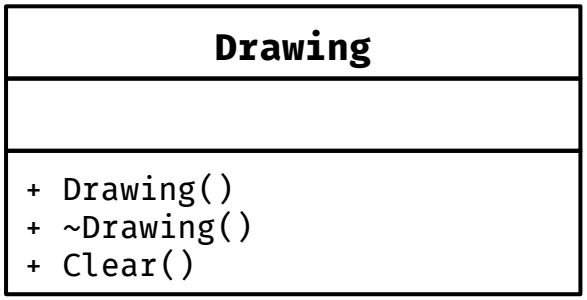
Клас

Синтаксична конструкція



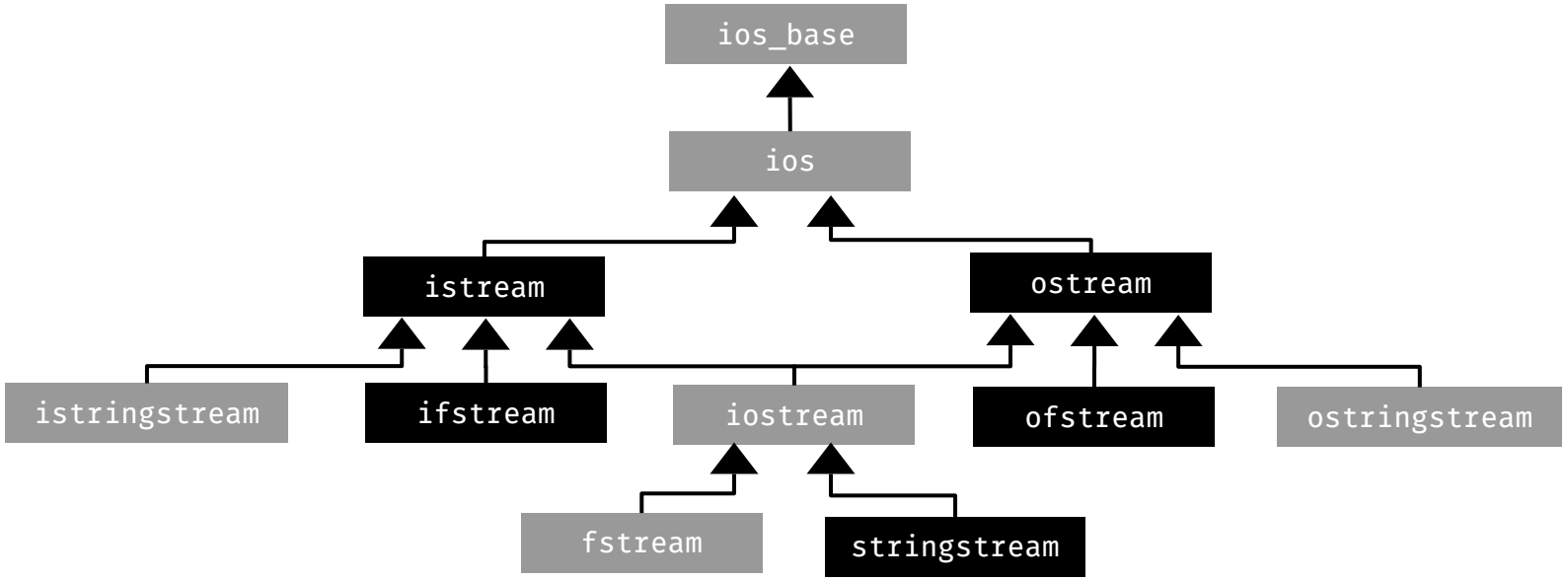

```
#include <iostream>
using namespace std;

class Drawing
{
public:
  Drawing()    { cout << "Drawing created\n";   }
  ~Drawing()   { cout << "Drawing destroyed\n"; }
  void Clear() { cout << "Drawing cleared\n";   }
};

int main()
{
  Drawing drawing;
  drawing.Clear();
}
```

