Лекція 3. Композиція об’єктів

1. *Класифікація об’єктів. Наслідування чи включення?*
2. *Конструктор об’єкта-композита. Наслідування реалізації. Делегування.*
3. *Ієрархія об’ємних геометричних фігур «Циліндр», «Призма», «Піраміда», «Конус».*
4. *Функціонал абстрактного класу.*
5. *Особливості класів, що використовують динамічну пам’ять.*
6. *Приведення типу поліморфного вказівника.*
7. *Розширення функціональності класів. Вкладені класи.*
8. *Статичний фабричний метод.*

Продовжимо наше знайомство з можливостями класів щодо моделювання сутностей реального світу. Розглянемо для цього наступне завдання.

> *Завдання.* Оголосіть ієрархію класів, що моделюють сутності «піраміда», «циліндр», «конус», «паралелепіпед», «призма». Усі фігури повідомляють свої об’єм, площу основи, площу бічної та площу всієї поверхні, вміють будувати своє зображення рядком літер.
> Використайте для цього оголошені раніше класи, що моделюють плоскі фігури.

Для побудови ієрархії потрібно класифікувати поняття, які ми збираємося моделювати. Перше, що спадає на думку – зробити циліндр підкласом круга, а паралелепіпед – прямокутника. Все логічно: додаємо в підкласах поле для зберігання висоти, визначаємо методи обчислення об’єму та площ – і все готово. На рис. 1 – схема такої ієрархії.

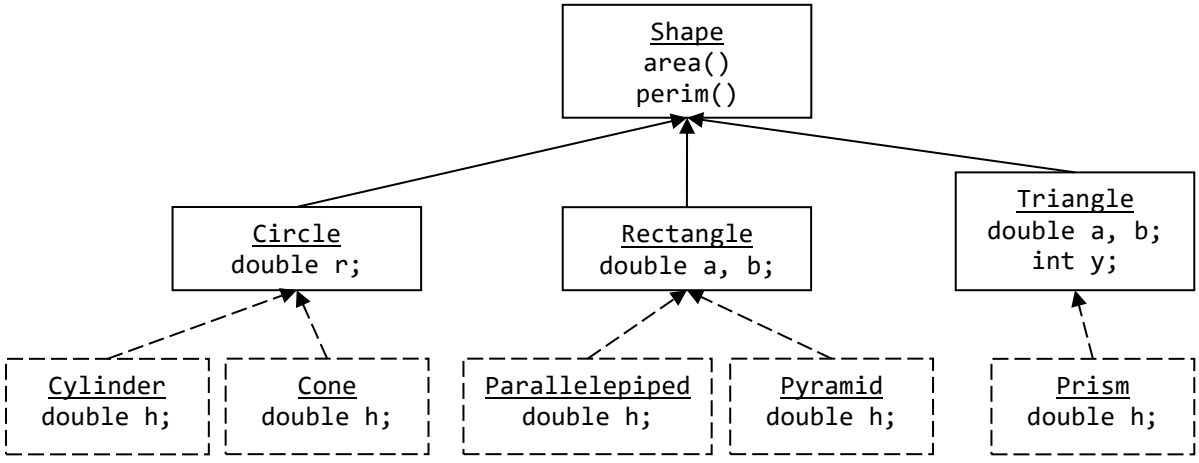

Рис. 1. Проект оновленої ієрархії класів

У якому класі оголосити метод обчислення об’єму? Якщо на найнижчому рівні ієрархії, то це означатиме, що здатність повідомляти об’єм – унікальна можливість окремо кожного класу, і скористатися нею можна буде лише в тому випадку, коли точно відомо тип об’єкта. Але це спільна властивість усіх об’ємних фігур! Про неї має бути відомо на рівні оголошення протоколу взаємодії з ієрархією класів. Так, щоб у будь-якої фігури можна було довідатися її об’єм, незалежно від її типу.

