Обектно-ориентирано програмиране

Въведение

Иван Георгиев, Христо Иванов, Христо Стефанов

Технологично училище "Електронни системи",
Технически университет, София

18 юни 2019 г.

Съдържание

Съдържание

Сложност на програмни системи

- Сложността е присъща на по-голямата част от програмните продукти, които се разработват в софтуерната индустрия
- Сред основните причини за сложността на програмните системи са:
  - Сложността на предметната област, която програмните системи се стараят да описват (моделират)
  - Размерът (напр. в брой редове код) на програмните продукти
  - Нуждата от развитие на програмните системи - да могат сравнително лесно да бъдат разширявани

Сложност на програмни системи

```c
#include <stdio.h>
int main() {
    char screen[20][20] = { 0 };
    int i;
    int j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            screen[i][j] = ' ';
        }
    }
    for (i = 5; i < 15; i++) {
        screen[i][5] = '-';
    }
    for (i = 5; i < 15; i++) {
        screen[i][15] = '+';
    }
    for (i = 5; i < 15; i++) {
        screen[5][i] = '#';
    }
    for (i = 5; i < 15; i++) {
        screen[15][i] = '%';
    }
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            printf("%c", screen[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Фрагмент 1: Пример за сложна програма, която изчертава квадрат

Декомпозиция

- "Начинът за управление на сложни системи е известен още от древността — разделяй и владей (divide et impera)"
  (Dijkstra, E., 1979, "Programming Considered as a Human Activity")
- Прилагането на този подход в една програмна система води до нейното *декомпозиране* на отделни части наречени *модули (или компоненти)*
- Отделните модули трябва да могат да се развиват самостоятелно и независимо един от друг
- Връзките между модулите трябва да са слаби. Те трябва да се осъществяват чрез специално дефинирани интерфейси. Интерфейсите най-често представляват конкретен набор от функции

Капсулация

- При разделянето на една програма на модули с интерфейси най-често се следва принципа за *капсулация*
- Капсулацията е принцип, съгласно който всеки модул на програмата трябва да предоставя подходящ интерфейс за работа и да капсулира (скрива) вътрешното си устройство от другите модули в програмата (*encapsulation*, *information hiding*)
- Следването на този принцип води до лесното използване на модулите, тъй като програмиста не е нужно да разбира в детайли как работи модула за да може да го използва
- Пример от реалният свят за принципа на капсулацията е телевизор с дистанционно
  - Дистанционното предоставя бутонен интерфейс, чрез който се управлява трудна за разбиране електроника
  - Не е нужно потребителят да разбира от електроника за да използва интерфейса
  - Предоставения бутонен интерфейс е пълен и достатъчен за работа с телевизора

Капсулация

- За да бъде спазен принципа на капсулация, всякаква комуникация/взаимодействие между модулите трябва да се случва само чрез дефинираните интерфейси на модулите
- Неспазването на това условие може да доведе до проблеми в грешно достъпвания модул, тъй като по време на неговата имплементация такъв достъп не е бил предвиден
- Аналогия за това е да се отвори кутията на телевизор и да се започне пипането на електрически компоненти за да бъде сменен текущия канал, вместо да се използва дистанционното управление
- Ако липсва функционалност, която не е достъпна чрез интерфейса на модула, по-правилното решение е тази функционалност да се добави в модула, отколкото да се опитва по заобиколен начин да се взаимодейства с него
  - Причината за това е, че ако вътрешното устройство на модула се промени (аналогия със закупен нов модел телевизор), трябва да се измисля нов заобиколен начин, което може и да не е възможно

Декомпозиция

```c
#include <stdio.h>
char screen[20][20];
void screen_init() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            screen[i][j] = ' ';
        }
    }
}
void screen_draw_vline(
int x, int y, int len, char symb)
{
    int i;
    for (i = y; i < y + len; i++) {
        screen[i][x] = symb;
    }
}
```

