Programski jezici

Uvodno predavanje

Igor Jovančević mail: igorjovan@gmail.com

Danas...

- Informacije o organizaciji kursa
- Informacije o polaganju ispita
- Informacije o sadržaju kursa

- Uvodno predavanje
- Uvod u vježbe

Termin za predavanja + vježbe

Petak 12h15 – 16h

Par pravila

- Termin za konsultacije
  - Petkom od 10h do 12h (kancelarija preko puta sale 210)
  - Uz najavu makar dan prije, mail-om

- Mail-u koji se tiče svih **mora** da prethodi vaš dogovor
- Provjeravajte sajt i DL platformu

- Slanje tuđeg rada je strogo zabranjeno i kažnjivo za obje strane

Bodovi za ocjenu

- 40%
  - kolokvijum
- 40%
  - završni ispit
- 20%
  - Domaći/projekat (izvještaj + odbrana)

Domaći/projekat

- Šalje se mail-om:
  1. **JEDAN** pdf dokument (Izvještaj)
  2. **Spakovan** folder sa programskim kodom

- Primjer Izvještaja
- Odbrana na kraju semestra pred profesorom

Struktura kursa

- Predavanja
  - UML: use case dijagrami, dijagrami klasa, dijagrami objekata, dijagrami stanja, dijagrami interakcije (sekvence i kolaboracije), ograničenja (OCL jezik)
  - Implementacione strategije
  - Objektno-orijentisani principi
  - Projektni obrasci
- Vježbe
  - C++ (i Python ako stignemo)
  - Implementacione strategije
    - modelovanje i
    - prevod modela u kod

Uvodno predavanje

UML

- Unifed Modeling Language
- De facto industrijski standard za modelovanje softvera (uglavnom grafički)
- Veliki jezik – mi ćemo raditi samo esencijalni dio
- Nastao kao potreba :
  - standardizacije faze modelovanja softvera
  - motivacija programerima da što više modeluju prije nego što započnu razvoj
- Istorijski trebalo je više godina da se konvergira ka jednom standardu po priznanju tvoraca: *Grady Booch, Ivar Jacobson, James Rumbaugh*

UML

- Nastao krajem 80ih i početkom 90ih kao potreba novonastale objekto-orijentisane paradigme (i jezika C++)

- Omogućava bolji OO dizajn jer omogućava jasno opažanje sistema i njegovih djelova

- Važan u **komunikaciji** između programera i timova (postojećih i novih) ali i sa klijentima (u slučaju dijagrama visokog nivoa apstrakcije – npr Use Case, model domena)

    1. *Prirodni jezik suviše neprecizan i nije lako izraziti kompleksne koncepte*
    2. *Programski kod je precizan ali suviše detaljan i zahtijeva vrijeme za razumijevanje*
    3. *UML je između 1 i 2 -> precizan ali samo za važne detalje*

Dijagrami UML-a

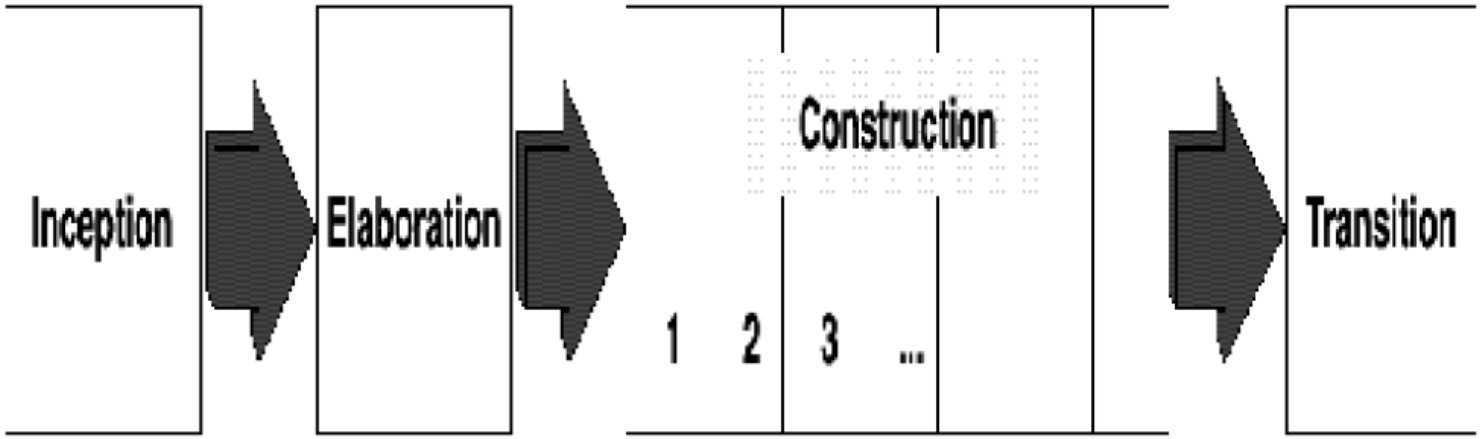
- UML je alat kojim modelujemo i **statičke** i **dinamičke** aspekte sistema
  - Npr. dijagram **paketa** daje uvid u glavne velike komponente sistema...za svaki od paketa možemo napraviti klasni dijagram
  - Npr. **klasni** dijagram nam pruža uvid u statičke aspekte, u strukturu sistema (koncepti OOP)
  - Npr. dijagram **kolaboracije** nam pruža uvid u to kako objekti klasa sarađuju međusobno u jednom momentu
  - Npr. dijagram **interakcije** nam može otkriti da je sistem pre-centralizovan -> sva odgovornost na jednom objektu

U ranim fazama razvoja

- Koristan je u fazi popisivanja korisničkih zahtjeva
- Use Case dijagrami mogu da predstavljaju sliku sistema jasnu i korisnicima i programerima
- Dobar klasni dijagram takođe
  - 1 klasa = 1 koncept u jeziku korisnika (u domenu)
- Dijagrami aktivnosti
  - važni kada kod korisnika postoje radni procesi koji se uvijek žele optimizovati više, obično kroz paralelizam

Proces razvoja softvera

- **Začetak**: identifikacija potrebe i definisanje opsega projekta
- **Elaboracija**: detaljnija specifikacija za analizu visokog nivoa -> prva arhitektura i plan za Razvoj, rizici (tehnološki, ljudski, korisnički, politički)
- **Razvoj**: iterativan i inkrementalan
- Obično se softver ne izdaje odjednom
  - Iteracije eksterno ili interno
- Jedna iteracija: analiza, dizajn, implementacija, testiranje
- **Tranzicija**: beta testiranje, pojačavanje performansi i trening korisnika



Elaboracija

- Početni intervjui sa korisnicima -> Use Case dijagrami

- Konceptualni model domena -> klasni dijagram
- Ako postoji jak aspekt radnog toka -> dijagram aktivnosti (korisni za identifikaciju paralelnih tokova)
- Korisnici aktivno mogu da učestvuju u kreiranju ovih dijagrama
  - (često u ovim fazama se ne postupa rigorozno po pravilima jezika)

- Tehnološki rizici
  - Izrada prototipa, testiranje kompatibilnosti različitih komponenti
  - Definisanje arhitekture sistema (package, deployment dijagrami)

- Na kraju ove faze:
  - moguće dati vremenske procjene za sve use case-ove
  - identifikovani su rizici i planovi za prevazilaženje

Razvoj

- Release plan: definisane funkcionalnosti koje će biti isporučene pri svakoj iteraciji
- Svaka iteracija je mini-projekat
  - procjena vremena za svaku iteraciju: analiza, dizajn, kodiranje, testiranje, integracija

- Inkrementalan proces – postojeći use case-ovi se nadgrađuju novim
- Iterativan proces – u svakoj iteraciji se neki djelovi koda pišu ponovo
- **Svi UML dijagrami su korisni u ovoj fazi**

Tranzicija

- Nema dodavanja funkcionalnosti osim ako nije neophodno
- Optimizacija koja nije urađena tokom razvoja
- Debug

Use case dijagrami
(dijagram slučajeva korišćenja)

Use case dijagrami

- Dijagrami scenarija
- Prikazuju korisnike koji interaguju sa sistemom i slučajeve tog korišćenja bez detalja kako se interakcija odvija
- Važni u ranim fazama (elaboracija)
- Dugo bili zanemareni – neformalno tretirani i nedokumentovani

Scenario

- Scenario je sekvenca ili niz koraka koji opisuju interakciju **između korisnika i sistema**.
- Primjer:
  - sistem: **online prodavnica**
  - scenario: **kupovina proizvoda**
  - sekvenca koraka:
    1. kupac pretražuje katalog proizvoda i željene proizvode dodaje u korpu.
    2. prilikom plaćanja, kupac unosi potrebne informacije o platnoj kartici i potvrđuje kupovinu.
    3. sistem provjerava unijete parametre i prikazuje poruku kojom se obavještava kupac o uspješno obavljenoj kupovini,
    4. nakon čega se generiše i šalje odgovarajuće obavještenje na mejl adresu kupca.

