Problem

Korzystając z przedstawionej na wykładzie infrastruktury służącej do odczytywania figur, chcielibyśmy doprowadzić sprawę do końca, czyli wyświetlenia figur w oknie.
Na początek pominiemy kwestię skalowania mapy – w dostarczonym pliku testowym współrzędne punktów figur są zadane w pikselowym układzie współrzędnych okna.
Sprawą, która przysporzy najwięcej pracy będzie reorganizacja (*refaktoring*) kodu: zastąpienie klasy `figure` „od wszystkiego” hierarchią klas, w której każdy rodzaj figury będzie mieć odpowiednią klasę, a `figure` będzie sprowadzone do bazy (abstrakcyjnej) całej hierarchii.
Sporo radości powinno też dostarczyć utworzenie projektu, w którym pliki nagłówkowe i biblioteki są rozrzucone po licznych folderach.

Projekt

Na ostatnich ćwiczeniach nie wystarczyło czasu na wykonanie ostatniego kroku – zapisania szablonu projektu. Zrobimy to teraz, zgodnie z instrukcją z poprzednich ćwiczeń.

Teraz pora utworzyć projekt na podstawie zapisanego właśnie szablonu.

Zaczynam standardowo: `New > Project`, ale w oknie `New from template` wybieram `User templates` i dalej `graph_lib_project`. Z wyborem foldera projektu trzeba uważać – potrzebujemy nowy, pusty folder, którego jeszcze nie ma! Tworzę go w oknie wyboru foldera i upewniam się, że nazwę nowego foldera (u mnie `Zad_1`) widać w polu `Folder`. `Code::Blocks` proponuje jeszcze zmianę nazwy projektu. Skrzętnie z tego korzystam i zmieniam ją na `zad_1`.

Projekt

Po utworzeniu projektu zmieniam natychmiast nazwę pliku `tst_graph.cpp` na `figure_test.cpp`. W drugim kroku kopiuję `figure_test.cpp`, `figure.cpp`, `figure.h` i `mapa_test.txt` z archiwum ćwiczeń do foldera projektu.
Dodajemy kod źródłowy (pliki .cpp i .h) do projektu poleceniem `Project > Add files`).
W plikach `figure.h` i `figure.cpp`. mamy implementację metod przedstawionych na ostatnim wykładzie – odpadnie nam mozolne kopiowanie kodu. Trzeba jednak będzie nieco wysiłku umysłowego, żeby zorganizować hierarchię klas, która pozwoli nam na wyświetlenie na koniec ćwiczeń (no, może przed następnymi ćwiczeniami) na zobaczenie w oknie i prostokąta, i trójkąta.

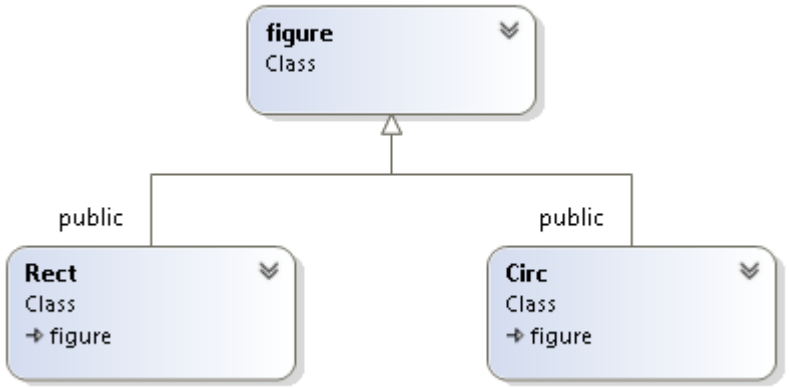
Klasa `figure`

W bieżącej wersji klasy `figure` nie mamy jeszcze niczego, co rozróżniałoby typy figur (jest identyfikator, ale na razie w ogóle nie sprawdzamy jego wartości). Gdyby pozostać przy jednej klasie za moment zaczną pojawiać się w niej instrukcje warunkowe `if`, które w miarę rozbudowywania klasy o kolejne konkretne figury będą się niepokojąco rozrastać i stanowić potencjalne źródło trudnych do wychwycenia i zdiagnozowania błędów.

Rozwiązaniem, które w tej sytuacji (i wielu podobnych) trzeba przynajmniej rozważyć, jest zastąpienie jednej, monolitycznej klasy obejmującej wszystkie figury, hierarchią klas, z których każda będzie odpowiadać za obsługę tylko jednego rodzaju figury.

Hierarchia figur

Sprawa wydaje się dość oczywista:




```
class figure {
    // implementacja
};
class Rect : public figure {
    // implementacja
};
class Circ : public figure {
    // implementacja
};
```

Problem z użyciem klas

W tej chwili mamy jedno krytyczne miejsce, w którym w jednej instrukcji spotykają się – mało tego **powinny być tworzone** – różne figury:

```
figure fig;
while (ifs >> fig)
{
   cout << fig << endl;
}
```

Takie rozwiązanie jest nie do utrzymania przy hierarchii klas: nie możemy utworzyć sobie obiektu klasy `figure`, po czym stwierdzić: od teraz będziesz prostokątem, bo na to wskazuje identyfikator. Musimy od razu wywołać właściwy konstruktor!

