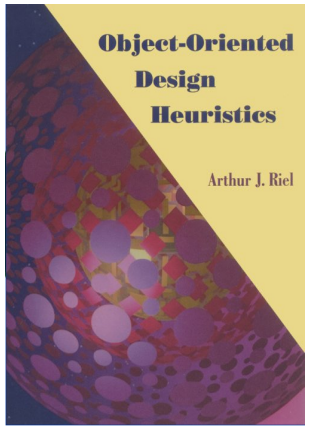
Object-Oriented Design in a Nutshell

We have a new site!

- OOSE Home: loose.upt.ro/~oose/index.html
- **GOOD: loose.upt.ro/~oose/good/index.html**
  - **labs will be added soon**
  - **Resources will be added as well**

Bibliography



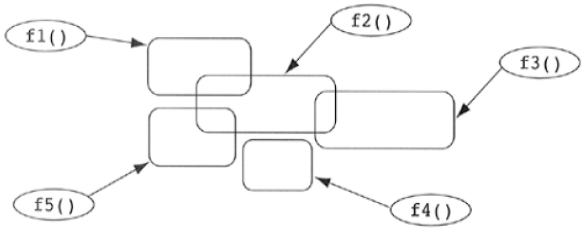
The Object-Oriented ... Hype

- What are object-oriented (OO) methods?
  - OO methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures
  - In general, OO methods are characterized by structuring the system architecture on the basis of its *objects* (and classes of objects) rather than the *actions* it performs
- What is the rationale for using OO?
  - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time

Use it for **large systems**
Use it for **systems that change often**

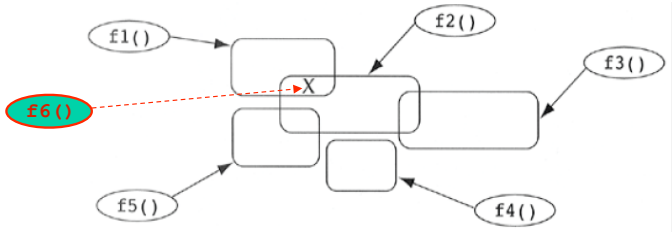
Topologia Aplicatiilor Orientate pe Obiecte

Topologia Orientata pe Actiune



- Functiile sunt centrale
- Datele sunt determinate de functionalitatea de implementat
  - apar numai cand o cere functionalitatea

Cine Stie de Cine?



- Din orice functie putem sti ce date foloseste
- Reciproca nu e valabila....
  - o data poate fi folosita in mod neasteptat de o terta functie
  - .... cu rezultate nefericite

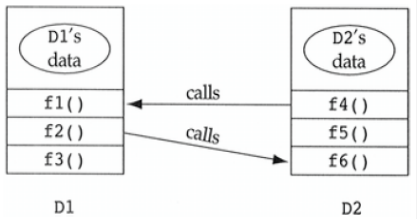
Cand Are Succes Topologia Orientata pe Actiune?

- Fiecare structura de date e plasata intr-un fisier si ....
- ... PRIN CONVENTIE nu este utilizata decat de acolo.

➔ Clasa din POO e substituita de o **conventie**

Programarea prin conventie dauneaza grav sanatatii!

Topologia Orientata pe Obiecte



- Revolutia datelor...
  - descompunerea sistemului se face functie de “clustere” de date
- Descompunerea OO =
  - o colectie descentralizata de “gramajoare” de date +
  - interfete bine definite

Clasele Omnipotente (God Class [Riel96])

- Tentatia de a avea un mecanism centralizat de control
- Apar clase omnipotente care fac majoritatea lucrurilor
  - Acapareaza date de la cateva clase “satelit”
  - Clasele “satelit” nu prea au nici un fel de comportament
- Cum le evitam?