Протокол взаємодії визначають у базовому класі, то ж оголосимо метод обчислення об’єму на найвищому рівні – в класі *Shape*. Тепер ми зіткнемося з іншою проблемою: як обчислювати об’єм у класах плоских фігур? Просто повертати нуль? Але круг не має об’єму. Він для нього просто невизначений. Це не єдина суперечність у запропонованій вище схемі наслідування. Якщо *Cylinder* підклас *Circle*, то це означає, що «циліндр – особливий різновид круга». Такий собі «товстий круг». :) Знову щось не те…

Давайте спочатку розберемося з простішим завданням: як побудувати клас «паралелепіпед» із використанням класу «прямокутник»? У попередніх лекціях ми обговорювали можливості наслідування класів. Пригадуємо, що відкрите наслідування моделює відношення «*is-a*» або «*є*» українською. Квадрат є прямокутник з рівними сторонами, квадрат є особливим випадком або різновидом прямокутника. Якщо б у нас виникла потреба у фігурах, що пам'ятають перелік своїх вершин і використовують їх при виведенні на друк, ми могли б оголосити новий підклас прямокутника – іменований прямокутник, як на рис. 2.

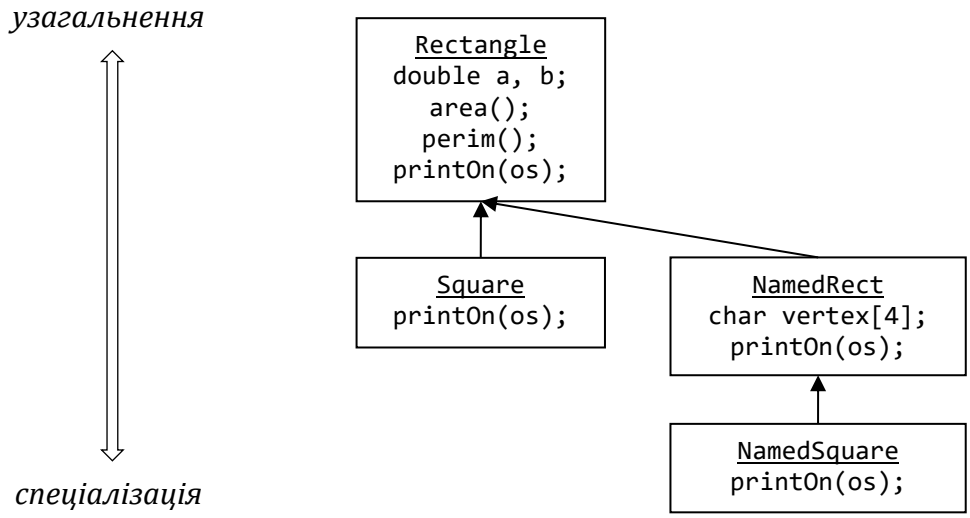

Рис. 2. Ієрархія класу «прямокутник»

Нагорі дерева класів розташовано найбільш загальний клас, унизу – найбільш спеціалізований, з особливими можливостями. У попередній лекції ми узагальнили прямокутник, круг і трикутник і ввели у використання поняття «довільна плоска фігура» (що має площу і периметр). Намагання вбудувати в цю ієрархію клас об'ємної фігури означатиме, що ми трактуємо паралелепіпед як «товстий» прямокутник.

А як є на практиці? Кожна бічна грань, основи прямокутного паралелепіпеда є прямокутниками. Якби ми будували розгортку паралелепіпеда, то сказали б, що він складається з шести прямокутників. У нас інше завдання: обчислити об'єм і площі. Пригадаємо, $V = a\,b\,c$, $S_б = 2(a + b)\,c$, $S_о = a\,b$, де a і b – сторони основи. З огляду на ці формули, паралелепіпед містить основу-прямокутник і висоту, а формули можна переписати так: $V = S_о\,h$, $S_б = P_о\,h$, де $S_о$ – площа прямокутника, а $h = c$ – висота.