  - ako parametri kartice nisu tačni -> odvojen scenario

Use case

- Obično definišemo jedan primarni „sve-prolazi-dobro“ *(eng. all-goes-well)* scenario i nekoliko alternativa kao varijacije primarnog scenarija
- Use case je skup scenarija koji imaju **zajednički korisnički cilj**
- Use case: **kupovina proizvoda**
  - Jedan scenario: uspješna kupovina
  - Drugi scenario: odbijeno plaćanje i neuspješna kupovina
  - Treći scenario ?

Use case

- Što rizičniji slučaj korišćenja, to više detalja u dijagramu
- Odluka dizajnera da li će treći scenario biti poseban slučaj korišćenja
- Zbog nepostojanja formalnog standarda moguće su razne varijante, npr. dozvoljeno je definisati preduslove koji moraju biti zadovoljeni da bi otpočeo slučaj upotrebe.

Buy a Product

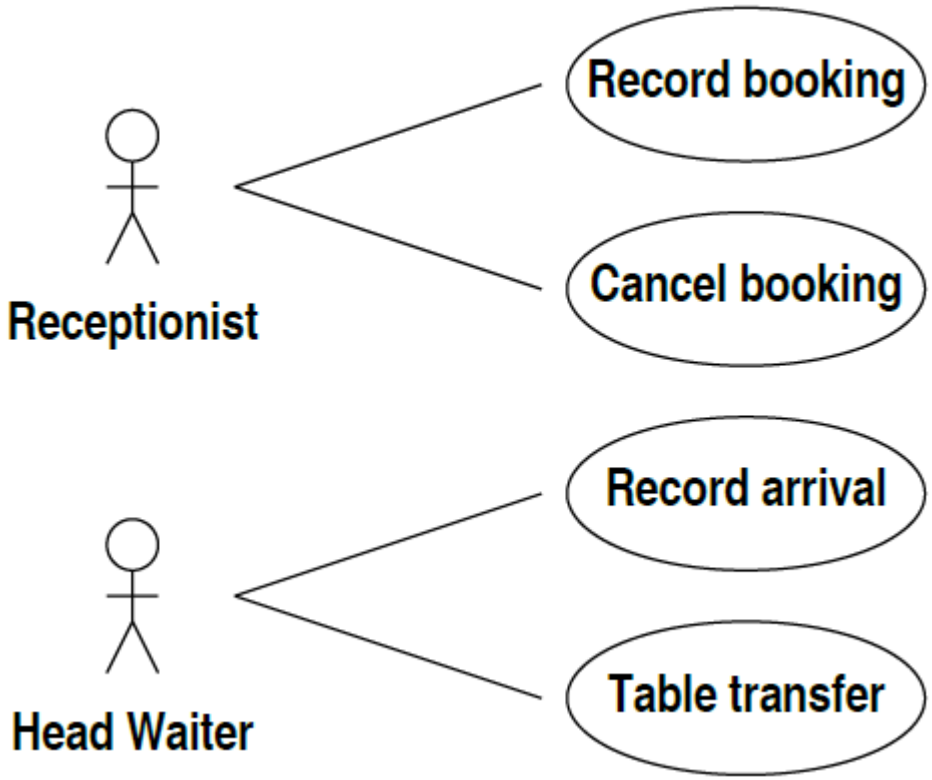
1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

Alternative: *Authorization Failure*
At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: *Regular Customer*
3a. System displays current shipping information, pricing information, and last four digits of credit card information
3b. Customer may accept or override these defaults
Return to primary scenario at step 6

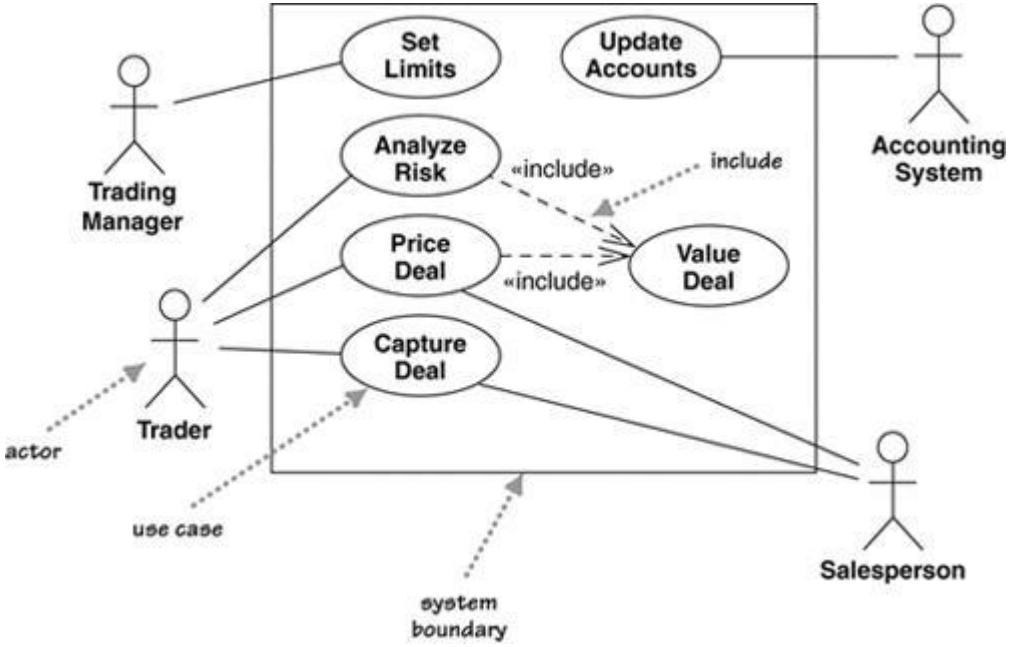
Use case dijagram

- Sumira **aktere** i **slučajeve korišćenja i veze među njima** (koji akter učestvuje u kom slučaju korišćenja)
- Dio UML jezika
- Primjer: restoran
- Simboli
  - veza (asocijacija) – puna linija
  - slučaj korišćenja - elipsa
  - akter – simbol za čovjeka



Use case dijagram

- Primjer: finansijski sistem za trgovinu
  - razumjeti za sada **aktere** i **veze** između aktera i **slučajeva korišćenja**



