- ***Iterative development !!!***
  - Sistem ne "napadamo" u cjelini već dio po dio
  - U svakoj iteraciji se kreira *production-quality software* koji zadovoljava određeni podskup zahtjeva sistema
  - Svaka iteracija se sastoji od cijelog skupa aktivnosti:
    - Dizajn
    - Implementacija
    - Testiranje
  - Cilj je postupno proširiti i "rafinirati" sistem kroz kontinuiranu komunikaciju s korisnicima

- Što izbjegavamo takvim procesom ?
  - *"Rush to code"* – odmah za tipkovnicu krenuti kucati kod koji neće valjati
  - *"Analysis paralysis"* – pokušava se dizajnirati model u svojoj cjelini sa svim detaljima i tek onda krenuti u implementaciju (kucanje koda)

- Iterativni razvoj rješava problem *feedback*-a, ali često se ne uspijeva akumulirati znanje zbog nedostatka **apstrakcije!**

Što je "apstrakcija"

"Apstrakcija označava esencijalne karakteristike objekta koje ga razlikuju od svih drugih vrsta objekata i koje definiraju oštre konceptualne granice objekta"

Enkapsulacija

OO enkapsulacija je pakiranje operacija i atributa koji predstavljaju stanje u tip objekta (odnosno klasu) tako da je stanje dostupno ili promjenjivo samo preko sučelja definiranog enkapsulacijom

Što nam donosi ?

- Smanjuje se složenost sustava
  - Lakše je graditi veće sustave kad su unutrašnji detalji manjih konstitutivnih dijelova sakriveni
  - **Modularnost !**
- Olakšava se održavanje
  - Promjene su **lokalizirane**!

- Koncept **statičkog i dinamičkog povezivanja** (*early and late binding*)
- Statičko povezivanje
  - Tipovi svih varijabli i izraza su fiksni (i poznati!) kod prevođenja
- Dinamičko povezivanje
  - Prilikom prevođenja **nisu** poznati tipovi svih varijabli !?
  - Zar nije osnovna namjena deklaracije tipova da kompajler precizno zna koja varijabla je kojeg tipa kako bi zauzeo potrebni memorijski prostor ?

- Dinamičko povezivanje je tehnika pomoću koje se točan komad kôda koji će se izvršiti određuje tek prilikom izvođenja

---

Apstraktni razredi

- Apstraktan razred predstavlja “nedovršenu” klasu
- Predviđeni za daljnje nasljeđivanje i **ne mogu** se instancirati

Cemu služi ?

- Služi za modeliranje koncepata (objekata) koji su nepotpuni u smislu da im izvedena klasa **mora** dodati određeno ponašanje

Sučelja - *interface*

- Sučelje razreda predstavlja sve ono što razred "daje" prema van (odnosno na što se vanjski razredi mogu **osloniti**)
  - Sučelje čine svi javni dijelovi razreda (*public* članske varijable i članske funkcije)
- Što je to razred sučelja ?
  - To je apstraktni razred koji definira skup operacija koje **moraju** implementirati izvedeni razredi
  - Tada kažemo da razred **zadovoljava** sučelje
  - Koncept "*programing to interface*"
    - Razredi komuniciraju jedan s drugim preko sučelja i oslanjaju se samo na usluge koje su specificirane u sučelju !!!

UML

- Ovo mapiranje omogućava *forward engineering*:
  - Generiranje koda iz UML modela u prog. jeziku
- Mogućnost *reverse engineering*:
  - Generiranje UML modela na osnovu prog. kôda

"Pogledi" UML-a (*views*)

- Pogled – podskup modelirajućih elemenata koji predstavljaju jedan aspekt sistema
- Klasifikacija pogleda:
  - Strukturna klasifikacija
    - Opisuje stvari u sistemu i njihove međusobne odnose
    - Razredi, *use cases*, komponente i čvorovi
  - Dinamičko ponašanje
    - Opisuje ponašanje sistema tijekom vremena
  - Upravljanje modelom
    - Opisuje organizaciju samih modela u hijerarhijske jedinice

- Tri vrste odnosa
  - Asocijacije – predstavljaju strukturne odnose među objektima
  - Generalizacija – povezuje generalizirane razrede sa njihovim specijalizacijama
  - Ovisnost (*dependency*) – predstavlja relaciju korištenja među razredima

---

- *Objektno-orijentirana analiza podrazumijeva ispitivanje zahtjeva iz perspektive razreda i objekata nađenih u vokabularu domene problema*
- *Objektno-orijentirani dizajn podrazumijeva provođenje objektno-orijentirane dekompozicije uz notaciju za opisivanje kako logičkih i fizičkih, tako i statičkih i dinamičkih modela sistema koji se dizajnira*