Паралелепіпед *містить* (англійською «*has-a*») прямокутник у якості основи. Мовою програмування відношення «*has-a*» моделюють *композицією* – включенням одного об'єкта до складу іншого. Оголошення класу «паралелепіпед» матиме вигляд:

```
#include "..\FlatShapes\FlatShapes.h"

/* Паралелепіпед містить об'єкт прямокутник і дійсне число - основу і висоту.
   Паралелепіпед не наслідує інтерфейс прямокутника, натомість він отримує
   його реалізацію для власних потреб.
*/
class Parapd
{
private:
      Rectangle base;  // тут міститимуться сторони основи паралелепіпеда
      double h;        // тут - сторона, перпендикулярна до основи
public:
      // синтаксис ініціалізатора основи в конструкторах паралелепіпеда
      // подібний до оголошення об'єкта-прямокутника, при цьому працюють
      // конструктори класу Rectangle
      Parapd() :base(), h(1.) {}
      Parapd(double a, double b, double c) :base(a, b), h(c) {}
      void printOn(ostream& os) const
```

```
	{
		os << "Parallelepiped(" << h << " x " << base << ')';
	}
	// Про площу основи паралелепіпед питає свою основу. Таку поведінку
	// називають делегуванням
	double baseArea() const
	{
		return base.area();
	}
	// Інші обчислення дещо складніші. Вони також використовують функціонал основи
	double volume() const
	{
		return base.area()*h;
	}
	double sideArea() const
	{
		return base.perim()*h;
	}
	// Площа поверхні залежить від площ частин.
	double surfaceArea() const
	{
		return this->baseArea()*2. + this->sideArea();
	}
};
ostream& operator<<(ostream& os, const Parapd& p)
{
	p.printOn(os); return os;
}
```

У результаті включення клас-композит наслідує реалізацію вкладеного об'єкта, але не його інтерфейс. Екземпляр класу *Parapd* не зможе відповісти на повідомлення *perim()* чи *area()*, як це робить екземпляр *Rectangle*. Натомість він використовує вкладений об'єкт *base* для зберігання двох своїх сторін і для виконання обчислень. Наприклад, при обчисленні площі бічної поверхні суму сторін основи рахує *base.perim()*.

Метод *Parapd::baseArea* лише огортає виклик *base.squre()* – всю роботу виконує метод вкладеного об'єкта. Такий спосіб поведінки називають *делегуванням*: клас-композит делегує «почесний обов'язок» виконати всю роботу вкладеному об'єктові.

Зверніть увагу на визначення конструкторів класу-композита: запис *base()* у списку ініціалізації означає виклик конструктора за замовчуванням класу *Rectangle*, а *base(a,b)* і іншому конструкторі – виклик конструктора з параметрами цього класу.

Повернімося до завдання, сформульованого на початку лекції. Чому ієрархія, зображена на рис. 1, невдала? Проблема в тому, що ми знову піддалися спокусі класифікувати об'єкти за їхньою структурою, а не за поведінкою. Циліндр схожий не до круга, а до прямокутного паралелепіпеда та до прямої призми: усі вони мають об'єм, який рахуємо за однією формулою $V = S_O \times h$. Натомість циліндр *містить* круг (як основу). Англійською «*Cylinder* has a *Circle*». У програмі, як уже було сказано, таке відношення моделюють включенням одного об'єкта до складу іншого:

```
class Cylinder
{
protected:
	Circle base;
	double h;
public:
	Cylinder(double radius=1., double height=1.) : base(radius), h(height) {}
	. . . . .
};
```

Для того, щоб класифікувати згадані у завданні об'ємні фігури, пригадаємо формули для обчислення їхніх об'ємів та різноманітних площ.

Фігура	Об'єм	Площа бічної поверхні	Площа всієї поверхні
Прямокутний паралелепіпед	$V = S_{\text{О}} \times h$	$S_{\text{б}} = P_{\text{О}} \times h$	$S = 2S_{\text{О}} + S_{\text{б}}$
Пряма призма	$V = S_{\text{О}} \times h$	$S_{\text{б}} = P_{\text{О}} \times h$	$S = 2S_{\text{О}} + S_{\text{б}}$
Піраміда	$V = \frac{1}{3} S_{\text{О}} \times h$	Залежно від типу піраміди	$S = S_{\text{О}} + S_{\text{б}}$
Конус	$V = \frac{1}{3} S_{\text{О}} \times h$	$S_{\text{б}} = \pi \times r \times L$	$S = S_{\text{О}} + S_{\text{б}}$
Циліндр	$V = S_{\text{О}} \times h$	$S_{\text{б}} = C \times h$	$S = 2S_{\text{О}} + S_{\text{б}}$

Легко бачити, що однакову поведінку демонструють паралелепіпед, призма і циліндр (довжина круга *C* у формулі для обчислення $S_{\text{б}}$ циліндра – це той же периметр основи). Усі вони є *прямими фігурами*. Піраміда і конус – представники *конічних фігур*. І прямі, і конічні є *об'ємними фігурами*. Таким чином ми приходимо до ієрархії, схематично зображеної на рис. 3.