- Ключове слово class взаємозамінне з **`struct`**, але класи, визначені через `struct`, за замовчанням мають відкритий (**`public`**) доступ до членів. Якщо вживано `class` - то за замовчанням доступ закритий (**`private`**).
- Тим не менш, завжди явно вказуйте тип доступу до членів класу. Використовуйте `struct` лише для позначення простих структур даних з мінімальною логікою або без неї (**POD** - plain old data).
- Механізм просторів імен (**`namespace`**) покликаний допомогти при дублюванні ідентифікаторів. Так, все з префіксом **`std::`** належить простору імен `std` стандартної бібліотеки.
- Не рекомендується включати увесь простір імен для уникнення написання префіксів (**`using namespace ...`**), особливо в h-файли.
- Менше зло - вибіркові команди типу **`using std::cout;`**
- У прикладі оголошено та реалізовано 3 методи - конструктор за замовчанням (без аргументів), деструктор та метод-модифікатор. Це поганий приклад - **не включайте реалізацію методів в інтерфейс класу**.

Потоки введення-виведення

Ієрарїія класів, об’єкти для введення-виведення в консоль

std::istream - клас потоків введення
std::ostream - клас потоків виведення
std::iostream - клас потоків введення-виведення
std::fstream - клас потоку введення-виведення в файл
std::stringstream - клас потоку введення-виведення в текстовий рядок



Для класів-потоків перевантажені оператори << (як “помістити в потік”) та >> (як “дістати з потоку”):

```
std::cout << "Hello, user!";     // виведення в консоль (на екран) через об’єкт класу std::ostream
std::cerr << "Run-time error!"; // виведення в потік помилок через об’єкт класу std::ostream
std::clog << "Run-time debug";  // виведення в буферизований потік журналу класу std::ostream
std::cin  >> i;                 // введення з консолі (клавіатури) через об’єкт класу std::istream
```

Потоки введення-виведення

Форматування чисел при виведенні, наприклад, на екран

```
#include <iostream>
#include <iomanip>

int main ()
{
  int i;
  std::cout << "i = ";
  std::cin >> i;
  std::cout << std::hex << std::setw(5) << std::left << i << std::endl;
  std::cout.flags (std::ios::right | std::ios::hex | std::ios::showbase);
  std::cout.width (10);
  std::cout << i << std::endl;

  double d;
  std::cout << "d = ";
  std::cin >> d;
  std::cout.unsetf ( std::ios::floatfield );
  std::cout.precision(10);
  std::cout << d << std::endl;
  std::cout.setf( std::ios::fixed, std:: ios::floatfield );
  std::cout << d << std::endl;
  std::cout << std::scientific << d << std::endl;

}
```

Результат:

```
i = 12345
3039
    0x3039
d = 123.45
123.45
123.4500000000
1.2345000000e+002
```

Потоки введення-виведення

Клас std::string, перетворення “число-текст”

```
#include <iostream>
#include <string>
#include <sstream>

int main ()
{
  std::string str;
  std::cout << "Enter your name: ";
  std::getline (std::cin, str);
  std::cout << "Hello " << "Mr. " + str << std::endl;
  std::cout << "Your name starts with " << str[0] << std::endl;

  int born;
  std::cout << "Enter the year of birth: ";
  std::cin >> born;
  std::stringstream converter;
  converter << born;
  std::cout << "Your nickname is: " << str + converter.str() << std::endl;

  unsigned age;
  std::cout << "Enter you age: ";
  std::cin.sync();
  std::getline (std::cin, str);
  std::stringstream(str) >> age;
  std::cout << "Next year you'll be " << ++age << std::endl;
}
```

Результат:

```
Enter your name: John
Hello Mr. John
Your name starts with J
Enter the year of birth: 1990
Your nickname is: John1990
Enter you age: 25
Next year you'll be 26
```