```c
void screen_draw_hline(
int x, int y, int len, char symb)
{
    int i;
    for (i = x; i < x + len; i++) {
        screen[y][i] = symb;
    }
}
void screen_print() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            printf("%c", screen[i][j]);
        }
        printf("\n");
    }
}
```

Фрагмент 2: Декомпозирана програма изчертаваща квадрат. Модул за екран. (1/2)

Декомпозиция

```
int main() {
    screen_init();
    screen_draw_vline(5, 5, 10, '-');
    screen_draw_hline(5, 5, 10, '#');
    screen_draw_vline(15, 5, 10, '+');
    screen_draw_hline(5, 15, 10, '%');
    screen_print();
    return 0;
}
```

Фрагмент 3: Декомпозирана програма изчертаваща квадрат. Модул за изчертаване на квадрат. (2/2)

Съдържание

Обектно-ориентирано програмиране

- Обектно-ориентираното програмиране представлява парадигма (подход/начин) на програмиране, който използва систематичното прилагане на декомпозиция чрез принципа за капсулация[1]
- Обектно-ориентираното програмиране може да се прилага независимо от езика, с който се работи. Разликата между отделните езици за програмиране е колко добре и колко лесно могат да се приложат идеите на обектно-ориентираното програмиране
- Обектно-ориентиран език се нарича език, който има специална поддръжка за обектно-ориентирано програмиране

[1] Обектно-ориентираното програмиране включва и други идеи и концепции, но за целите на въведението са подбрани най-основните

Обектно-ориентирано програмиране

- Първият обектно-ориентиран език е SIMULA I, разработен в периода 1962–1965 от Ole-Johan Dahl и Kristen Nygaard. Малко по-късно се появява и SIMULA 67 (1967)
- В езикът SIMULA 67 са въведени повечето от ключовите концепции на обектно-ориентираното програмиране — обекти, класове, наследяване, полиморфизъм
- През 90 години на XX век обектно-ориентираното програмиране се превръща в доминираща софтуерна технология

Обектно-ориентирано програмиране

- В обектно-ориентираното програмиране модулите, получени чрез декомпозиране и принципа за капсулация, се наричат *обекти*, а отделните функции от интерфейса на модула - *методи*
- Комуникацията между обектите се извършва чрез *съобщения*, които представляват заявки за изпълнение на даден метод
- Целта на обектно-ориентираното програмиране е да се представи цялата програма под формата на обекти, които изпращат съобщения един на друг
- По-конкретните дефиниции на термините са:
  - *Обект* — обединение между данни и функциите, предназначени за обработването на тези данни
  - *Метод* — предоставя услуга, характерна за даден обект
  - *Съобщение* — заявка за изпълнение на даден метод

Съдържание

Проблем

- При писането на програма, използвайки обектно-ориентирано програмиране, много често се стига до писане на обекти, които са подобни
- Например в програмата за чертане на квадрат, ако искаме да използваме два “екрана” едновременно, трябва да създадем втори обект, който като код е почти еднакъв - единствената разлика е, че функциите трябва да работят върху друга глобална променлива от същия тип

Проблем

```c
#include <stdio.h>
char screen2[20][20];
void screen2_init() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            screen2[i][j] = ' ';
        }
    }
}
void screen2_draw_vline(
int x, int y, int len, char symb)
{
    int i;
    for (i = y; i < y + len; i++) {
        screen2[i][x] = symb;
    }
}
```

```c
void screen2_draw_hline(
int x, int y, int len, char symb)
{
    int i;
    for (i = x; i < x + len; i++) {
        screen2[y][i] = symb;
    }
}
void screen2_print() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            printf("%c", screen2[i][j]);
        }
        printf("\n");
    }
}
```

Фрагмент 4: Прогр. чертаеща квадрат на два екрана едновременно. Обект за екран 2. (1/3)

Проблем