Reguli de Aur

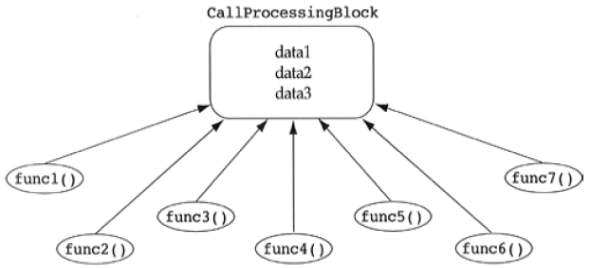
Distribuie inteligenta unui sistem cat se poate de uniform intre principalele clase din sistem

Nu scrie clase omnipotente (god class) in sistemul tau.
Fii suspicios fata de clasele care se numesc: *Driver*, *Manager*, *System*, *Subsystem* etc.

Fii atent la clasele care au multe metode accesor (get/set) declarate in interfata. Acesta poate fi un semn ca datele si functionalitate sunt in locuri diferite!

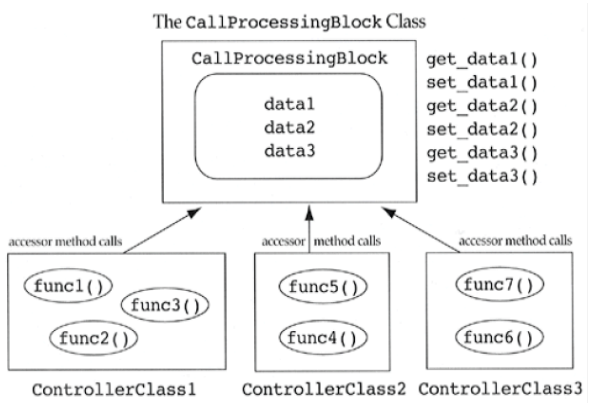
Fii atent la clasele care au bucati de functionalitate necoeziva, adica seturi de metode fara o corelatie semantica clara.

Clasele Omnisciente (God Class – Data Form [Riel96])

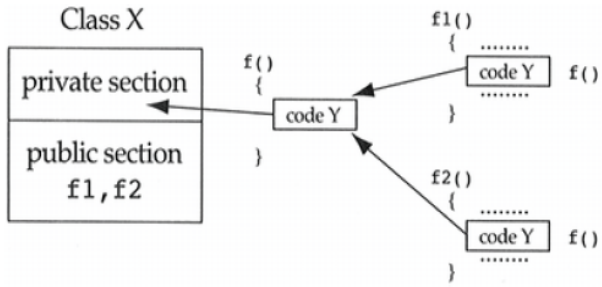


- migrarea unor sisteme mostenite scrise intr-o topologie orientata pe actiune (ex. sistem de telefonie)

Prima Solutie de Migrare



Ascunderea Informatiei



Ascunderea Informatiei: Exemplu

Ceasul desteptator...

- Cati stiti sa montati un ceas desteptator daca va dau toate piesele?
- Cati stiti sa folositi un ceas desteptator?

De ce stii sa folosesti un ceas desteptator pe care nu l-ai mai vazut?

- aceeasi **interfata**
- nu conteaza implementarea (“piesele”)

Problema cu Metodele get/set

- Metodele accesor sunt ok daca atunci cand se modifica datele corespunzatoare lor nu trebuie modificate si ele.
  - Altfel ele nu “ascund” datele
- Metodele accesor creaza suspiciunea ca functionalitatea corespunzatoare datelor e in alta parte

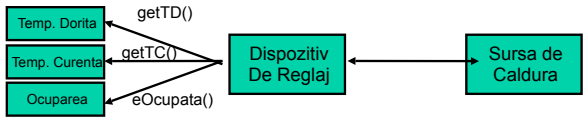
Doua Întrebări de Aur:

Cand scrii o metoda get/set intreba-te doua lucruri:

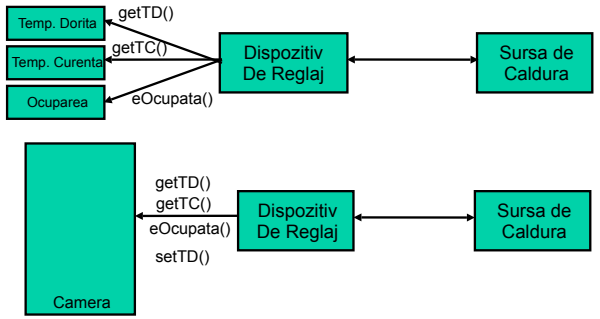
1. **Pentru ce** are cineva nevoie de acel get/set?
2. De ce nu face clasa mea acel lucru?

