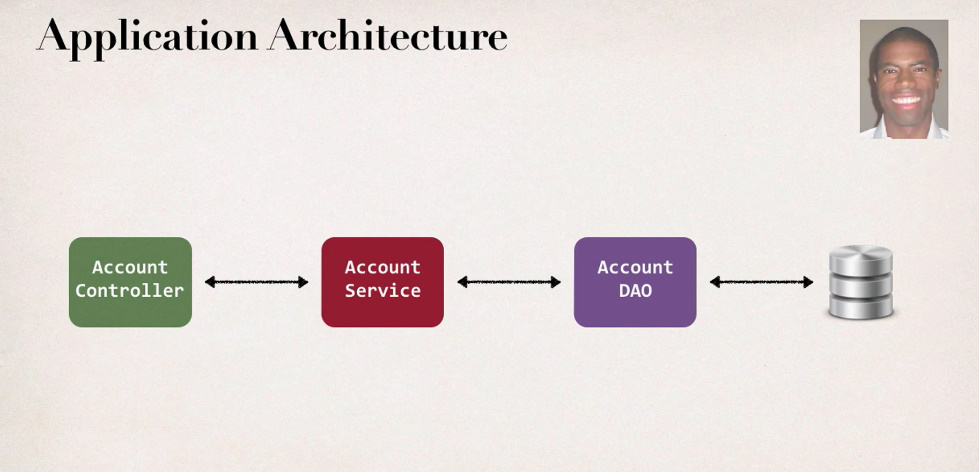
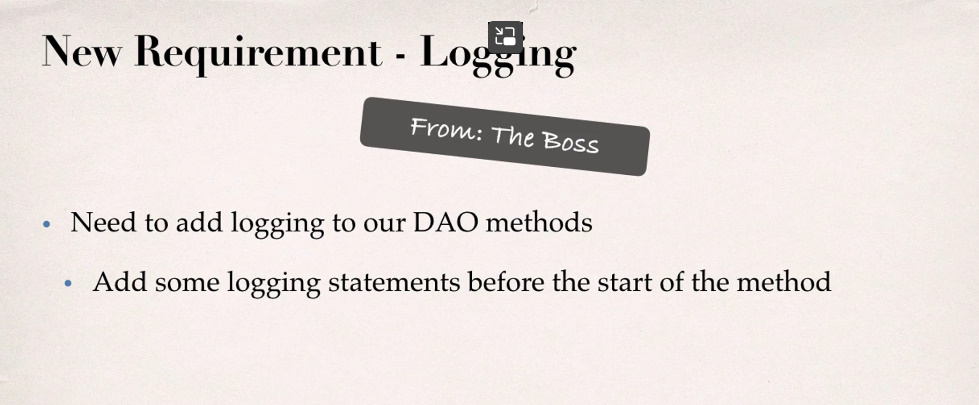
AOP – Aspect Oriented Programming

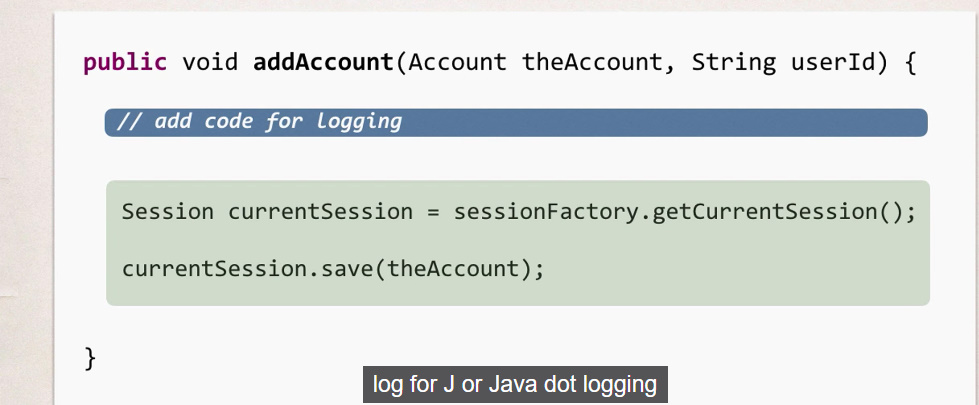
Exemplu de proiect folosit:

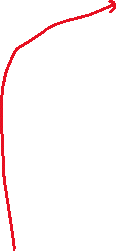


**Necesitate**

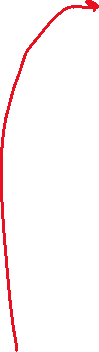
* Posibil ca sa avem nevoie sa adaugam niste logging methods intr-o metoda din DAO inainte de inceperea ei