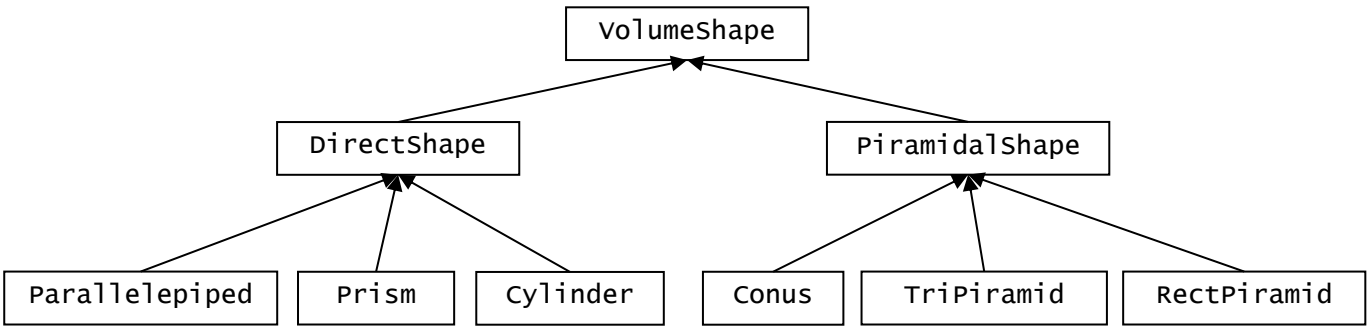

Рис. 3. Проект ієрархії об'ємних фігур

Тепер з'ясуємо, що з певністю можна сказати про обов'язки кожного з цих класів. Перш за все доведеться вирішити, на якому рівні ієрархії міститимуться поля даних. Можна було б використати досвід, здобутий у попередньому завданні, і оголосити висоту й основу в кожному з конкретних класів фігур. Класи *Parallelepiped* і *RectPiramid* матимуть *Rectangle base*, а *Cylinder* і *Cone* – *Circle base*. Це хороший вибір, оскільки кожна фігура «знатиме» конкретний тип своєї основи. Але є декілька «але». За такого вибору існування класів *DirectShape* та *PiramidalShape* втрачає сенс. Справді, ми не зможемо оголосити в них методи обчислення об'єму чи площі за загальними для всіх прямих фігур і для всіх конічних фігур формулами, бо не оголошено поля даних. Ми розташуємо поля даних у кореневому класі. Зрештою, *кожна* об'ємна фігура має основу і висоту. Типом основи тепер буде *Shape** – поліморфний вказівник на плоску фігуру. Відмова від конкретного типу на користь поліморфного дещо ускладнює завдання, але ми не боїмося труднощів – так, навіть, цікавіше! Адже ООП – це «програмування за допомогою надсилання повідомлень об'єктам невідомого типу»!

Клас *VolumeShape* – найвищий рівень абстракції. Він описує *довільну* об'ємну фігуру. Відомо, що вона має основу (*якусь* плоску фігуру) і висоту. «Якась плоска фігура» – це вказівник на базовий тип *Shape* (поліморфний вказівник), висота – дійсне число. У класі *VolumeShape* потрібно також оголосити протокол взаємодії з ієрархією об'ємних фігур. Більшість методів будуть абстрактними, бо поки що не відомо, наприклад, як обчислити об'єм. Проте дещо конкретне можна сказати вже тепер. Розглянемо оголошення абстрактного класу *VolumeShape*.

```
class VolumeShape
{
protected:
	double h;
	Shape* base;
public:
	VolumeShape(double high=1., Shape* s=0) : h(high), base(s) {}
	virtual ~VolumeShape() { delete base; }
// методи обчислення площ:
	virtual double baseArea() const { return base->area(); }  // основи
	virtual double sideArea() const = 0;                      // бічної поверхні
	virtual double surfaceArea() const = 0;                   // повної поверхні
// метод обчислення об'єму
	virtual double volume() const = 0;
// виведення в потік зовнішнього вигляду об'єкта
	virtual void printOn(ostream&) const = 0;
	char * baseToStr() const { return base->toStr(); }
};
```