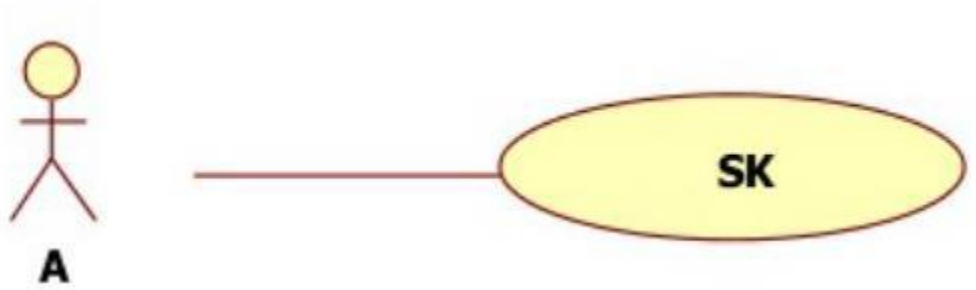
Akteri *(eng. Actors)*

- Akter je **uloga** (role) koju korisnik ostvaruje gledano sa strane sistema
  - 4 aktera u našem primjeru
  - *When dealing with actors, it is important to think about roles rather than people or job titles(Fowler)*
  - Više korisnika može da ima istu ulogu
  - Jedan korisnik može da ostvaruje više uloga

- Akteri realizuju slučajeve korišćenja
  - Jedan akter može da bude uključen u više slučajeva korišćenja
  - Jedan slučaj korišćenja može da obuhvati više aktera
  - Da li je akter obavezno čovjek?
    - Primjer: Accounting system

- Preporuka za velike sisteme: prvo identifikovati aktere pa onda slučajeve korišćenja za svakog aktera

Veze između aktera i slučaja korišćenja



- Komunikaciju može inicirati akter ili slučaj korišćenja (bidirekciona veza)

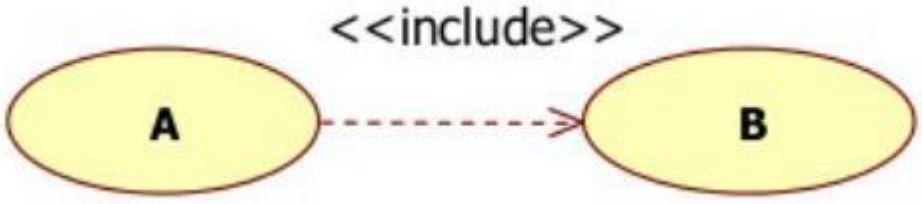
- Multiplikativnost >1 na strani aktera
  - za pokretanje slučaja korišćenja potrebno je više aktera (konkurentno ili sekvencijalno)

Veze između slučajeva korišćenja

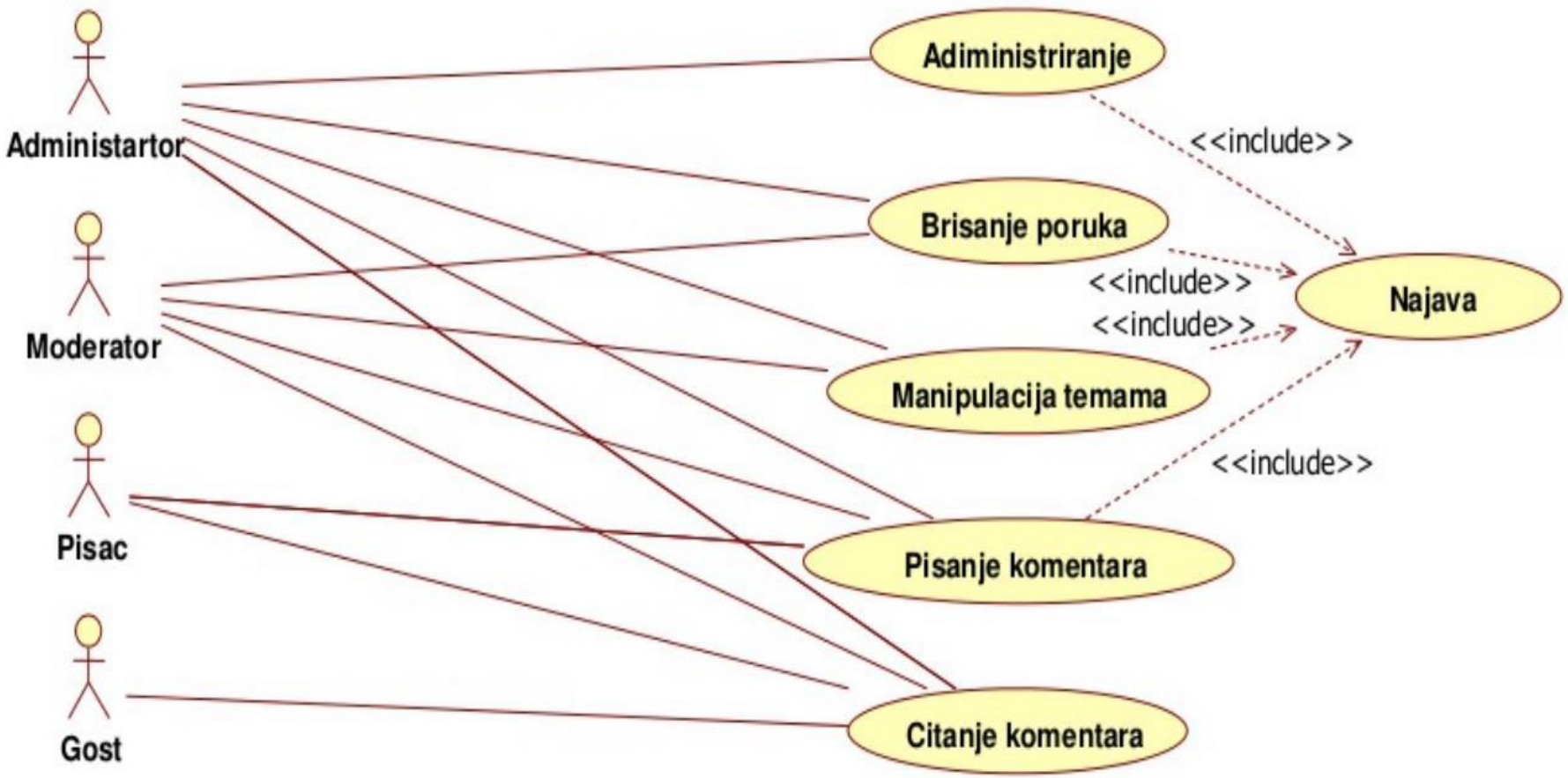
1. Veza uključivanja (<<include>>)
2. Generalizacija
3. Veza proširivanja (<<extend>>)

Veza uključivanja

- Kada postoji zajedničko ponašanje (niz koraka) koje se ponavlja u više slučajeva korišćenja
- Relacija uključivanja od slučaja korišćenja A prema slučaju korišćenja B ukazuje da će slučaj korišćenja A uključiti i ponašanje slučaja korišćenja B
- Ponašanje opisano u B obavezno je za A

