* **Refactoring** – proces de a face modificari la cod asa incat ceea ce face el propriu zis sa nu fie modificat, dar structura interna e imbunatatita. Se fac modificari strucurale mici si se testeaza ca ele sa nu modifice comportamentul la code asa cum el era pana la modificari.
* **Anti**-**pattners** - practici rele de code

**Comments**

* Nu e bine sa nu avem de loc comentarii
* Totusi, sa avem prea multe comentarii tot e rau
* Totusi, daca avem un code care descrie un cod foarte complicat, cel mai probabil avem un design rau.

**Duplicated Code**

* **Duplicated Code** – apare cand avem linii de code identice, poate cu mici diferente, in mai multe parti din program
* Duplicarea de code e foarte rea, deoarece, la un moment dat, am putea hotara sa mai modificam ceva in acest cod care se repeta, si va trebui sa il modificam peste tot unde apare.
* Mai bine folosim Generalization
* Avand sa facem modificari doar intr-un singur loc usuram munca si evitam posibilele erori facute prin a scapa code, a face greseli etc.

**Long Method**

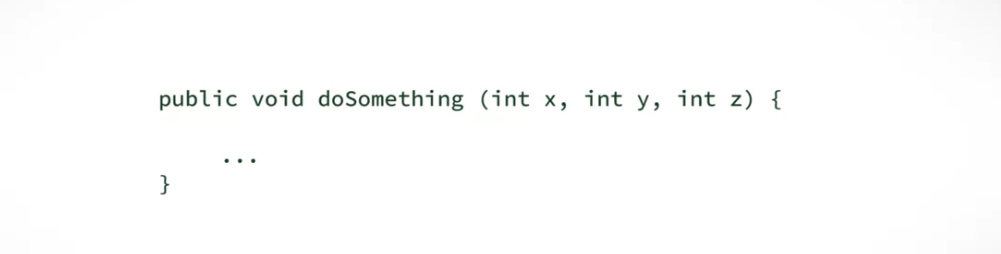
* O metoda prea lunga e un semn ca metoda face prea multe lucruri sau ca e prea complexa
* Totusi, nu exista o lungime care spune ca metoda e suficient de lunga sau nu. Unele metode pot fi lungi, de ex, pentru user interface, si asa e si normal sa fie in acest caz
* Totusi, e recomandat ca metoda sa nu fie mai mare decat putem vedea pe ecran, deci maxim vreo 15 linii

**Large Class**

* Sa avem clase prea mari tot e rau
* O clasa lunga are adesea multe responsabilitati, si asta e rau.
* Totusi, nu e bine nici sa avem clase foarte mici. Acestea sunt Data Classes.
* **Data Classes** – sunt clase ce contin doar date, inclusiv si getteri si setteri, dar nu au nimic functiuni propriu zise

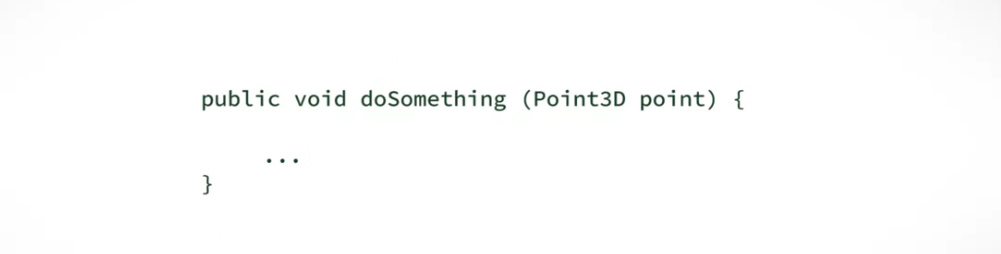
**Data Clumps(gramezi)**

* **Data Clumps** – grup de date care apar impreuna in instance variables sau class, sau parametrii metodei
* De ex, daca avem o asa metoda



Daca sunt mai multe metode ce folosesc x,y,z atunci ar fi mai logic sa avem deja un obiect cu aceste coordonate decat sa le trimitem mereu toate 3

* Asa e mai logic



**Long Parameter List**

* O metoda cu multi parametri e greu de utilizat
* Adesea, mai bine cumva incapsulam parametrii comuni in clase

**Divergent Change**

* **Divergent Change** – situatie cand trebuie sa modificam clasa in difertie feluri de mai multe ori

**Shotgun Surgery**

* Apare atunci cand daca vrem sa facem o modificare la o parte din code, gen in clasa, trebuie sa incepem a face modificari deja si in multe alte parti, gen in multe alte clase
* Uneori chiar si cu un design bun, nu il putem evita