Площею основи всякої об'ємної фігури є площа тієї плоскої фігури, яка лежить в основі. У загальному випадку ми не знаємо її типу, але точно знаємо, що вона вміє обчислювати свою площу. Тому вже на рівні абстрактного базового класу *VolumeShape* вдається повністю визначити метод *baseArea()*: об'ємна фігура не повинна турбуватися про тип своєї основи та вишукувати формули для обчислень, бо це є відповідальність самої основи. Даний простий приклад наочно демонструє потужність і елегантність об'єктно-орієнтованого підходу.

Подібно до обчислення площі основи легко можна отримати і рядок-зображення основи. Цього досягають простою переадресацією повідомлення відповідному об'єктові.

Клас *DirectShape* описує всі прямі фігури. Він все ще досить загальний, але вже на цьому рівні, завдяки схожості фігур, вдається описати більшість необхідної функціональності.

```
class DirectShape : public VolumeShape
{
public:
	DirectShape(double high=1., Shape* s=0) : VolumeShape(high,s) {}
	virtual double sideArea() const
	{	return base->perim() * h;
	}
	virtual double surfaceArea() const
	{	return baseArea() * 2 + sideArea();
	}
	virtual double volume() const
	{	return base->area() * h;
	}
	virtual void printOn(ostream&) const;
};
```

Класи *Parallelepiped*, *TriPrism*, *Cylinder* мають зовсім небагато обов'язків: вони повинні лише створити відповідний об'єкт-основу. (Тут TriPrism – трикутна призма.)

Ми знаємо з попередніх лекцій, що класи, які в своїх екземплярах використовують динамічну пам'ять, повинні перевизначити конструктор копіювання і оператор присвоєння. Так і зробимо.

```
class Cylinder : public DirectShape
{
public:
	Cylinder(double high=1., double radius=1.): DirectShape(high, new Circle(radius)){ }
	Cylinder(const Cylinder& c): DirectShape(c)
	{	base = new Circle(*dynamic_cast<Circle*>(c.base));
	}
	Cylinder& operator=(const Cylinder& c)
	{	if (this != &c)
		{	h = c.h;
			delete base;
```

```
                        base = new Circle(*dynamic_cast<Circle*>(c.base));
                }
                return *this;
        }
};
class Parallelepiped : public DirectShape
{
public:
        Parallelepiped(double high=1., double sideA=1., double sideB=1.):
                        DirectShape(high, base = new Rectangle(sideA,sideB)) { }
        Parallelepiped(const Parallelepiped& p): DirectShape(p)
        {       base = new Rectangle(*dynamic_cast<Rectangle*>(p.base));
        }
        Parallelepiped& operator=(const Parallelepiped& p)
        {       if (this != &p)
                {       h = p.h;
                        delete base;
                        base = new Rectangle(*dynamic_cast<Rectangle*>(p.base));
                }
                return *this;
        }
};
class TriPrism : public DirectShape
{
public:
        TriPrism(double high=1., double sideA=3., double sideB=4., int angle=90):
                        DirectShape(high, new Triangle(sideA,sideB,angle)) { }
        TriPrism(const TriPrism& t): DirectShape(t)
        {       base = new Triangle(*dynamic_cast<Triangle*>(t.base));
        }
        TriPrism& operator=(const TriPrism& t)
        {       if (this != &t)
                {       h = t.h;
                        delete base;
                        base = new Triangle(*dynamic_cast<Triangle*>(t.base));
                }
                return *this;
        }
};
```

Маємо ще одну тему до обговорення: хто повинен знищити динамічний об'єкт *base*? Логіка перша: хто каже *new*, той каже *delete*. Тоді в кожному з зазначених вище класів мав би бути деструктор з інструкцією *delete base*;. Але власником *base* є екземпляр базового класу, тому, за логікою номер два, основу знищує деструктор класу *VolumeShape*. Саме так є у запропонованій реалізації класів. Компілятор автоматично генерує віртуальні деструктори кожного з підкласів. Знищення екземпляра, наприклад, класу *Cylinder* відбувається в такій послідовності:

`компілятор → ~Cylinder() → ~DirectShape() → ~VolumeShape()`

і все працює, як треба. На практиці можна застосовувати або перший, або другий варіант. Ми використали деструктор базового класу як більш надійний і менш багатослівний варіант.