Exemplu: Distribuirea Inteligentei intre Clase

- Exemplu – Sistem de incalzirea a unei camere
  - Dispozitiv pt. selectarea temperaturii dorite in camera
  - Senzor de temperatura curenta
  - Senzor de ocupare
    - Daca camera nu e ocupata temp. curenta poate scadea cu pana la 5 grade sub cea dorita, altfel trebuie imediata reglata

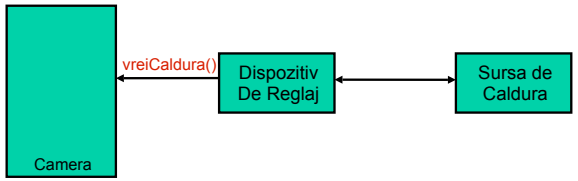


Instalatie de Climatizare a Unei Camere…



- Cum aplicam întrebarile de aur la exemplul nostru?

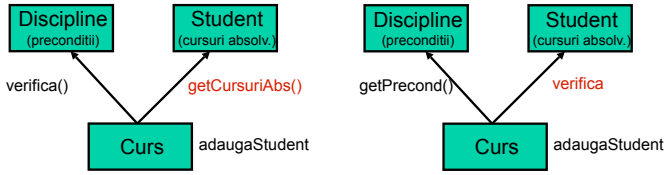
Solutia



- Camera stie cel mai bine daca are nevoie de caldura
- Cine vorbeste cu cine?
  - Camera spune DispozitivuluiDeReglaj ca vrea caldura
  - DispozitivuluiDeReglaj intreaba periodic Camera (polling)
  - Functie de “politica de schimbare” si de detaliile de hardware se poate decide una din cele doua variante

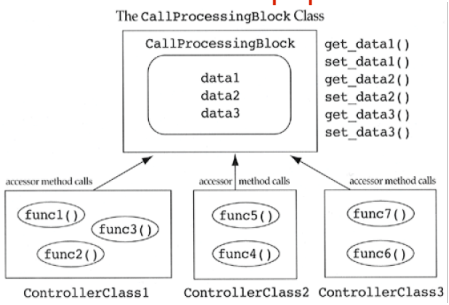
Cand e ok sa folosim metodele accesor?

1. Cand o anumita clasa implementeaza o “politica”(regula) de colaborare intre 2 sau mai multe clase
   - Mai mult nevoie de “**get”** decat de “set”
   - Exemplu: inregistrarea unui student la un curs



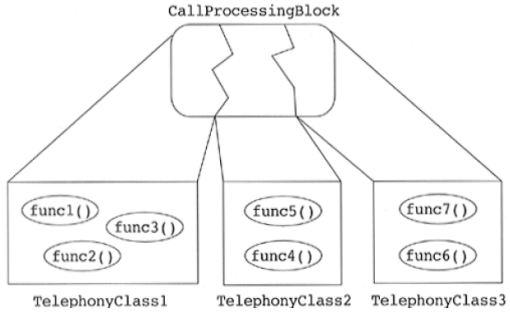
2. Un model orientat pe obiecte interactioneaza cu un UI
   - Nevoie de metode get, dar si de **set**

Care este de fapt problema?



Problema metodelor accesor (get/set)
nu este atat ca expun reprezentarea interna,
ci faptul ca ele reprezinta unui **semnal de avertisment**
ca **datele** si **functionalitate** nu stau in acelasi loc

Adevarata Solutie

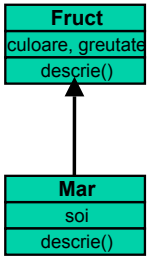


Relaţia de Moştenire

De ce Moştenire?