Przekształcenie pętli odczytu

Ponieważ liczba figur nie jest znana z góry, jedyną opcją jest tworzenie ich na stercie (za pomocą operatora `new`). W naturalny sposób dostaniemy wskazanie na nowoutworzony obiekt i taką wartość powinna mieć funkcja odczytu figury ze strumienia:

```
figure *fig;
while ((fig = get_figure(ifs)) != nullptr)
{
    // zapamiętać wskazanie w wektorze?
}
```

Zapis jest mniej wygodny, niż z operatorem `>>`, ale odpowiednio napisana funkcja `get_figure` będzie czytać ze strumienia opis figury i tworzyć obiekt właściwej dla rodzaju figury klasy.

Funkcja fabryczna

```
figure* get_figure(istream& is)
{
   string id = get_id(is);
   if (id.length() == 0)
      return nullptr;

   vector<FPoint> pts = get_points(is);

   if (id == Rect::class_id())
      return new Rect(pts);
   if (id == Circ::class_id())
      return new Circ(pts);

   throw std::runtime_error("Unknown figure id.");
}
```

Inne składowe `figure`

Konstruktor i destruktor w klasie bazowej:

```
figure(const std::vector<FPoint>& fv) : fdef(fv) {}
virtual ~figure() {}
```

Konstruktor w pochodnej:

```
Rect(const std::vector<FPoint>& fv) : figure(fv)
{
    if (fdef.size() != 2)
        throw std::runtime_error("Rect: … points.");
}
```

Jaki musi być dostęp do `fdef`, żeby to kompilowało się?

Wreszcie:

```
virtual std::pair<FPoint, FPoint> bbox() const
// ...
```

Figura przedstawia się

W klasie `figure`:

```
static std::string class_id()
{
    return "Unknown";
}
virtual std::string get_id() const = 0;
```

… i w pochodnych (tu `Rect`):

```
static std::string class_id()
{
    return "Rect";
}
virtual std::string get_id() const
{
    return Rect::class_id();
}
```

Figura do wyświetlenia

Do kompletu potrzebne nam są jeszcze obiekty klas pochodnych po Shape – trzeba coś w końcu wyświetlić.
W klasie bazowej mamy:

```
virtual Graph_lib::Shape* get_shape() const = 0;
```

… a w Rect:

```
Graph_lib::Shape* get_shape() const
{
    return new Graph_lib::Rectangle(fdef[0], fdef[1]);
}
```

Tu jest problem, bo Rectangle potrzebuje argumentów typu Graph_lib::Point, a dajemy typ FPoint. W FPoint należy dodać operator konwersji do Point:

```
operator Graph_lib::Point() const {
  return Graph_lib::Point(int(this->x), int(this->y)); }
```

Refaktoring

Refaktoring kodu można podsumować w następujących krokach:

1. Implementacja funkcji `get_figure`, która ma zastąpić operator >> z klasy `figure`.
2. Implementacja klas `Rect` i `Circ` w minimalnej postaci (tzn. tylko z konstruktorem i statyczną `class_id`), co wymaga zmian także w klasie bazowej.
3. Poprawki operatora << w klasie `figure` (dołożenie wirtualnej funkcji `get_id` – czystej w klasie bazowej i zaimplementowanej w pochodnych). Nie mam już składowej `id` w `figure`!
4. Zmiana `bbox` na metodę wirtualną (+ przeniesienie fragmentu kodu do `bbox` w klasie `Circ`)

Nie trzeba chyba wspominać, że w `main` powieliłem sobie „echo” figur na konsolę, żeby na bieżąco śledzić, czy figury odczytują się właściwie.

Wyświetlanie

Jestem gotowy od usunięcia komentarza części wyświetlającej figury w oknie. Od razu trzeba mi zdefiniować w `figure`:

```
virtual Graph_lib::Shape* get_shape() const = 0;
```

Implementacje w `Rect` i `Circ` będą wołać konstruktory pochodnych po `Shape`, odpowiednio `Rectangle` i `Circle`. Zanim się wezmę za to, muszę poradzić sobie z masą błędów, które pojawiły się, kiedy włączyłem (`#include`) na początku `figure.h` plik `graph.h`. Powodem jest makrodefinicja z pliku `std_lib_facilities.h`:

```
#define vector Vector
```

Jest bardzo szkodliwa i należy ją wyłączyć (zakomentować):

```
//#define vector Vector
```

Po tym ruchu mogę wrócić do implementacji `get_shape`.

Wyświetlanie

Oba przypadki wymagają nieco pracy. Do `Rectangle` potrzebuję struktur `Graph_lib::Point` , bo takie dwa punkty są argumentami konstruktora. Najłatwiej będzie mi zdefiniować operator konwersji w `FPoint`:

```
operator Graph_lib::Point() const
```

W konstruktorze `Circle` drugi parametr jest promieniem koła, więc dokładam sobie funkcję:

```
float distance(const FPoint & lf, const FPoint & rt);
```

Zakładam, że przyda mi się dokładna (`float`) odległość między punktami; w wywołaniu konstruktora `Circle` dołożę konwersję wyniku tej funkcji do `int`.

W tym momencie kompilacja kończy się sukcesem i można się spodziewać wyświetlenia figur w oknie programu.

Oddawanie

Definicje wszystkich (czterech) klas i funkcji pomocniczych proszę umieścić w pliku `figure.h`. Implementacje metod klas (o ile są dłuższe, niż jedna instrukcja) oraz funkcji pomocniczych proszę zamieścić w pliku `figure.cpp`.

Rozwiązania (`figure.h` i `figure.cpp`) należy złożyć w `Moodle` do:

26 marca 2023 23:59