Класи об'ємних фігур використовують динамічну пам'ять для зберігання вкладеного об'єкта-основи. Ця обставина змушує нас перевизначити також конструктор копіювання і конструктор присвоєння. Справді, автоматично згенерований конструктор виконує поверхневе копіювання екземпляра: створює копії значень його полів. Це добре для дійсних чисел (висоти), але не підходить для вказівників (основи). Копіювання значення вказівника – адреси – в інший екземпляр означатиме, що дві об'ємні фігури поділяють одну основу. Як тільки одна з фігур перестане існувати, в іншої виникнуть проблеми.

На перший погляд, визначення конструктора копіювання класу *Cylinder* не мало б викликати жодних труднощів, адже в класі *Circle* є свій конструктор копіювання, то й основу циліндра ми зможемо копіювати без проблем.

```
Cylinder ::Cylinder(const Cylinder& c): DirectShape(c)
{
	base = new Circle(*c.base);
}
```

Тут конструктор базового класу скопіює висоту, а код в тілі конструктора використає розіменований вказівник *c.base*, щоб отримати круг циліндра *c* і скопіювати його. На жаль, такий код навіть не відкомпілюється. Причина в тому, що для компілятора *c.base* вказує на *Shape*, а не на *Circle*, а в класі *Circle* нема конструктора з параметром типу *Shape*. Саме тому в оголошені класів *Cylinder*, *Parallelepiped*, *TriPrism* (див. код вище) використано *динамічне приведення типу вказівника*. Це безпечний спосіб перетворення поліморфного вказівника на конкретний. Докладніше поговоримо про нього в одній з наступних лекцій.

Оператор присвоєння влаштовано схожим чином.

Інша вітка ієрархії, конічні фігури, влаштована дещо інакше. Методи обчислення бічної площі доведеться визначати аж на рівні конкретних класів, оскільки для різних фігур застосовують різні формули. Стосовно пірамід нам доведеться зробити деякі припущення. Клас *TriPiramid* моделюватиме трикутну піраміду, вершина якої проектується в центр вписаного в основу кола. У такої піраміди всі апофеми однакові. Їх величину можна обчислити за формулами $h_6 = \sqrt{h^2 + r_O^2}$, $r_O = 2S_O / P_O$. Пригадуємо, що потрібні для обчислення площі та периметра трикутника методи вже є.

Клас *RectPiramid* моделюватиме чотирикутну піраміду, вершина якої проектується в точку перетину діагоналей основи. Як не крути, а для обчислення висот бічних граней потрібні довжини сторін прямокутника. Халепа, бо клас *Rectangle* не має методів доступу до полів даних. Можна було б спробувати обчислити їх за значеннями площі та периметра, але ми зробимо природніше для програмістів. Якщо наявний клас не забезпечує потрібної функціональності, оголосіть його підклас і визначте у ньому необхідні методи – розширте можливості базового класу! На щастя поля даних класу *Rectangle* оголошені в захищеній частині, не в закритій, і підкласи мають до них доступ. Тому в *class RectAB : public Rectangle* ми оголосимо методи *getA()*, *getB()* – і це все, що нам треба. Але де оголосити цей клас? Тут ми скористаємося ще однією чудовою можливістю C++ і оголосимо *RectAB* всередині класу *RectPiramid*. Навіть у його закритій частині! Зрештою, це спосіб вирішення проблеми прямокутної піраміди, і вона може не афішувати його.

```
class RectPiramid : public PiramidalShape
{
	class RectAB : public Rectangle
	{
	public:
		RectAB(double sideA=1., double sideB=1.) : Rectangle(sideA,sideB) {}
		double getA() const { return a; }
		double getB() const { return b; }
	};
public:
	RectPiramid(double high=1., double sideA=1., double sideB=1.)
		: PiramidalShape(high, new RectAB(sideA,sideB)) {}
	RectPiramid(const RectPiramid& p): PiramidalShape(p)
	{	base = new RectAB (*dynamic_cast<RectAB*>(p.base));
	}
	virtual double sideArea() const
	{	double a = dynamic_cast<RectAB*>(base)->getA();
		double b = dynamic_cast<RectAB*>(base)->getB();
		return a*sqrt(h*h+b*b*0.25)+b*sqrt(h*h+a*a*0.25);
	}
};
```

Решту класів і методів нескладно дописати самому за зразком наведених вище оголошень.