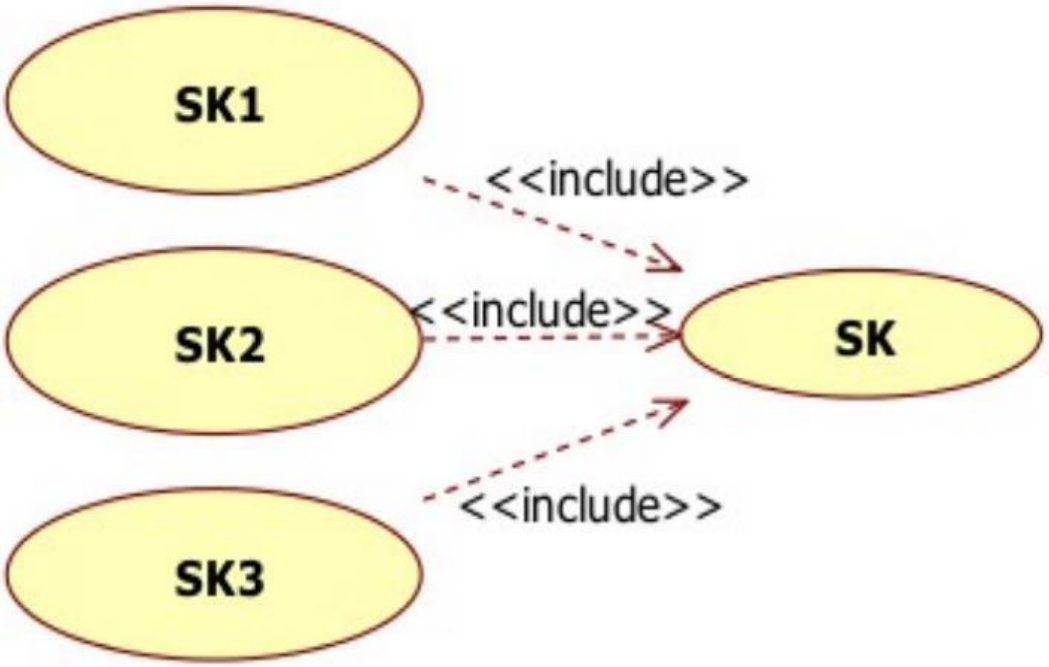


Veza uključivanja



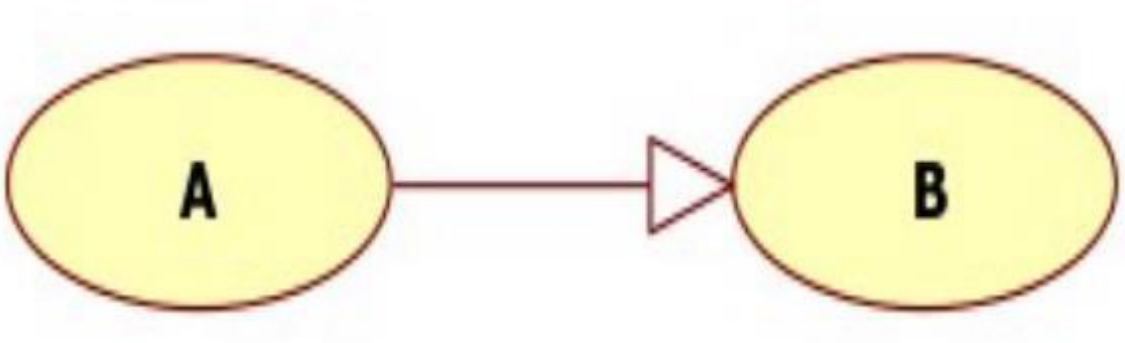
Veza uključivanja

- Tipična situacija
- Još jedan primjer: *Analyze Risk* i *Price Deal* uključuju *Valuation (procjenu vrijednosti)*



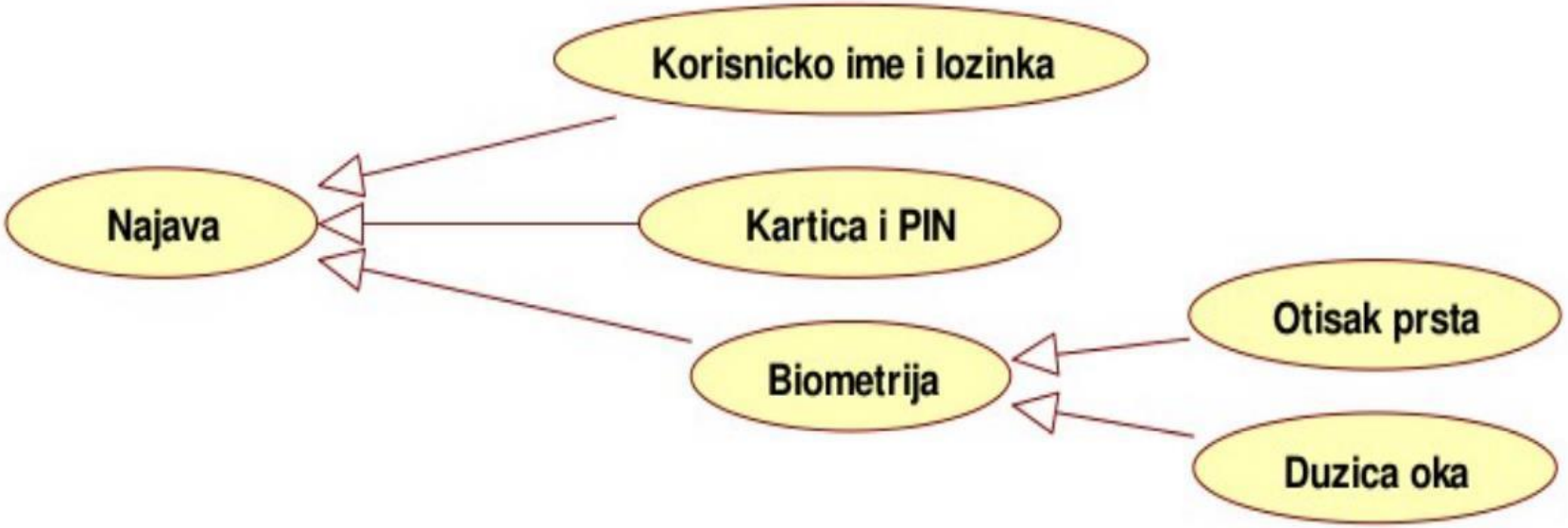
Generalizacija

- Relacija generalizacije od slučaja korišćenja A prema slučaju korišćenja B ukazuje da je slučaj korišćenja A **specifičan slučaj opštijeg slučaja** B
  - korisna za predstavljanje alternativnih scenarija



Generalizacija

- Neformalno: koristimo generalizaciju kada imamo jedan slučaj korišćenja koji je sličan drugom ali radi nešto više



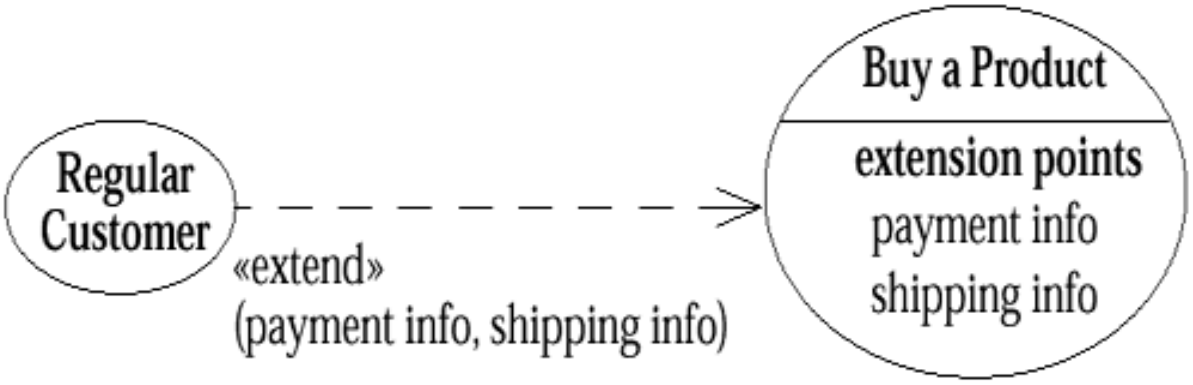
Generalizacija

- Još jedan primjer:
  - *Capture Deal* – „all-goes-well“ slučaj
  - *Exceeds Limit* – alternativni scenario kada se prekočio limit vezan za klijenta
  - Moguće oba slučaja staviti u isti kao što smo uradili kod *Kupovine proizvoda* ali odlučili smo da je slučaj dovoljno različit od originalnog da treba da bude modeliran zasebnim slučajem korišćenja
  - Specijalni slučaj može da bude vrlo različit od primarnog ali treba da dijele isti korisnički cilj

Veza proširivanja (<<extend>>)

