Tema laborator 9 - 10
OOP concepte avansate

In acest laborator vom vedea 2 concepte mai putin evidente dar foarte importante

1) Compozitia:
Simplu spus, compozitia inseamna ca o clasa (clasa A) are ca membru un obiect al altei clase (clasa B). Obiectele din clasa A pot apela cod din clasa B: `b.callSth()` De obieci, obiectul b (de tip B) va fi trimis prin constructor la crearea obiectelor de clasa A. (`a = new A(b)`)

Avantajul este ca putem refolosi codul din B, fara a „importa" toate tipurile lui B. Java nu permite mostenire multipla, deci putem pastra mostenirea pentru altceva. Daca cumva clasa B se va modifica (ex. devine abstracta, sau impune alte modificari a claselor copii), prin compozitie, clasa A va ramane nemodificata. In caz ca s-ar fi folosit mostenirea, clasa A ar fi trebuit modificata.

Mai exista „pattern"-ul numit Composite, in care, facem o ierarhie de obiecte sub forma unui arbore. (Exemplul clasic e o expresie aritmetica) In laborator vom aplica principiul de compozitie pentru a ne apropia de acest pattern. (vezi documentatia de la sfarsit)

2) Principiul „open-closed". Adica, dupa ce am scris un cod, este foarte important ca respectivul cod sa poata fi extins (open code) fara a fi modificat (closed code). Pare imposibil, dar, cu un pic de atentie si planificare este usor.
In laboratorul 4 am schimbat codul din laboratorul 3 pentu a implementa polinoamele folosind tabelele. Dupa introducerea interfetelor am putut sa realizam functii sinus, logaritm, etc.
DAR inca nu putem sa avem ceva de genul x^2 + log(x). Evident, combinatiile de log, sin, exp, etc. sunt foarte multe si nu pot fi implementate manual.

Vom implementa clase ce vor implementa conceptul de adunare si inmultire a functiilor. Adica, avand f(x) = x^2 si g(x) = log(x) putem sa realizam functia h(x) = f(x) + g(x). Conceptul de compozitie apare pentru ca cele 2 functii, f si g vor fi specificate in momentul crearii obiectului h.
Open-closed pentru ca, NU modificam codul existent (nu se modifica interfata Function, nu se modifica codul lui f si g sau din RootFinding) DAR extindem functionalitatea programului adaugand clase noi. Aceste clase se integreaza cu functionalitatea existenta.

Prin urmare, codul a ramas closed (nu s-a modificat) dar design-ul este open, pentru ca l-am putut extinde.