Можемо переглянути також діаграму класів цілої ієрархії.

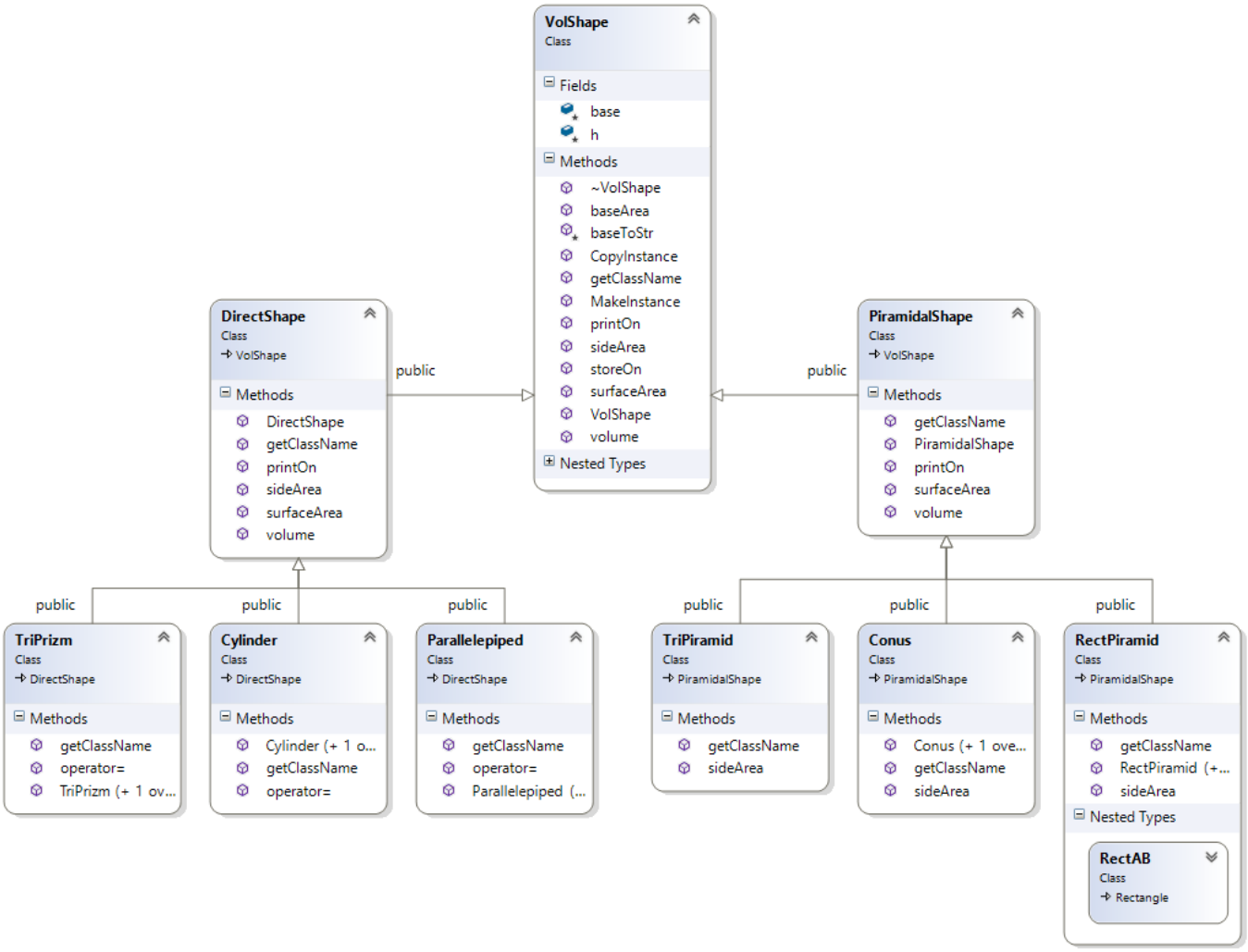

Рис. 4. Діаграма класів об’ємних фігур

Ще одна важлива тема для обговорення – зберігання об’єктів до файла та завантаження їх звідти. У попередній лекції ми використовували кодування типів плоских фігур окремими літерами латинського алфавіту. Для об’ємних фігур використаємо щось краще: можна запропонувати простий спосіб записування до файла повного імені типу екземпляра.

До екземплярів класів, що мають віртуальні методи, можна застосовувати оператор *typeid*. Він повертає посилання на екземпляр класу *type_info*, що докладно описує тип операнда. Зокрема, методом *name()* можна отримати рядок з іменем класу. Наприклад, якщо оголошено *Shape* s = new Circle();* то *typeid(*s).name()* поверне *'class Circle'*. Звертання до *typeid* заховано у віртуальні методи *getClassName()* кожного класу. Наявність таких методів дала змогу оголосити єдиний для всієї ієрархії метод зберігання об’єкта до файла:

```
void VolumeShape::storeOn(ofstream& fout) const
{
	const char * line = this->getClassName();
	for (int i = 6; line[i] != ' '; ++i) fout << line[i];
	fout << ' ' << this->h << ' ';
	this->base->storeOn(fout);
}
```

У програмі з попередньої лекції колекцію фігур з файла завантажував спеціальний фрагмент коду головної програми. Він розпізнавав типи фігур за літерою-кодом, читав значення параметрів і викликав конструктори відповідних класів. Було б добре мати зручний спосіб використання такого коду в різних частинах програми – всюди, де виникне потреба завантаження об’єктів з файла. Він тісно пов’язаний з оголошенням ієрархії класів-фігур, спільний для всіх підкласів, тому мав би розташовуватися «десь поруч». Вирішенням стане оголошення статичного методу в кореневому класі. Такий метод відіграє роль *фабричного*.