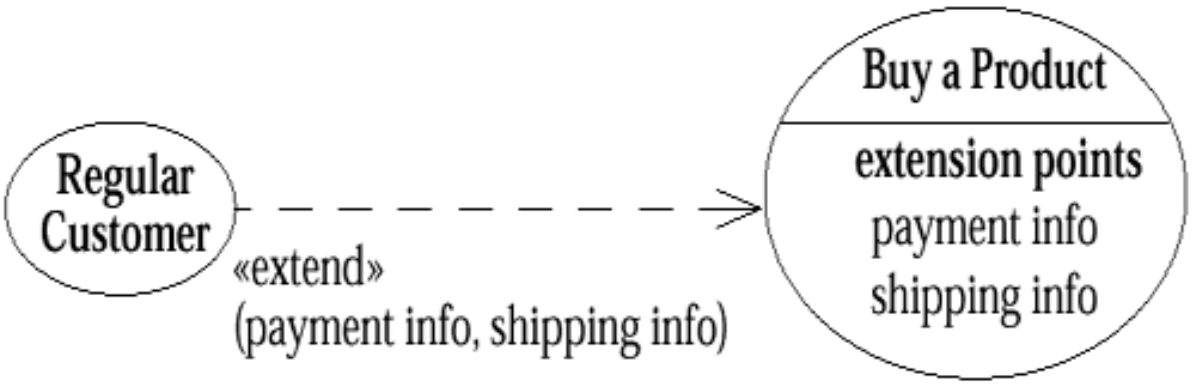
- Slična generalizaciji ali sa više pravila

- Proširujući slučaj može dodati ponašanje na primarni slučaj ali samo u određenim **tačkama proširenja** koje definiše primarni slučaj



Veza proširivanja (<<extend>>)

- Primarni slučaj može imati **više** tačaka proširenja
- Proširujući slučaj korišćenja može proširivati **jednu ili više** tačaka proširenja
- Tačke koje se proširuju se navode ispod ključne riječi **<<extend>>**
- I generalizacija i proširivanje omogućavaju da se podijele slučajevi korišćenja

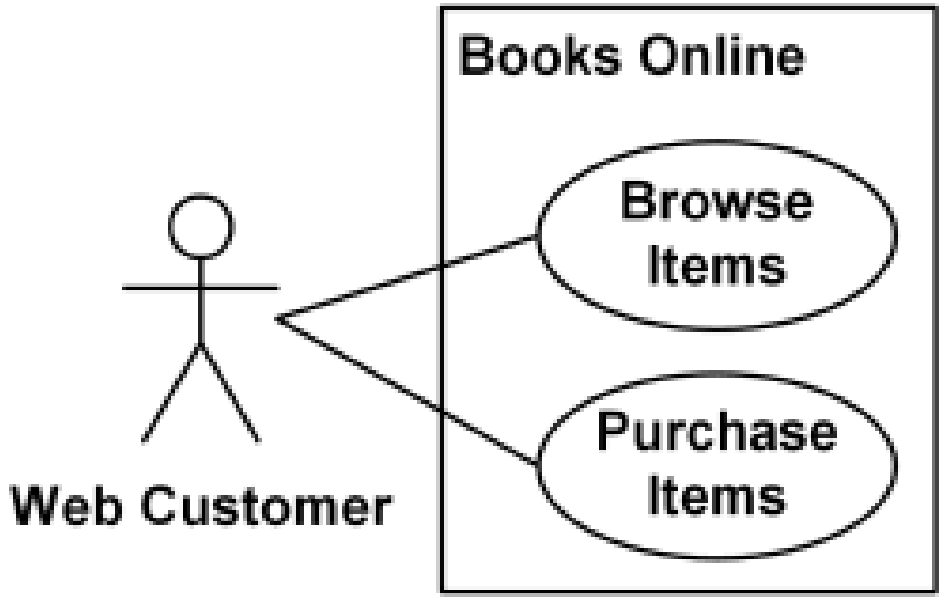


Preporuke (Fowler)

- Use *include* when you are repeating yourself in two or more separate use cases and you want to avoid repetition.
- Use *generalization* when you are describing a variation on normal behavior and you wish to describe it casually.
- Use *extend* when you are describing a variation on normal behavior and you wish to use the more controlled form, declaring your extension points in your base use case.

Subjekat

- Sistem koji izvršava slučaj korišćenja se predstavlja pravougaonikom i imenom
- Može biti podsistem/komponenta/klasa



Biznis i sistem slučajevi korišćenja

- Sistem slučajevi korišćenja su interakcije sa sistemom
- Biznis slučajevi korišćenja definišu kako biznis odgovara korisnicima

- Fowler:
  - *In my work, I focus on business use cases first, and then I come up with system use cases to satisfy them. By the end of the elaboration period, I expect to have at least one set of system use cases for each business use case I have identified-at minimum, for the business use cases I intend to support in the first delivery*

Literatura

- UML Distilled - A Brief Guide to the Standard Object Modeling Language by Martin Fowler, 2003

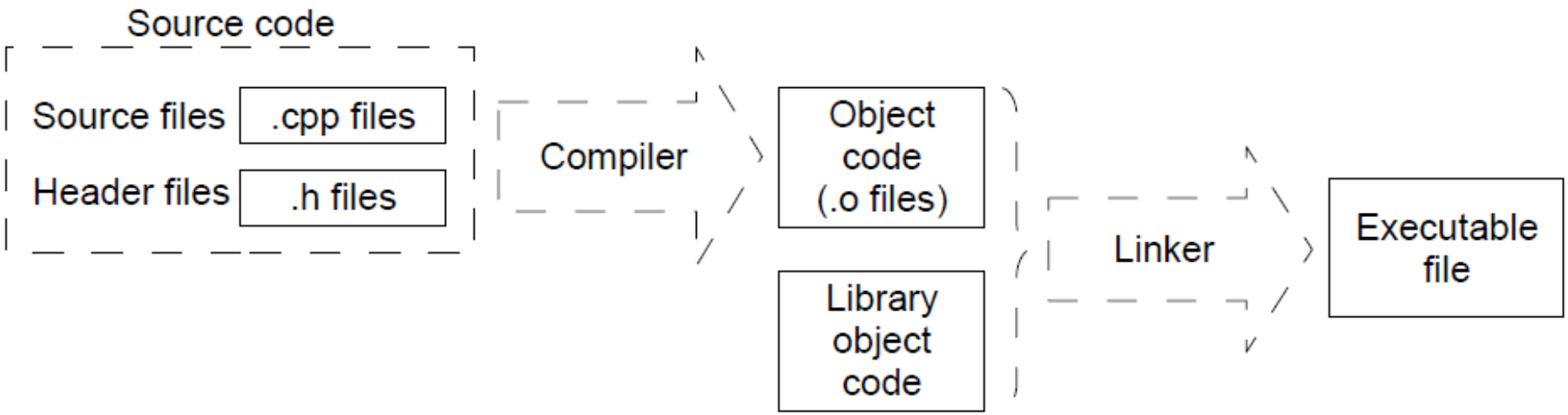
- Practical Object-Oriented Design with UML, Second Edition, Mark Priestley

Vježbe

C++

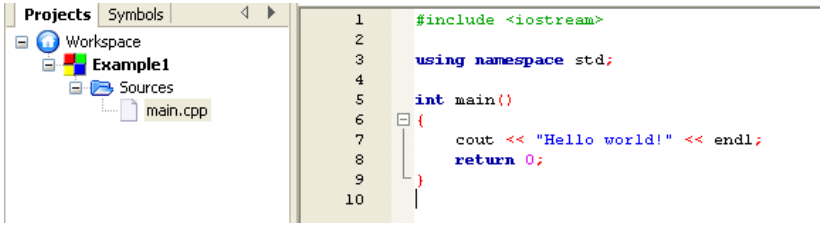
- Popularan i široko korišćen u industriji
- Statički tipiziran jezik – tipovi varijabli su poznati u vrijeme kompajliranja (deklaracija)
  - *int a;*
  - *char b;*
- Jezik slobodna forme (free-form) – „uvlačenja“ linija (*eng. indentation*) nemaju značenje kao npr. kod Python-a
- Podžava i **proceduralnu** i **objektno-orijentisanu** paradigmu
- Jezik srednjeg nivoa apstrakcije
- Najnoviji standard je C++20
- Mnogi kompajleri za C++: GCC (open source), Intel C++ compiler, Visual C++ (Microsoft), Turbo C++ (Borland) itd.
- 2 glavne komponente jezika: *Core* i *Standard library*