**Feature Envy**

* Apare cand avem o metoda care e mai interesata de detaliile din alta clasa decat de detaliile pe care le are propria clasa
* Metoda ar trebui mai bine sa fie in acea clasa deja

**Innapropriate Intimacy**

* Apare cand clasele au metode ce se apeleaza prea des unele pe altele si pot aparea asa si recursii infinite
* Mai bine doar una sa apeleze cealalta sa sa punem metodele undeva intr-o clasa separata

**Message chains**

* Fie clasa A si B. Clasa A are o metode getB() si clasa B o metoda getC() si asa din A ajungem in C daca vrem



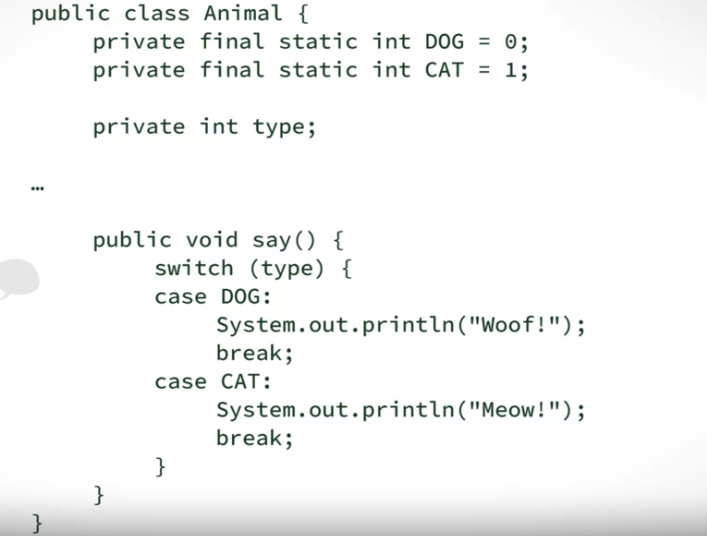
Asta e un message chain. Violam legea Least Knowledge

**Primitive Obsession**

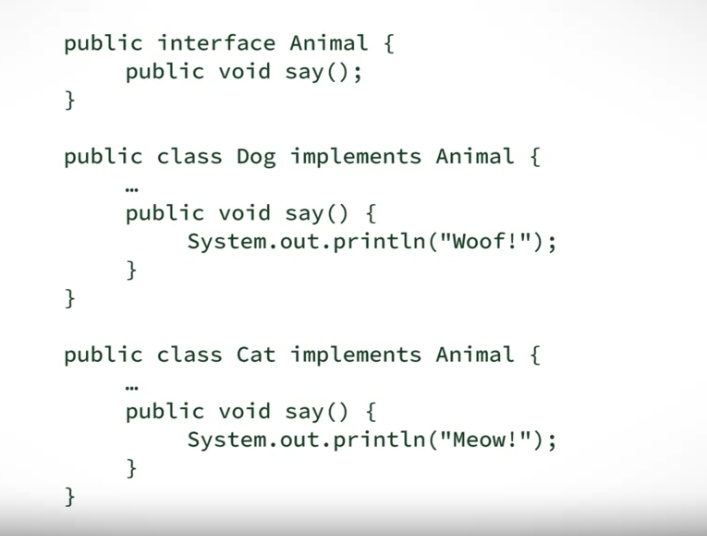
* Apre cand ne bazam prea mult pe date primitive
* Ele trebuie puse cand chiar sunt necesare
* Utilizarea loc excesiva arata ca lipseste abstraction
* De multe ori, folosirea lor excesiva face ca sa nu fie clar de ce le folosim
* Gen, postal code este un String. Dar, daca vom folosi pur si simplu un string ca variabila, nu va fi clar mereu ca asta e un postal code. Daca vom avea o clasa PostalCode cu un string in el ,va fi mai clar

**Switch Statements**

* Folosirea lor uneori arata ca nu facem ceva bine
* DE ex:



Mai bine folosim polimorfism



**Speculative Generality**

* Apare atunci cand adaugam ceva si o facem caci ne spunem “Poate vom avea nevoie de asta in viitor”
* Asta inseamna ca cream clase sau metode care nu sunt necesare acum, ci le cream ca poate vor fi necesare in viitor.

**Refused Bequest(mostenire)**

* Apare cand o clasa mosteneste de la alta si nu are nevoie de ceea ce mosteneste