Потоки введення-виведення

Робота з текстовими файлами

```
#include <iostream>
#include <string>
#include <fstream>

int main ()
{
  std::string s = "string";
  int i = 10;

  std::ofstream outfile ("temp.txt");
  if (outfile.is_open())
  {
    outfile << s << std::endl << i;
    outfile.close();
  }
  else std::cout << "Unable to open file";

  std::ifstream infile ("temp.txt");
  if (infile.is_open())
  {
    while (std::getline (infile, s))
    {
      std::cout << s << std::endl;
    }
    infile.close();
  }
  else std::cout << "Unable to open file";
}
```

Результат:

```
string
10
```

Збірка програми C++

Препроцесор, компілятор, компонувальник

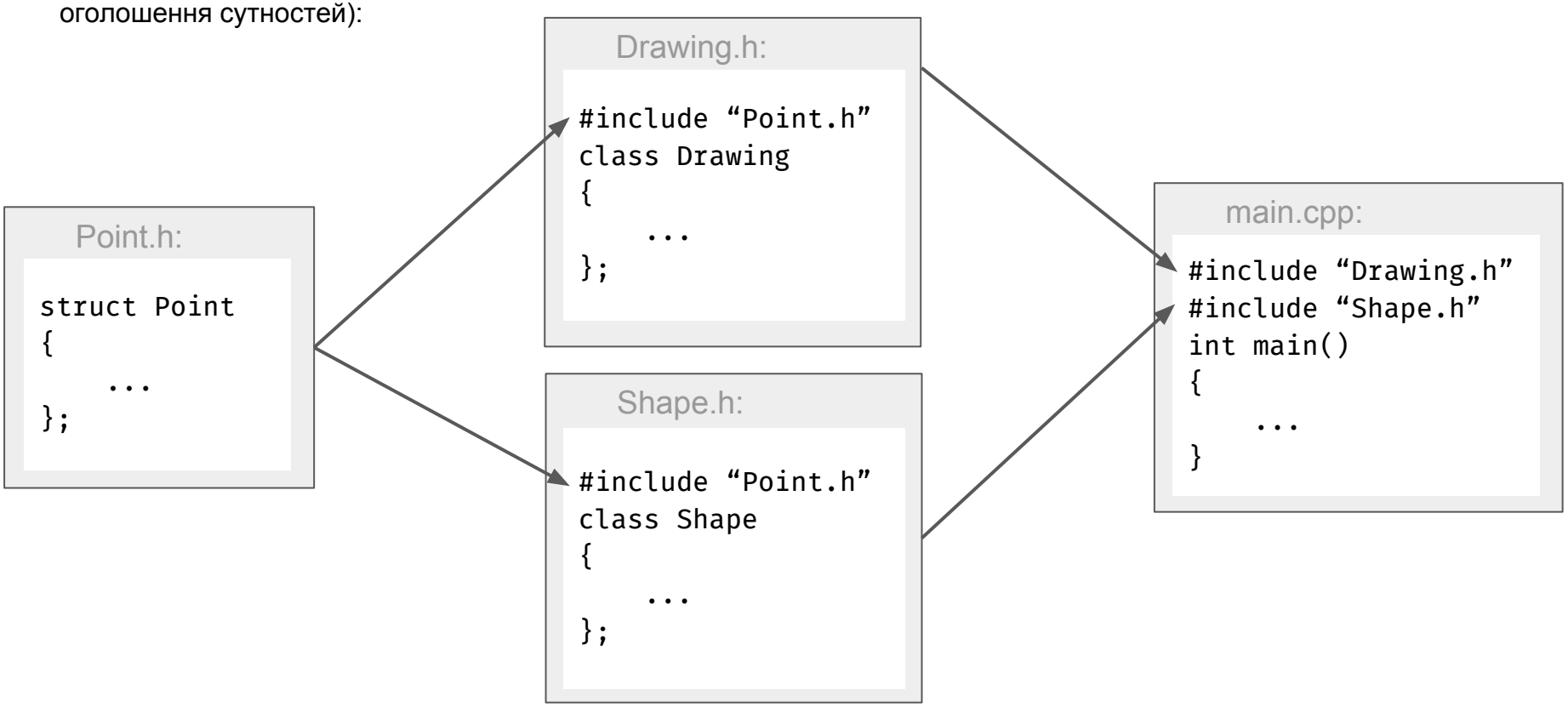
cl.exe *.cpp (компілятор MS VS)
g++ *.cpp (компілятор GCC)