```
//
char screen[20][20];
void screen_init() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            screen[i][j] = ' ';
        }
    }
}
void screen_draw_vline(
int x, int y, int len, char symb)
{
    int i;
    for (i = y; i < y + len; i++) {
        screen[i][x] = symb;
    }
}
```

```
void screen_draw_hline(
int x, int y, int len, char symb)
{
    int i;
    for (i = x; i < x + len; i++) {
        screen[y][i] = symb;
    }
}
void screen_print() {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            printf("%c", screen[i][j]);
        }
        printf("\n");
    }
}
```

Фрагмент 5: Прогр. чертаеща квадрат на два екрана едновременно. Обект за екран 1. (2/3)

Проблем

```
int main() {
    screen_init();
    screen2_init();
    screen_draw_vline(5, 5, 10, '-');
    screen2_draw_vline(5, 5, 10, '-');
    screen_draw_hline(5, 5, 10, '#');
    screen2_draw_hline(5, 5, 10, '#');
    screen_draw_vline(15, 5, 10, '+');
    screen2_draw_vline(15, 5, 10, '+');
    screen_draw_hline(5, 15, 10, '%');
    screen2_draw_hline(5, 15, 10, '%');
    screen_print();
    screen2_print();
    return 0;
}
```

- Този фрагмент може да се разглежда като обект с един метод `main`

Фрагмент 6: Програма чертаеща квадрат на два екрана едновременно. Обект за изчертаване на квадрат на два екрана едновременно (3/3)

Класове и инстанции

- Тъй като единствената разлика между подобните обекти е върху кои конкретни променливи работят методите, се заражда идеята за създаване на шаблон, по който да се правят обекти - наречен още *клас*
- Класът съдържа описание на данните и методите нужни за направата на един обект
- В рамките на един клас методите се пишат не спрямо конкретни променливи от програмата, а спрямо описанието на данните в класа (шаблона)
- Обект, който е създаден чрез използване на клас (шаблон), още се казва, че е *инстанция на класа*

Класове и инстанции

```c
#include <stdio.h>
struct Screen {
    char screen[20][20];
};
void screen_init(struct Screen* this){
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            this->screen[i][j] = ' ';
        }
    }
}
void screen_draw_vline(struct Screen* this,
int x, int y, int len, char symb)
{
    int i;
    for (i = y; i < y + len; i++) {
        this->screen[i][x] = symb;
    }
}
```

```c
void screen_draw_hline(struct Screen* this
int x, int y, int len, char symb)
{
    int i;
    for (i = x; i < x + len; i++) {
        this->screen[y][i] = symb;
    }
}
void screen_print(struct Screen* this) {
    int i, j;
    for (i = 0; i < 20; i++) {
        for (j = 0; j < 20; j++) {
            printf("%c", this->screen[i][j]);
        }
        printf("\n");
    }
}
```

Фрагмент 7: Клас (шаблон) за направата на обект за екран в C. (1/2)

Класове и инстанции

```
struct Screen screen1;// create variables holding the data for
struct Screen screen2;// the first and second Screen objects (instances)
int main() {
// the following code calls the _same_ methods but on _different_ objects
    screen_init(&screen1);
    screen_init(&screen2);
    screen_draw_vline(&screen1, 5, 5, 10, '-');
    screen_draw_vline(&screen2, 5, 5, 10, '-');
    screen_draw_hline(&screen1, 5, 5, 10, '#');
    screen_draw_hline(&screen2, 5, 5, 10, '#');
    screen_draw_vline(&screen1, 15, 5, 10, '+');
    screen_draw_vline(&screen2, 15, 5, 10, '+');
    screen_draw_hline(&screen1, 5, 15, 10, '%');
    screen_draw_hline(&screen2, 5, 15, 10, '%');
    screen_print(&screen1);
    screen_print(&screen2);
    return 0;
}
```

Фрагмент 8: Обект за изчертаване на квадрат без клас (шаблон) в C. (2/2)