```
VolumeShape* VolumeShape::MakeInstance(std::ifstream& fin)
{
	int y; double h, r, a, b;
	string name;
	fin >> name;
	if (name == "Cylinder")
	{
		fin >> h >> r;
		return new Cylinder(h,r);
	}
	if (name == "Parallelepiped")
	{
		fin >> h >> a >> b; return new Parallelepiped(h,a,b);
	}
	if (name == "TriPrizm")
	{
		fin >> h >> a >> b >> y; return new TriPrizm(h,a,b,y);
	}
	if (name == "Conus")
	{
		fin >> h >> r; return new Conus(h,r);
	}
	if (name == "RectPiramid")
	{
		fin >> h >> a >> b; return new RectPiramid(h,a,b);
	}
	if (name == "TriPiramid")
	{
		fin >> h >> a >> b >> y; return new TriPiramid(h,a,b,y);
	}
	throw BadClassname(name.c_str());
}
```

У якості ідентифікаторів типу використано імена класів, ті, які самі об'єкти записують до файла в ході зберігання. Якщо файл з описом об'єктів створено «вручну», то може трапитися неправильно записане ім'я типу. Про цю помилку метод повідомляє винятком класу *BadClassname*.

Ми могли б використати будь-який стандартний тип винятків, але використання власного здалося цікавішим і більш інформативним, адже вирішальним серед винятків є саме тип. Для того, щоб підкреслити приналежність *BadClassname* до потреб ієрархії *VolumeShape*, його оголошено всередині відкритої частини класу *VolumeShape*.

```
// --------------------- абстрактний базовий клас
class VolumeShape
{
protected:
	double h;
	Shape* base;
	string baseToStr() const { return base->toStr(); }
public:
	// клас для опису винятка, що сигналізує про неправильне задання імені класу
	class BadClassname : public std::exception
	{
	public:
		BadClassname(const char* mess) :exception(mess){};
	};
. . .
};
```

Поза межами *VolumeShape* на клас винятків посилаються за допомогою кваліфікатора імені *VolumeShape::BadClassname*.