- **Препроцесор** розпізнає #-директиви у вихідних файлах та виконує прості дії, пов'язані з автоматичним редагуванням вихідного коду:
  - **`#include`** - включити в дане місце текст файлу (наприклад, заголовочного файлу з оголошеннями певних сутностей, необхідними компілятору), попередньо виконавши усі директиви в ньому.
  - **`#define`** - визначити макро-підстановку, і т.д.
- **Компілятор** перекладає код вихідних файлів (після препроцесора) в код, близький до машинного: одиниці компіляції (*.**cpp**-файли) з високорівневою реалізацією перетворюються на об'єктні модулі, придатні для остаточного збирання у виконуваний файл.
  - На цьому етапі відслідковуються синтаксичні помилки в коді.
  - Проста ідея в основі багатьох конструкцій та шаблонів: винести якомога більше можливих помилок для автоматичного виявлення компілятором. Так простіше їх усунути, аніж на етапі виконання програми (відлагодження).
- **Компонувальник** збирає виконуваний файл з "частково" скомпільованих об'єктних модулів (*.**o**-файлів).
  - На цьому етапі часто виявляються помилки типу "нерозв'язаних зовнішніх посилань" (**unresolved externals**), коли компонувальник в жодному з файлів не може знайти реалізації певних процедур або методів. Точка входу в програму - **`int main().`**
- Існує чимало засобів автоматизації збірки програм, як у складі інтегрованих середовищ розробки (англ. **IDE**), так і окремих інструментів (**Make**)

Збірка програми C++

Заголовочні файли. Уникнення їх повторного включення

- Компілятор може скомпілювати одиницю компіляції в об’єктний файл, навіть якщо для певних сутностей не знайде реалізації, але матиме їх оголошення.
- Заголовочні файли призначені для опису оголошень сутностей, що використовуються у багатьох одиницях компіляції. Заголовки не повинні містити коду реалізації процедур або методів без особливої на те потреби.
- Однакові заголовки можуть бути включені кілька разів, таким чином призводячи до помилок компіляції (повторне оголошення сутностей):



- Використовуйте **`#ifndef ... #define … #ednif`**
- або **`#pramga once`** для недопущення такої ситуації

Клас (коректне оформлення)

Інтерфейс + реалізація + клієнт-споживач

Drawing.h:

```
#ifndef DRAWING_H
#define  DRAWING_H

class Drawing
{
public:
  Drawing();
  ~Drawing();
  void Clear();
};

#endif // DRAWING_H
```

Drawing.cpp:

```
#include <iostream>
#include "Drawing.h"

Drawing::Drawing()
{
  std::cout << "Drawing created"
            << std::endl;
}

Drawing::~Drawing()
{
  std::cout << "Drawing destroyed"
            << std::endl;
}

void Drawing::Clear()
{
  std::cout << "Drawing cleared"
            << std::endl;
}
```

main.cpp

```
#include "Drawing.h"

int main()
{
  Drawing drawing;
  drawing.Clear();
}
```

Результат:

```
Drawing created
Drawing cleared
Drawing destroyed
```

Інкапсуляція

Поля класу. Методи-селектори, методи-модифікатори

Інкапсуляція:

- заключення даних (стану) сутності всередині класу (як полів класу)
- обмеження доступу до полів виключно через відкритий інтерфейс (методи класу) для забезпечення несуперечливості стану об’єктів.

```
class Drawing
{
public:
  Drawing();
  ~Drawing();
  void Clear();
  void SetName
       (const std::string & name);
  std::string GetName() const;
  void SetSize(float w, float h);
  float GetWidth() const;
  float GetHeight() const;

private:
  float width, height;
  std::string name;
};
```

```
void Drawing::SetName(const std::string & newName)
{
  name = newName;
}
std::string Drawing::GetName() const
{
  return name;
}
void Drawing::SetSize
    (float newWidth, float newHeight)
{
  width = newWidth;
  height = newHeight;
}
float Drawing::GetWidth() const
{
  return width;
}
float Drawing::GetHeight() const
{
  return height;
}
```

Перевантажені конструктори

Списки ініціалізації - використовуйте їх, де тільки можна

Перевантаження:

- повторне використання імені процедури або методу, але з іншою реалізацією для іншого набору аргументів

```
class Drawing
{
public:
  Drawing(float initWidth = 100.,
          float initHeight = 100.);
  Drawing(const Drawing & orig);
  ~Drawing();
  void Clear();
  std::string & Name();
  void SetSize
     (float newWidth,
      float newHeight);
  float GetWidth() const;
  float GetHeight() const;

private:
  float width, height;
  std::string name;
};
```

```
Drawing::Drawing(float initWidth,
                 float initHeight)
  : width(initWidth), height(initHeight),
    name("drawing")
{
  std::cout << "Drawing created "
            << width << " x " << height
            << std::endl;
}

Drawing::Drawing(const Drawing & orig)
  : width(orig.width), height(orig.height),
    name(orig.name)
{
  std::cout << "Drawing created as copy"
            << std::endl;
}


// замість “геттера” та “сеттера”.
// Порушує інкапсуляцію
std::string & Drawing::Name()
{
  return name;
}
```

Перевантажені оператори

Використовуйте, де це доречно!

- Дія операторів може бути перевизначена через перевантаження відповідних процедур, функцій або методів.

- Не перевантажуються: `.` `.*` `::` `?:` `sizeof` `typeid`

```
struct Point
{
  float x, y;
  Point(float x = 0, float y = 0);
  Point & operator++ ();
  Point operator++ (int);
  Point & operator+= (const Point & p);
  float & operator[](unsigned idx);
  const float & operator[](unsigned idx) const;
};

std::ostream& operator<<(std::ostream& os, const Point & p);
bool operator== (const Point & a, const Point & b);
bool operator!= (const Point & a, const Point & b);
bool operator< (const Point & a, const Point & b);
bool operator>= (const Point & a, const Point & b);
Point operator+ (Point a, const Point & b);
```

Перевантажені оператори

Кращі практики перевантаження

```
std::ostream& operator<<(std::ostream& os, const Point & p)
{
  os << "(" << p.x << "," << p.y << ")";
  return os;
}

bool operator== (const Point & a, const Point & b)
{
  return (a.x == b.x) && (a.y == b.y);
}

bool operator!= (const Point & a, const Point & b)
{
  return !(a == b);
}

bool operator< (const Point & a, const Point & b)
{
  return (a.x * a.x + a.y * a.y) < (b.x * b.x + b.y * b.y);
}

bool operator>= (const Point & a, const Point & b)
{
  return !(a < b);
}
```

- Деякі оператори доцільніше перевантажувати як методи, інші - як функції
- Завжди перевантажуйте логічно повний набір операцій певної групи: логічні, арифметичні тощо.
- Уникайте повторення коду, реалізуйте одні оператори через інші