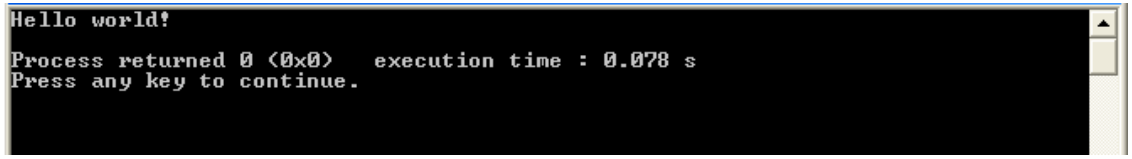
Tipični C++ projekat



STL ili 3rd party

Hello World primjer





Sintaksa

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

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

- Linije koje počinju sa # su direktive za pre-procesor
  - *#include <iostream>* uključuje se iostream standardni fajl -> poslije ove linije, možemo da koristimo funkcije definisane u *iostream* fajlu

Sintaksa

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

- U standardnoj biblioteci varijable i funkcije su definisane u opsezima (*namespace*)
- U ovom slučaju namespace je *std*

Sintaksa

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

- Main – početna tačka izvršavanja programa
  - *Nema značaja na koje mjesto u kodu je stavimo*
  - *() – deklaracija funkcije*
  - *{} – definicija (tijelo funkcije) – ono što se izvršava pri pozivu*

Sintaksa

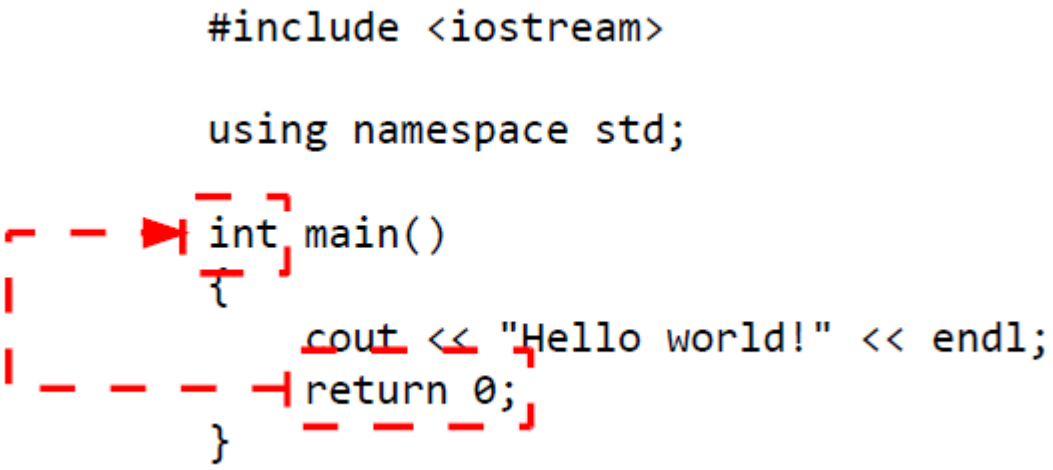
```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

- *cout* – standardni output (najčešće ekran)
- << insertuje niz karaktera (string) u standardni output
- *end* – novi red
- *cout* i *end* su članovi *std* namespace-a
- ; na kraju svake komande

Sintaksa



```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

- Naredbom *return*, *main* funkcija završava i vraća vrijednost 0
- Vraćena 0 je uobičajen način da se završi main funkcija – 0 znači da je izvršavanje proteklo kao što je očekivano

Sintaksa - komentari

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5
6  // Block comments can be done using  symbols /* and */
7
8  /*
9  Comments are crucial for programs to be understood
10
11 do not forget to comment your programs
12 */
13
14 int main()
15 {
16     cout << "Hello world!" << endl;                  //comment a line with //
17
18     cout << "This is a second example"<< endl        //one single statement
19          <<" of C++ program"<< endl                 //can be written
20          <<" with comments and more text"<<endl;    //on several lines
21     return 0;
22 }
23
```

- Komentar u jednoj liniji
- Blok komentar
- Preporuka: komentarišite kod

Sintaksa – imena varijabli/funkcija/klasa...

- Ime može da sadrži SAMO: slova, cifre, donju crtu (_)
- Ime MORA da počinje slovom ili donjom crtom (_) (nikako cifrom)
- „space“ i znaci interpunkcije nisu dozvoljeni u imenu
- C++ je case-sensitive – razlikuje mala i velika slova

Sintaksa – imena varijabli/funkcija/klasa...

- Rezervisane ključne riječi jezika se NE SMIJU upotrebljavati za imena varijabli ili funkcija

- Neke od rezervisanih riječi:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast, continue,
default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern,
false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new,
operator, private, protected, public, register, reinterpret_cast, return, short,
signed, sizeof, static, static_cast, struct, switch, template, this, throw, true,
try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile,
wchar_t, while
```

Osnovni tipovi podataka

Name	Description	Size	Range
char	Character or small integer	1 byte	signed: -127 to 127 unsigned : 0 to 255
short int (short)	Short integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer	8 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	Floating point number.	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)
bool	Boolean value. It can take one of two values: true or false.	1 byte	true or false
wchar_t	Wide character	2 or 4 bytes	1 wide character

Deklaracija varijable

- ...je obavezna

  int a;

  float b;
- Poslije deklaracije varijabla može da se koristi
- Moguće deklarisati i inicijalizovati u istoj liniji:

  float g = 9.81;

  float c (5);
- Moguće deklarisati više varijali odjednom

  int x,y,z;
- *char, int, short, i long* tipovi mogu biti signed i unsigned
  - *unsigned short NumberOfPeople; // ne može biti negativan*
  - *signed int MyAccountBalance; // može biti negativan*

Životni vijek varijable

globalne varijable

- **može im se pristupiti iz bilo kog dijela koda**
- **extern – pristup iz drugog fajla**
- **izbjegavati**

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

int gVar1, gVar2;

int main ( int argc, char**argv) //notice the arguments here
{

int num = 5;

cin>>gVar1>>gVar2; // input values from keyboard and store
return 0;
}
```

lokalne varijable

- **može im se pristupiti samo iz bloka u kome su definisane a koji je ograničen zagradama {}**

Aritmetičke operacije

- Ništa novo: +, -, *, -, %
- Mogu biti kombinovane sa dodjelom (=) i ova notacija je vrlo česta:
  - a = a +2; // ekvivalentno je sa a += 2;
  - b = b*.33; // ekvivalentno je sa b *= .33;
  - c = c/4; // ekvivalentno je sa c /= 4;
  - d = d*10 // ekvivalentno je sa d *= 10;
- Pažljivo sa cjelobrojnim dijeljenjem
  - int a(2), b(3);
  - a/b će vratiti 0
  - double (a)/double (b) će vratiti 0.6666

Aritmetičke operacije

- Skraćena notacija za uvećavanje i umanjivanje za 1
  - a++; // je ekvivalentno sa a = a + 1; i sa a += 1;
  - b--; // je ekvivalentno sa b = b - 1; i sa b -= 1;

- PAŽLJIVO sa ovim operatorima jer postoji razlika kada se koriste kao sufiks i kao prefiks