- **Krutost** – sistem se teško mijenja jer svaka promjena povlači mnoge druge promjene u ostalim dijelovima sistema
- **Krhkost** – promjene u sistemu uzrokuju pogreške u dijelovima sistema koji nemaju nikakve konceptualne veze s promijenjenim dijelom
  - Najčešći uzrok je nužnost unošenja jedne izmjene na više mjesta
- **Nepokretnost** – sistem se teško razdvaja u komponente koje bi se mogle iskoristiti u drugim sistemima
- **Nepotrebna složenost** – dizajn sadrži infrastrukturu koja ne pruža nikakvu dodatnu korist
- **Nepotrebno ponavljanje** – dizajn sadrži ponavljajuće strukture koje bi se mogle unificirati u jednoj apstrakciji

- Koje sve domene razreda imamo
  - **Aplikacijska domena** - Sadrži razrede korisne za aplikaciju
    - Razredi koji modeliraju konkretan koncept iz domene aplikacije
  - ***Business* domena** – sadrži razrede korisne za pojedinu industriju ili kompaniju
    - Razredi odnosa, uloga i atributa
  - **Arhitekturna domena** – sadrži razrede korisne za implementacijsku arhitekturu
    - GUI razredi, razredi za upravljanje bazama podataka
  - **Fondacijska domena** – sadrži razrede korisne za sve arhitekture i *business* –e

---

- *Encumbrance* mjeri ukupnu "potrebnu potpornu mašineriju" razreda
  - Obuhvaća sve druge razrede na koje se dani razred mora osloniti pri obavljanju svog zadatka

---

Demeterov zakon

- Govori o tome kome sve objekt **smije** slati poruke
- Tehnički, objekt bi trebao pozivati servise **samo** sljedećih objekata
  - this (odnosno samog sebe)
  - Objekti koji su mu predani kao parametri
  - Objekti koje je sam kreirao
  - Bilo koji objekt na koji ima referencu, a koji je njegov podobjekt

---

- Kohezija je mjera međupovezanosti (*inter-relatedness*) svojstava (atributa i operacija) u javnom sučelju razreda
- Identificirati ćemo tri problema kohezije u pridjeljivanju svojstava (odgovornosti) razredima
  - *Mixed-instance* – grozna ☺
  - *Mixed-domain* – tako, tako
  - *Mixed-role* – ma i ova je loša, ali ne toliko

Mixed-instance cohesion

- Razred sa *mixed-instance* kohezijom ima neka svojstva koja su nedefinirana za neke objekte !

Mixed-domain cohesion

- Malo definicija:
  - Razred sa *mixed-domain* kohezijom sadrži element koji direktno opterećuje razred sa *ekstrinsičnim* razredom neke druge domene

Mixed-role cohesion

- Razred **C** sa *mixed-role* kohezijom sadrži element koji direktno opterećuje razred sa ekstrinsičnim razredom koji leži u istoj domeni kao i **C**

Single-Responsibility Principle

Razred treba imati (modelirati) samo jednu odgovornost (*responsibility*).

Open-Closed Principle

- Softverski entiteti (razredi, moduli, funkcije, ...) trebaju biti otvoreni za proširenje, ali zatvoreni za modifikaciju

---

- Prostor stanja razreda **C** je skup svih dozvoljenih stanja objekata razreda **C**
  - Podsjetnik: stanje objekta je skup vrijednosti svih njegovih atributa
  - Dimenzionalnost prostora stanja je jednaka broju *koordinata* potrebnih za opis stanja danog objekta

Liskov Substitution Principle

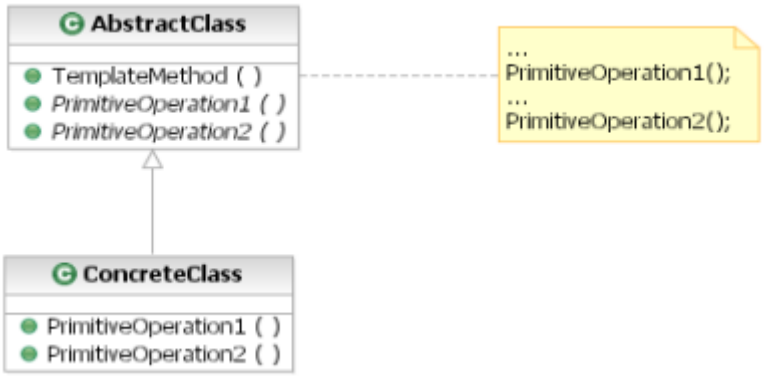
- Definira ispravno korištenje nasljeđivanja !!!