Перевантажені оператори

Кращі практики перевантаження

```
Point & Point::operator++ ()
{
  x++;
  y++;
  return *this;
}

Point & Point::operator +=
  (const Point & p)
{
  x += p.x;
  y += p.y;
  return *this;
}

const float & Point::operator[](unsigned idx) const
{
  return idx ? y : x;
}

float & Point::operator[](unsigned idx) // неконстантна форма методу - через приведення типів
{
  return const_cast<float &>(static_cast<const Point &>(*this)[idx]);
}
```

```
Point Point::operator++ (int)
{
  Point tmp(*this);
  operator++();
  return tmp;
}

Point operator+
   (Point a, const Point & b)
{
  a += b;
  return a;
}
```

Перевантажені оператори

Перевіряйте, чи не перевантажено оператори для невідомих Вам класів

```
#include “Point.h”

int main()
{
  Point a, b(1,1);
  a = b;
  std::cout << a << std::endl;
  std::cout << (a == b) << std::endl;
  std::cout << a++ << std::endl;
  std::cout << ++a << std::endl;
  a += b;
  std::cout << (a >= b) << std::endl;
  a[0] = 10;
  std::cout << a[0] << ", " << a[1] << std::endl;
}
```

Результат:

```
(1,1)
1
(1,1)
(3,3)
1
10, 4
```

Стек та динамічна пам’ять

Автоматичне та динамічне створення та знищення об’єктів

```
int main()
{
  Drawing drawing(200);
  std::cout << "name = " << drawing.Name() << std::endl;
  drawing.Name() = "drawing-1";
  std::cout << "name = " << drawing.Name() << std::endl;
  Drawing *drawingPtr = &drawing;
  drawingPtr->Name() = "drawing-2";
  std::cout << "new name = " << drawing.Name() << std::endl;
  drawingPtr = new Drawing(drawing);
  std::cout << "name of new drawing = " << drawingPtr->Name() << std::endl;
  delete drawingPtr;
}
```

Результат:

```
Drawing created 200 x 100
name = drawing
name = drawing-1
new name = drawing-2
Drawing created as copy
name of new drawing = drawing-2
Drawing destroyed
Drawing destroyed
```

Статичні члени

Фактично - глобальні дані в класі

- **Статичні члени** існують в єдиному екземплярі без прив'язки до окремих об'єктів класу, вони не впливають на розмір об'єктів, для роботи з ними не потрібно створювати об'єкти.

```
class Drawing
{
public:
  Drawing(float initWidth = 100.,
          float initHeight = 100.);
  Drawing(const Drawing & orig);
  ~Drawing();
  int GetId() const;
  ...
private:
  static int counter;
  int id;
  ...
};
```

```
int Drawing::counter = 0;
Drawing::Drawing(float initWidth, float initHeight)
  : width(initWidth), height(initHeight),
    name("unnamed"), id(++counter)
{
  std::cout << "Drawing " << id << " created "
      << width << " x " << height << std::endl;
}
Drawing::Drawing(const Drawing & orig)
  : width(orig.width), height(orig.height), name
(orig.name), id(++counter)
{
  std::cout << "Drawing " << id
        << " created as copy" << std::endl;
}
Drawing::~Drawing()
{
  std::cout << "Drawing " << id
            << " destroyed" << std::endl;
}
int Drawing::GetId() const
{
  return id;
}
```

Статичні члени

Фактично - глобальні дані в класі

```
int main()
{
  Drawing drawing(200);
  std::cout << "name of #" << drawing.GetId()
            << " = " << drawing.Name() << std::endl;
  drawing.Name() = "drawing-1";
  std::cout << "name of #" << drawing.GetId()
            << " = " << drawing.Name() << std::endl;
  Drawing *drawingPtr = &drawing;
  drawingPtr->Name() = "drawing-2";
  std::cout << "name of #" << drawing.GetId()
            << " = " << drawing.Name() << std::endl;
  std::cout << "name of #" << drawingPtr->GetId()
            << " by pointer = " << drawing.Name() << std::endl;
  drawingPtr = new Drawing(drawing);
  std::cout << "name of #" << drawing.GetId()
            << " = " << drawing.Name() << std::endl;
  std::cout << "name of #" << drawingPtr->GetId()
            << " by pointer = " << drawing.Name() << std::endl;
  delete drawingPtr;
}
```