```
a = 2;
b = a++;
```

```
a = 2;
b = ++a;
```

Standard Template Library (STL)

STL

- Skup C++ template klasa za najčešće korišćene strukture podataka: niz, lista, stek, red, skup, mapa itd.

- Namespace STL biblioteke je **std**
  - std::vector<int> myVector

- Sadrži:

1. Kontejnere ili strukture
   a) Sequence containers: vector, list, deque containers
   b) Container adaptors: stack, queue, priority_queue
   c) Associative containers: set, multiset, map, multimap, bitset
2. Algoritme (sortiranje, pretraga)
3. Iteratore (za rad sa sekvencama)

Prednosti

- Ne brinemo o upravljanju memorijom
- Brze i optimizovane operacije
- Algoritmi (#include<algorithm>)

- Svaka struktura sadrži korisne funkcije i operatore

- Duh generičkog programiranja: implementacija algoritama i struktura BEZ zavisnosti od tipa podataka

Prednosti

- Npr. std::vector<T> je niz promjenljive veličine
  - ne brinemo o memoriji – nema curenja memorije
  - sigurniji je (nema *buffer overflow* grešaka)

- Ne moramo da implementiramo strukture poput reda, steka, liste, asocijativnih nizova (mape)
- Efikasni algoritmi nezavisni od tipa (sort, search, reverse)

- Efikasna String klasa

- Manja vjerovatnoća za bug-ove

std::vector<T>

- Niz promjenljive veličine
- Slučajni pristup svim elementima (random access) preko [ ]
- Memorijski zahtjevi kao kod običnog niza
  - Zauzima kontinuirani dio memorije
- Dodavanje elementa svuda osim na kraj nije preporučljivo (nije efikasno) ali je moguće (insert)
- #include <vector>
- Pripada grupi *sequence containers*

std::vector<T>

- Mana: [ ] operator kao kod običnog niza NE PROVJERAVA granice -> opasnost *segmentation fault* grešaka
- Postoji anternativa *„at“* operator koji PROVJERAVA granice ali je malo manje efikasan

- Ugrađene funkcije:
  - unsigned int size(); //vraća broj elemenata niza
  - push_back(type element);//dodaje element na kraj niza
  - bool empty(); //vraća True ako je niz prazan
  - void clear(); //briše sve elemente niza
  - type at(int n); //vraća element na poziciji *n* sa PROVJEROM granica

- = //dodjela kompletnog sadržaja jednog vektora drugom
- == //upoređivanje jednog po jednog elemenata sa odgovarajućim u drugom nizu

std::vector<T>

```
#include <iostream>
#include <vector> //inclusion here
using namespace std; // to avoid std::vector everywhere

int main()
{
    vector <int> example;         //Vector to store integers
    example.push_back(5);         //Add 5 at the end of the vector
    example.push_back(-1);        //Same for -1
    example.push_back(4);         //Same for 4

    for(int x=0; x<example.size(); x++)     {//call size() in the loop
        cout<<example[x]<<" ";    //operator [] similar to arrays
    }

    if(!example.empty())          //Checks if empty
        example.clear();          //Clears vector
    vector <int> another_vector;  //Creates another vector of integers

    another_vector.push_back(10); //Adds to end of vector
    example.push_back(10);        //Same : both vector contain the number 10

    if(example==another_vector)   //To show testing equality
    {
        example.push_back(20);
    }
    for(int y=0; y<example.size(); y++)
    {
        cout<<example[y]<<" ";    //Should output 10 20
    }
    return 0;
}
```

```
5 -1 4
10 20
```

std::vector<T>

- Ne brinemo o mijenjanju veličine niza sa new i delete pri dodavanju i brisanju
- Nezavisan od tipa – template klasa

Iteratori

- Iterator se može razumjeti kao pokazivač na element neke veće strukture (npr. niza)
- Sve strukture implementiraju sljedeće funkcije:
  - begin() – vraća iterator koji ukazuje na prvi element u strukturi
  - end() – vraća pokazivač na kraj kontejnera (NE NA POSLJEDNJI ELEMENT već na mjesto poslije njega)
- Može se dereferencirati sa * poput pokazivača

Iteratori

- deklaracija:
  - *class_name<template_parameters>::iterator* ime

    vector <int> myIntVector;

    vector <int> :: iterator myIntVectorIterator;
- Dva iteratora mogu se porediti sa operatorima
  1. !=
  2. ==

Iteratori

- Dijele se na grupe po tome da li se element na koga iterator ukazuje može preko njega modifikovati ili samo čitati

- Također dijele se na 3 grupe po tipu kretanja: **forward**, **backward** i **bidirectional**
- Za prelazak na sljedeći element u kontejneru koristi se operator **++**
  - Radi samo na forwad i biderectional iteratorima
- Za prelazak na prethodni element u kontejneru koristi se operator **--**
  - Radi samo na backward i biderectional iteratorima

Iteratori

Tipični C/C++ pristup prolaska kroz niz - **IZBJEGAVATI**

```
//#include <usual headers>...

using namespace std;
vector<int> myIntVector;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

//span the array using index and .Size() function
for(int i = 0; i<myIntVector.size(); i++)
{
    cout << myIntVector[i] << " ";
}
```

```
1 4 8
```

Iteratori

Tipični **STL** pristup prolaska kroz niz – **koristiti iteratore**

```
using namespace std;

vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

//Span the array, but...
//No Size() function :  (supposedly) faster
//No index, only an iterator: close to ptr approach

for(myIntVectorIterator = myIntVector.begin();
    myIntVectorIterator != myIntVector.end();
    myIntVectorIterator++)
{
    cout<<*myIntVectorIterator<<" ";
}
```