- Nevoia de **ierarhizare**
  - modalitate de gestionare a complexităţii
  - gruparea abstractiunilor inrudite semantic
    - **Generalizare**: dau factor comun caractersticile comune
  - Complexitate e “invinsa” prin aceea ca toate clasele derivate pot fi privite de catre clienti ca fiind “sub caciula” clasei de bază
- Nevoia de **reutilizare**
  - se poate defini o clasa derivata in termenii clasei de baza
    - **Specializare:** precizez incremental doar ce o face diferita

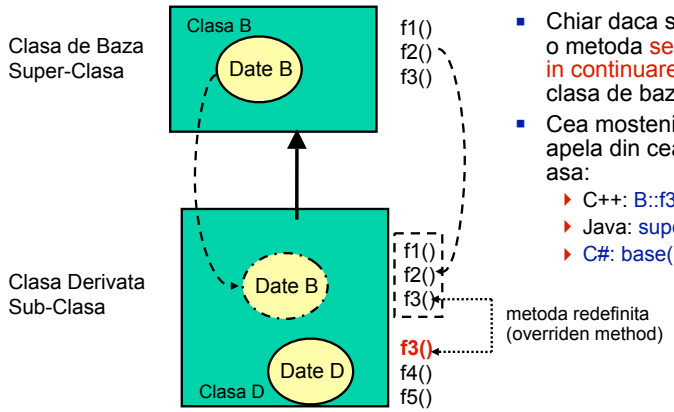
Mere, Pere si Alte Fructe...



- Fiecare obiect trebuie sa se “descrie”
  - Problema: cum punem atributele?
    - private? protected?
- Doua minciuni
  - I. Datele astea nu se schimba niciodata
    - Nu uitati: datele sunt extrem de volatile
  - II. Ierarhia asta de clase contine putine clase
    - Nu uitati: relatia de mostenire e tranzitiva ;-)

Toate datele dintr-o clasa de baza trebuie sa fie private!
Nu folositi date protected! Incalca incapsularea!

Redefinirea Metodelor



- Chiar daca se redefineste o metoda se mosteneste in continuare metoda din clasa de baza!
- Cea mostenita se poate apela din cea redefinita asa:
  - C++: B::f3();
  - Java: super();
  - C#: base();

Cum Dam “Factor Comun” intre Clase?

Daca doua sau mai multe clase au **numai** date (atribute) in comun, acele date trebuie plasate intr-o clasa ce va fi **continuta** (nu mostenita!) de aceste clase.

Daca doua sau mai multe clase
au in comun atat date (atribute) cat si functionalitate (metode),
atunci aceste parti comune trebuie sa fi plasate intr-o clasa ce va fi **mostenita** de clasele initiale.

Daca doua sau mai multe clase
au in comun doar o interfata comuna (adica doar metode),
acestea ar trebui date factor comun intr-o clasa de baza
doar daca acestea vor fi utilizate polimorfic.

Doua cazuri de mostenire...

- **Mostenirea de Clasa**
  - am date si functionalitati comune intre mai multe clase si le “dau factor comun” intr-o clasa de baza
  - clasele derivate mosteni si folosi acele metode/date comune fara a mai trebui sa fie declarate/implementate in clasele derivate
    - scade complexitatea
- **Mostenirea de Tip**
  - vreau sa rafinez (modific, redefinesc) functionalitatea (comportamentul) unei metode din clasa de baza
  - **suprascriu** (override) metoda din clasa de baza cu propria mea varianta.
    - **ATENTIE**: metoda in clasa derivata trebuie sa aiba aceeasi signatura

Polimorfism

- Problema legarii implementarii
  - ce cod (corp de functie) sa execut atunci cand este apelata o functie?
- Legare Statica
  - *uita-te la obiectul prin care se apeleaza metoda si vezi de ce tip e declarat*
  - apeleaza metoda din declarata a obiectului
  - se stabileste **la compilare**
- Legare Dinamica
  - *uita-te la obiectul prin care se apeleaza metoda si vezi a cui instanta este*
  - apeleaza metoda din clasa reala a obiectului
  - se stabileste **la executie**

Polimorfism (2)

- Comportamentul promis in interfata publica a superclasei
- implementata de subclase
  - In modul specific necesar fiecarei subclase
- De ce e asta important?
  - Asigura flexibilitate
    - Logica de nivel inalt definita in termenii unor interfete abstracte
    - Depinzand de implementarea specifica a oferita de subclase
    - Subclase noi pot fi *adaugate fara schimbarea* logicii de nivel inalt

Stabil in CE trebuie facut,
flexibil in CUM este facut!

Tiparul **Template Method**