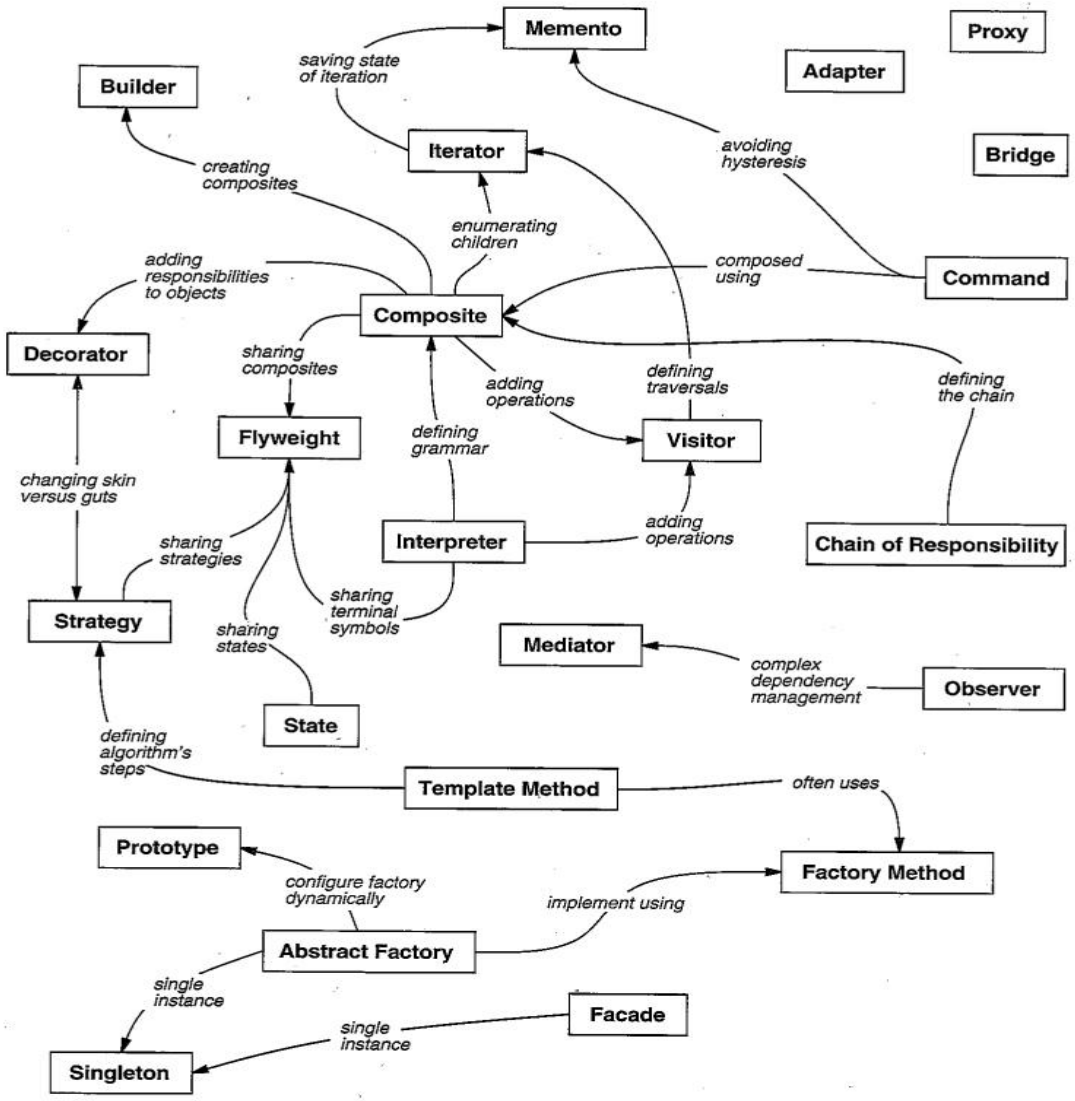
Результат:

```
Drawing 1 created 200 x 100
name of #1 = unnamed
name of #1 = drawing-1
name of #1 = drawing-2
name of #1 by pointer = drawing-2
Drawing 2 created as copy
name of #1 = drawing-2
name of #2 by pointer = drawing-2
Drawing 2 destroyed
Drawing 1 destroyed
```

