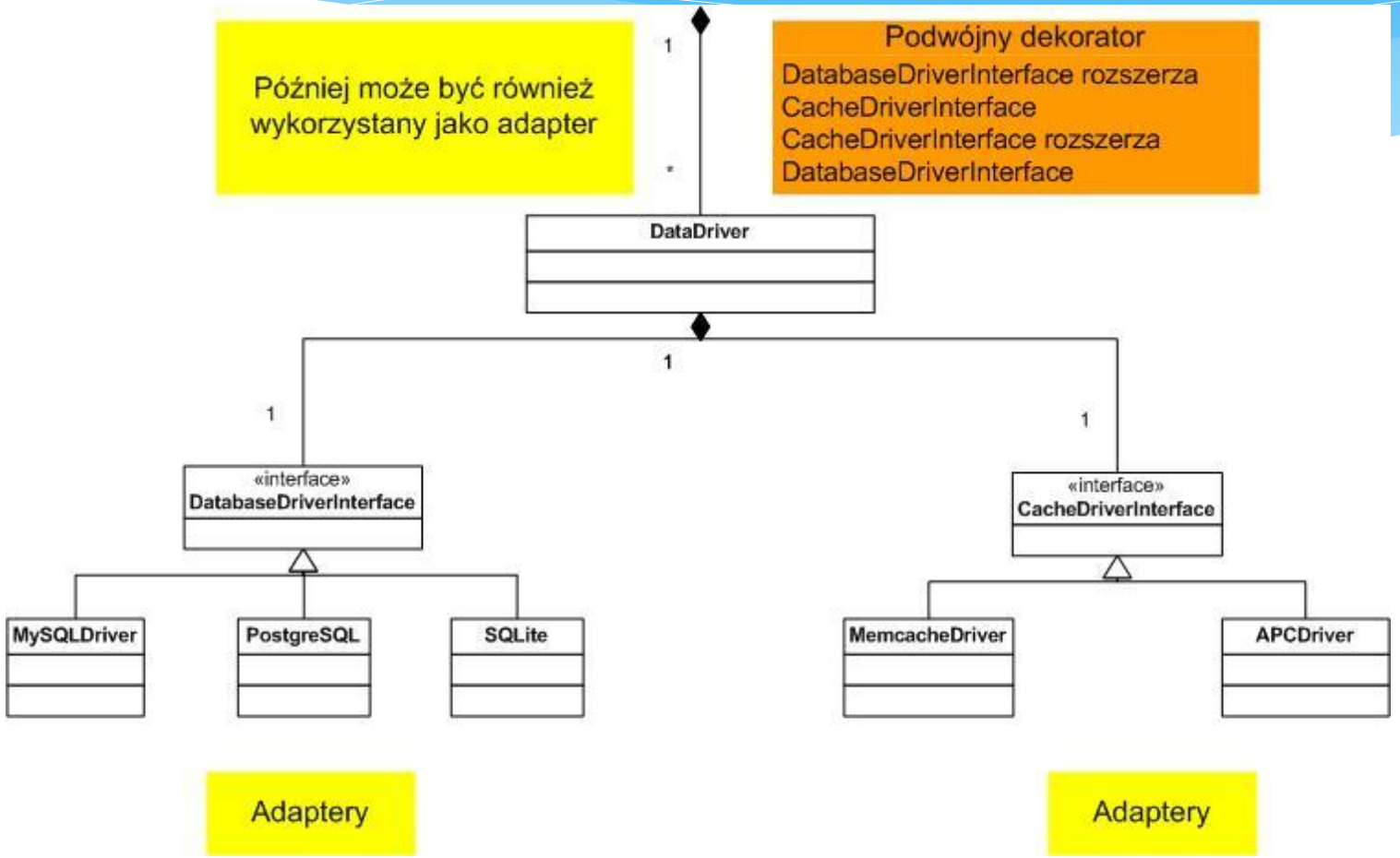
Adapter implementacja (cz. IV)

```
class SQLite : public Abstract {

	public:
		void insert() {
			cout << "SQLite::insert()" << endl;
		}
		void update() {
			cout << "SQLite::update()" << endl;
		}
		void remove() {
			cout << "SQLite::remove()" << endl;
		}

};
```

Adapter implementacja (cz. V)

```
int main(int argc, char** argv) {
        if(argc <= 1) exit(1);
        int cond = atoi(argv[1]);
        Abstract* a;
        if(cond == 1) a = new MySQL;
        else if(cond == 2) a = new PostgreSQL;
        else a = new SQLite;
        a->insert();
        a->update();
        a->remove();
}
```

Finał – łączenie wzorców



Literatura

* Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture. Addison-Wesley*
* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
  Design Patterns:
  Elements of Reusable Object-Oriented Software
* http://www.objectmentor.com/