Podtipovi moraju biti supstitabilni (umetljivi) umjesto svojih baznih tipova !!!

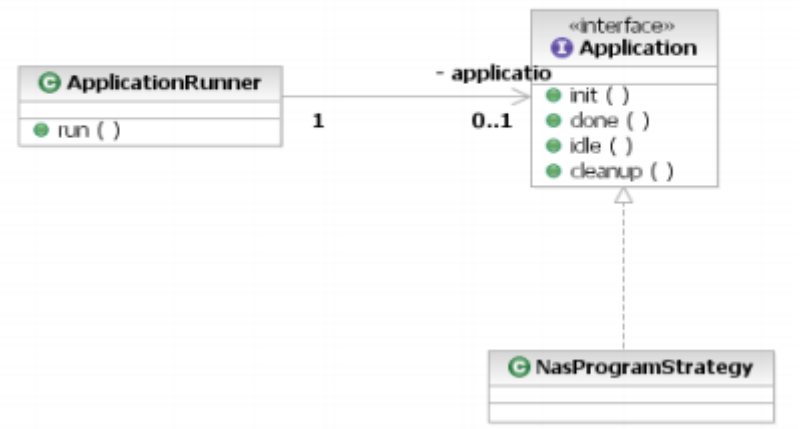
– U stvari kaže da ponašanje programa mora ostati nepromijenjeno kada umjesto objekta baznog razreda podmetnemo objekt izvedenog razreda!!!

- *Overload* – preopterećenje
  - Preopteretiti funkciju **f()** znači definirati drugu funkciju s istim imenom unutar istog dosega (*scope*), ali s različitim parametrima
  - Kod poziva funkcije **f()** kompajler pokušava naći najbolje slaganje i ta funkcija se poziva
- *Override* – prekrivanje
  - Prekrivanje *virtualne funkcije* **f()** znači definiranje iste funkcije (s istim imenom i istim parametrima) u izvedenom razredu
- *Hide* – skrivanje
  - Sakriti funkciju **f()** iz obuhvaćajućeg dosega (*enclosing scope*) (npr. bazni razred, namespace, ..) znači definirati funkciju s istim imenom u unutrašnjem dosegu (*inner scope*) (npr. izvedeni razred, *nested* razred, namespace, ...)

TEMPLATE PATTERN



Strategy design pattern



Što je proces razvoja softvera

Još se naziva i **metodologijom**

Jednostavnim riječima:

- Kako izgraditi traženi sistem polazeći od skupa zahtjeva korisnika

Što proces mora sadržavati

- Osnovne aktivnosti
  - Skupljanje zahtjeva i njihovo grupiranje u funkcionalne cjeline
  - Kreiranje logičke strukture sustava (dizajn)
    - Definiranje skupa razreda/objekata, njihovih statičkih i dinamičkih karakteristika
    - Identificiranje objekata, njihovih karakteristika (što predstavljaju i kakve odgovornosti imaju) i njihovog ponašanja (kako interagiraju sa drugim objektima)
  - Kreiranje fizičke strukture sustava (arhitektura)
    - Razlaganje sustava u (maksimalno nezavisne) komponente
  - Implementacija
  - Testiranje
  - Instalacija

Kako izgleda moderan proces ?

Dvije karakteristike:

Iterativan i inkrementalan

- Razvoj sistema se odvija u više iteracija (mini-projekata)
- U svakoj iteraciji se gradi funkcionalan sistem koji zadovoljava određeni (pod)skup zahtjeva

Pokretan zahtjevima (*use case driven*)

- Razvoj sistema se odvija uz konstantnu interakciju s korisnicima kako bi što bolje zadovoljio njihove stvarne potrebe

Kako ćemo mi definirati *use case*

Use case opisuje ponašanje sistema u različitim uvjetima kod reakcije sistema na zahtjev nekog *stakeholder*-a

Domain objects

- Objekti domene nam definiraju model nad kojim developeri i korisnici mogu diskutirati o aplikaciji

Neke tipične uloge objekata:

- ***Information holder*** – posjeduje i daje informacije
- ***Structurer*** – održava odnose među objektima
- ***Service provider*** – odrađuje posao, i općenito, pruža *computing services*
- ***Coordinator*** – reagira na događaje (*event*-e) delegirajući drugima zadatke
- ***Controller*** – donosi odluke i izbliza uspravlja tuđim akcijama
- ***Interfacer*** – transformira informacije i zahtjeve između različitih dijelova sistema