Шаблони проектування

Визнані практики у проектуванні об’єктно-орієнтованої архітектури програми

- Шаблони проектування - узагальнені рішення окремих поширених задач розробки архітектури програмного забезпечення. Це не завершені фрагменти коду на конкретній мові програмування, а, скоріше, підходи до розв’язання типових задач.
- Основна робота - книга Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software” (1994)
- У книзі описані 23 шаблона проектування. Команда авторів цієї книги відома суспільству під назвою “Банда чотирьох” (англ. Gang of Four - GoF).



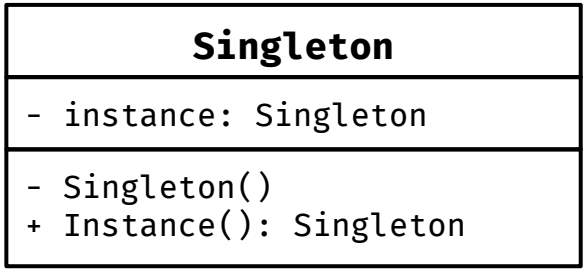
Design Pattern Relationships

Шаблони проектування

Патерн “Одинак” (Singleton)

- “Поліпшений” глобальний об’єкт. Надає доступ до гарантовано **єдиного** екземпляру класу.
- Проста глобальна змінна не забороняє створення копій. “Одинак” **контролює** свою єдиність.
- **На замітку**: глобальні змінні та об’єкти ускладнюють розвиток та підтримку програми (важко відстежити доступ до таких сутностей, вони привносять занадто багато залежностей в код). Вживайте цей шаблон виважено.
- Приклад реалізації за Мейерсом (для C++11 та вище - безпечний і при багатопоточному виконанні)

```
class Singleton
{
public:

    static const Singleton & Instance()
    {
        // статична змінна не створюється до першої вимоги

        static Singleton singleInstance;
        return singleInstance;
    }

private:

    // заборона створювати інші копії одинака

    Singleton(){}
    Singleton(const Singleton & sample);
    Singleton & operator=(const Singleton & sample);
};
```



- На замітку: реалізовуючи специфічну поведінку класів при створенні та копіюванні, визначайте конструктор за замовчанням, конструктор копіювання, оператор присвоювання та деструктор.

Шаблони проектування

Патерн “Одинак” (Singleton). Приклад застосування - головний клас програми

```
#include “Drawing.h”

class Editor
{
public:
    static const Editor & Instance()
    {
        static Editor editor;
        return editor;
    }
    inline Drawing & getDrawing()
    {
        return drawing;
    }
    ...
private:
    Drawing drawing;
    Editor(){}
    Editor(const Editor & sample);
    Editor & operator=(const Editor & sample);
};

int main ()
{
    Editor editor; // помилка компіляції!
    Drawing & drawing = Editor::Instance().getDrawing();
}
```

- **На замітку**: вбудовані (**inline**) методи були покликані покращити швидкодію програм. Код методу не викликається як процедура, а вставляється в кожне місце виклику.
- Прості методи-селектори (“геттери”) є гарними кандидатами для вбудовування.
- Це може зібльшити розмір програми, тож не є доцільним для складних методів.
- Якщо реалізація методу вказана прямо у визначенні класу, то метод вважається inline за замовчанням.
- В будь-яком випадку рішення, чи вбудовувати метод, приймає **компілятор**. Підказувати йому нині немає сенсу.