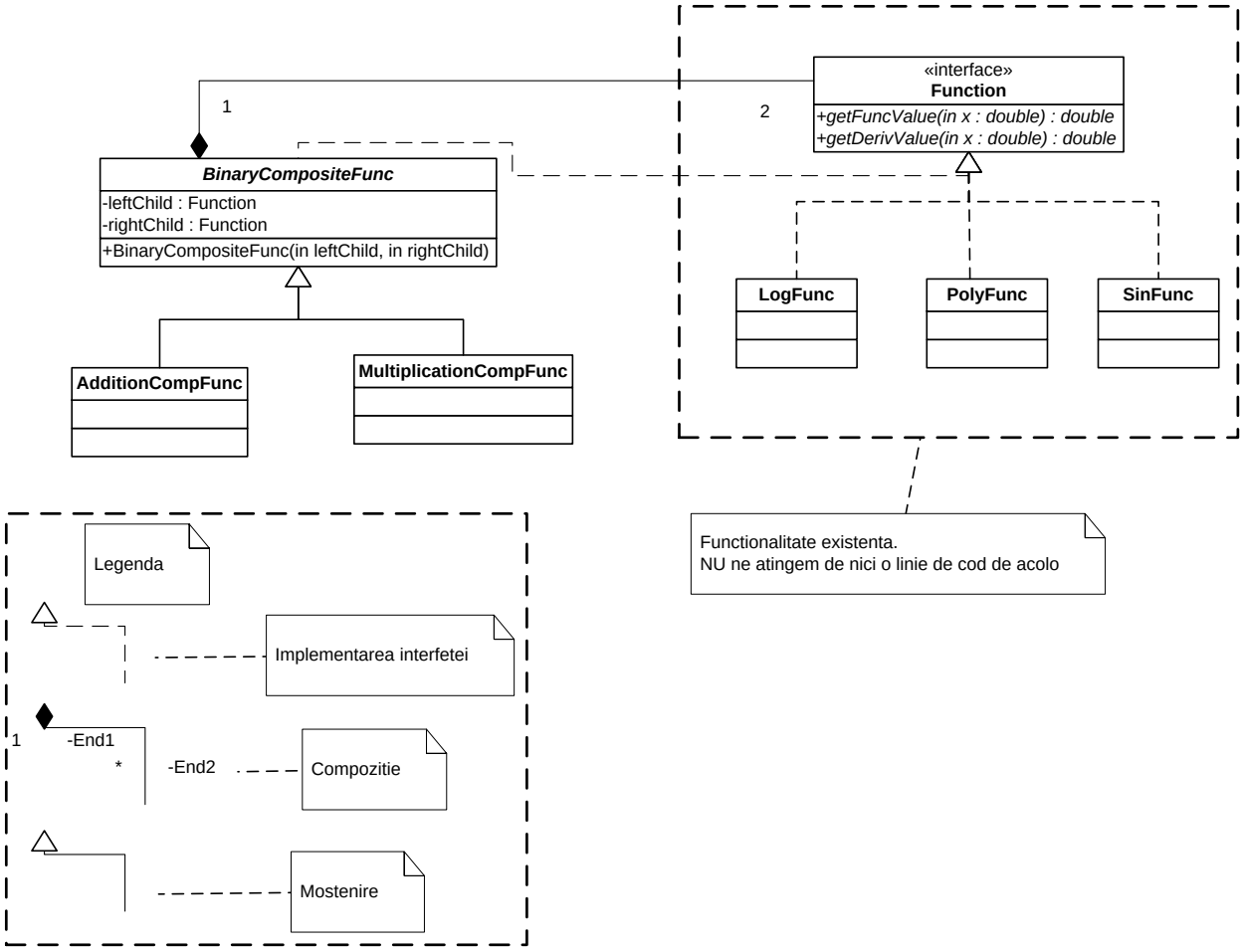
Alt argument pentru „closed" este faptul ca, de obicei, codul existent este testat si (teoretic) fara bug-uri. In momentul in care adaugam/schimbam ceva, riscam sa introducem bug-uri. Aceste bug-uri de multe ori scapa testelor existente. Asa ca, daca scriem cod nou in alte clase/pachete, ne putem concentra depanarea doar pe respectivele arii pentru ca acolo am adaugat cod nou.

Conceptul de Composition (design pattern) este pus in practica datorita faptului ca f(x) poate fi la randu lui o functie compusa: f(x) = 2x + sin(x) iar h(x) = f(x) + g(x) == 2x+sin(x)+log(x)

Aduceti-va aminte ca daca h(x) = f(x) + g(x) atunci h'(x) = f'(x) + g'(x)
Daca h(x) = f(x) * g(x) atunci h'(x) = f'(x)*g(x) + f(x)*g'(x)

1 Practica

Creati ierarhia de clase de mai jos:



Odata ce ati facut adunarea si multiplicarea scrieti teste care sa verifice daca f(x) obtinut prin compozitie este identic in valorile functiei si in valorile derivatei cu versiunea analitica. Va recomand sa luati un interval (-100 – +100) si sa testati.

Optional puteti rula functiile compuse prin RootFinding dar atentie la minimele locale! (va recomand sa desenati functia in google si sa luati o valoare de start in bazinul de atractie a radacinii. Acest lucru este foarte important mai ales la functiile cu sin/cos)

2 Evaluare

Executati toti pasii din laborator.

3 Referinte:

http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
http://java.dzone.com/articles/composite-design-pattern-java-0
http://javapapers.com/design-patterns/composite-design-pattern/

http://en.wikipedia.org/wiki/Open/closed_principle
http://www.objectmentor.com/resources/articles/ocp.pdf
http://www.oodesign.com/open-close-principle.html