```
1 4 8
```

Iteratori

- iteratori slučajnog pristupa (random access iteratosr) su iteratori vezani za strukture koje omogućavaju slučajni pristup kao npr. **std::vector**

- Sa njima možemo pristupati djelovima kontejnera

- Sa njima možemo koristiti aritmetičke operatori +, -, +=, -= kao i <, >, <=, >=
  - Npr
    - ***iterator + n*** gdje je n integer
    - Ova linija vraća pokazivač na element u nizu udaljen *n* mjesta od elementa na koji ukazuje iterator
    - PAŽLJIVO sa granicama niza
      - vector<int> myIntVector;
      - vector<int>::iterator myIntVectorIterator;
      - myIntVectorIterator = myIntVector.begin() + 2;

Šta će se desiti kad se izvrše ove 3 linije ?

Iteratori

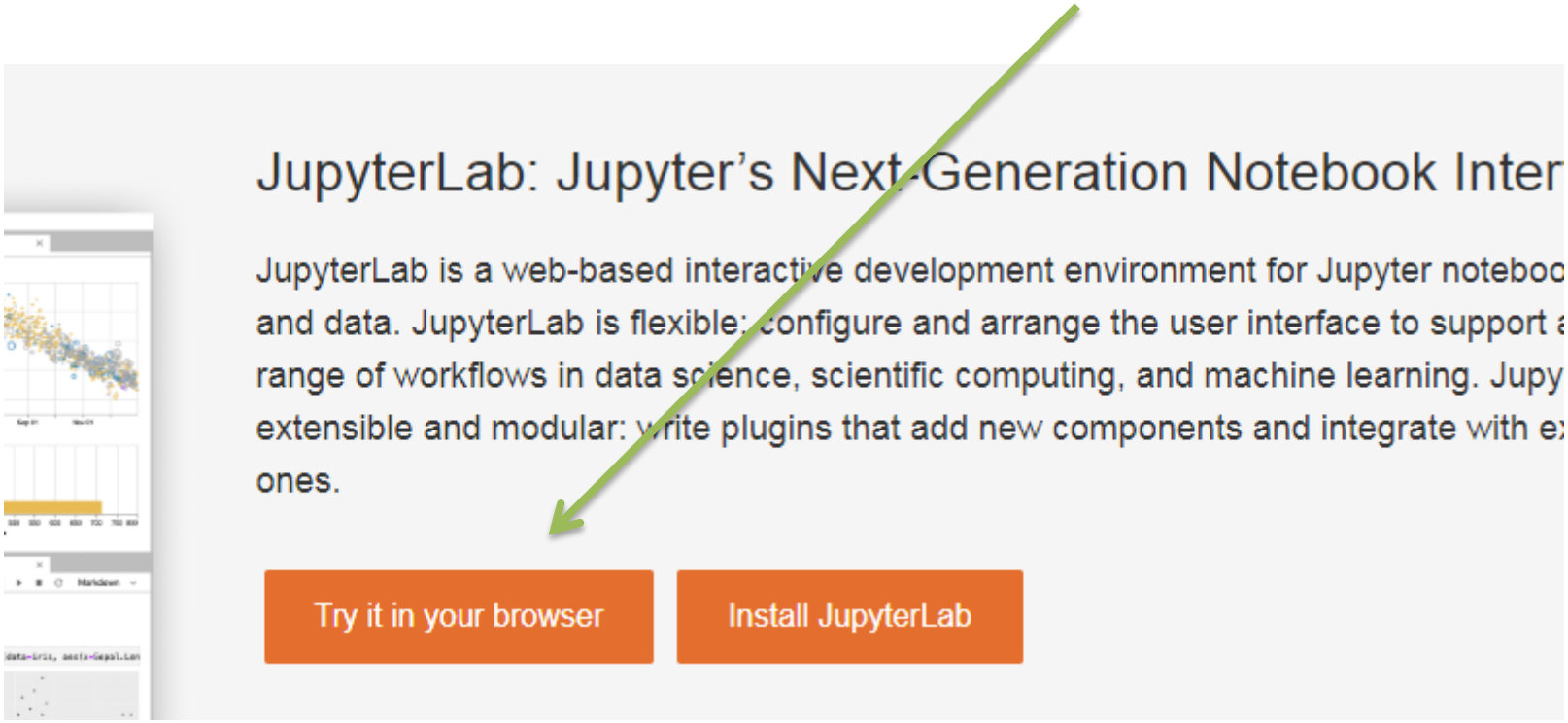
- Mnoge funkcije ugrađene u kontejnere zahtijevaju iteratore kao argumente jer rade na opsezima, recimo funkcija *erase*
  - myIntVector.erase(myIntVector.begin(), myIntVector.end());
  - myIntVector.erase(myIntVector.begin(), myIntVector.begin()+2);

  - Različiti kontejneri podržavaju različite vrste iteratora:
    - vector podržava najgeneralniji tip iteratora - random access iterator
    - lista podržava samo bidirekcioni iterator – jer lista ne podržava random access svojim elementima

- Iteratori su posebno korisni kod kontejnera koji nemaju očigledan način za prolaženje kroz sve elemente kao što ima vector (recimo mape)
- Mana: ne provjeravaju granice -> *segmentation fault* greške
- Ako promijenimo strukturu (kontejner), i dalje možemo da koristimo kod sa iteratorima – sve što treba promijeniti je tip iteratora
  - Pod uslovom da koristimo samo osobine koje podržavaju obje strukture (ovo je mana)

Online okruženje za C++

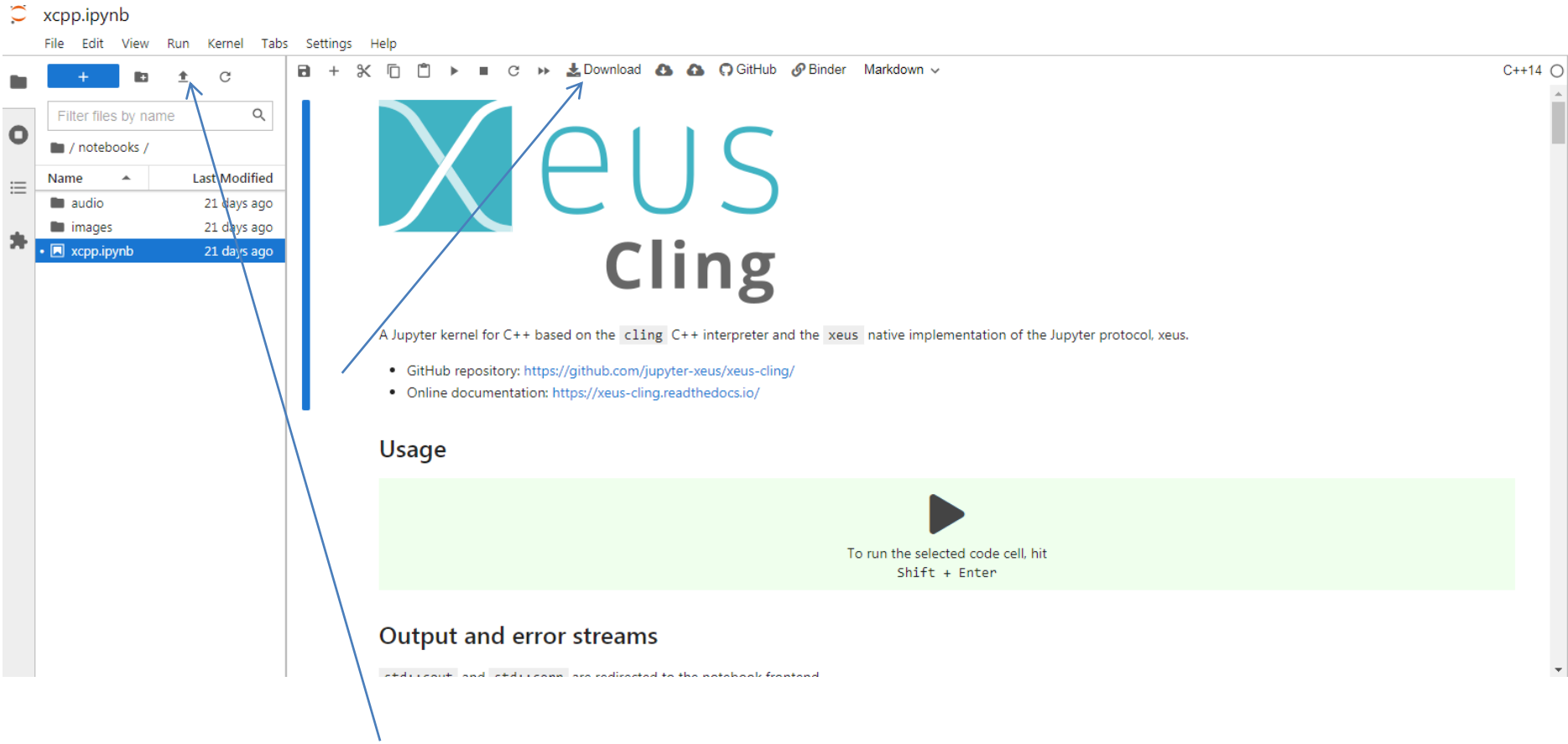
- https://jupyter.org



Online okruženje za C++

- https://jupyter.org

Na početku i kraju časa



Online okruženje za C++

- Kao u slučaju svih Jupyter sveski, možete da kombinujete tekst, kod i izlaz kod-a

- Shift + Enter je komanda za pokretanje koda koji se nalazi u ćeliji koju ste odabrali

- Napravićemo jednu praznu svesku

Druga mogućnost je lokalno okruženje

- Npr Visual Studio Code

  Uputstvo za instalaciju kompajlera i podešavanje okruženja:

  https://code.visualstudio.com/docs/languages/cpp

Vježba 1

- Podešavanje okruženja
- Hello world u C++
- Korišćenje vektora i mape u STL

Vježba 2

- Deklarisati vector koji čuva geometrijske figure: trouglove, krugove, četvorouglove
  - Odlučiti kako se čuvaju geom tijela
  - vector vector-a ? Ili vector objekata 3 klase ?
- Napraviti funkciju koja omogućava korisniku popunjavanje vector-a geometrijskim figurama
  - Dodati osnovne provjere (npr da radisu nije negativan u slučaju kruga)
- Napraviti funkciju koja štampa vector – za svaki element vector-a štampa se tip geometrijske figure („krug“, „trougao“, „četvorougao“)
- Napraviti funkciju koja sortira vector po **obimu** geom tijela
- Pokušajmo da napravimo kod otporan na greške i pogrešne unose

UML alati

- https://www.umlet.com/
- Online verzija: http://www.umletino.com/umletino.html

- https://app.diagrams.net/
- https://plantuml.com/