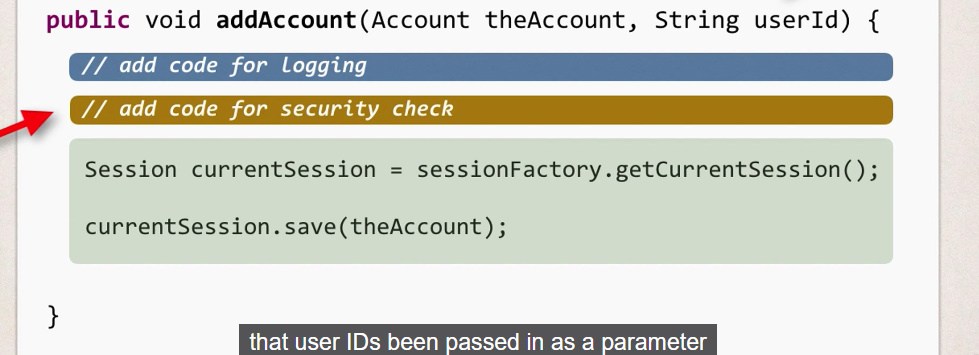


**Putem folosi Log4j, outputuri in consola etc. ca loguri**

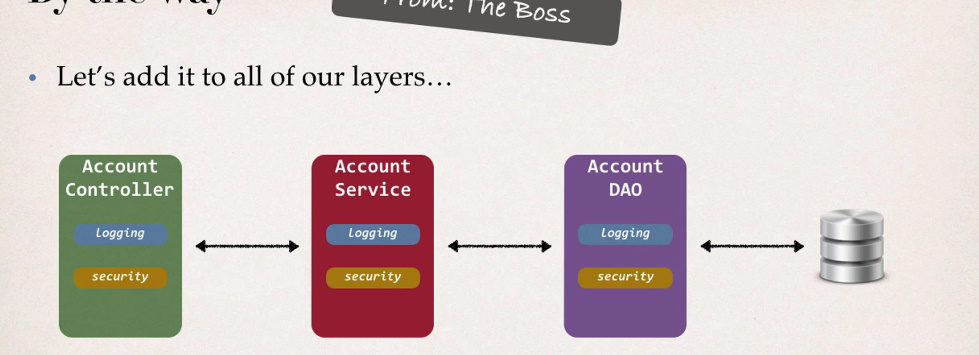


* Acum, s-ar putea sa avem nevoie sa adaugam ceva securitate in metoda, gen sa vedem daca userul este autorizat inainte sa se foloseasca DAO method



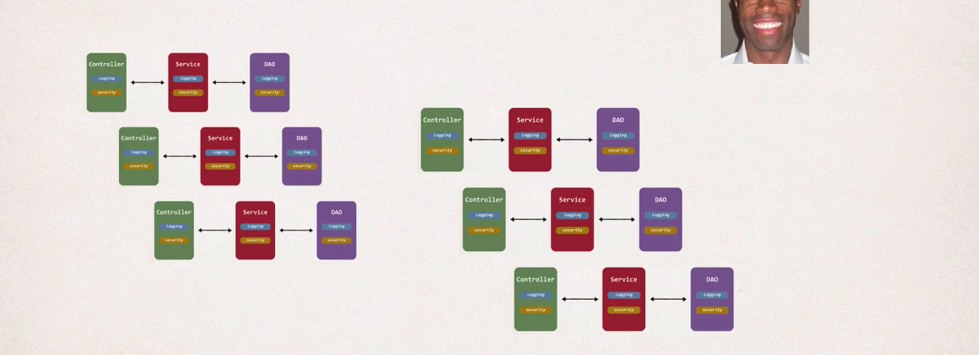


* Apoi poate aparea necesitatea de a adauga logging si security la toate clasele din DAO si inca in Service si Controller



Tot ce am putea face e sa copiem acelasi cod din DAO in restul layerelor, dar asta e foarte rau, asa cum pot fi mii de linii de cod.

Si apoi poate va trebui sa facem asta la tot sistemul:



**Probleme ce apar**

1. Fiecare metoda de logging si security e doar pentru o anumita metoda din DAO, Service etc. si asta e foarte rau
2. Daca vom avea nevoie sa modificam logging sau security code, va trebuie sa o facem in toate clasele, si asa cum am copiat pur si simplu codul din DAO, va lua un timp imens.
3. Am putea gen crea o clasa cu toate aceste metode si restul sa le mosteneasca, dar mostenirea multipla nu e permisa.

**Ce este AOP**

* Tehnica de programare bazata pe conceptul de aspect
* Aspectul incapsuleaza cross-cutting logic/concerns

concerns=logic

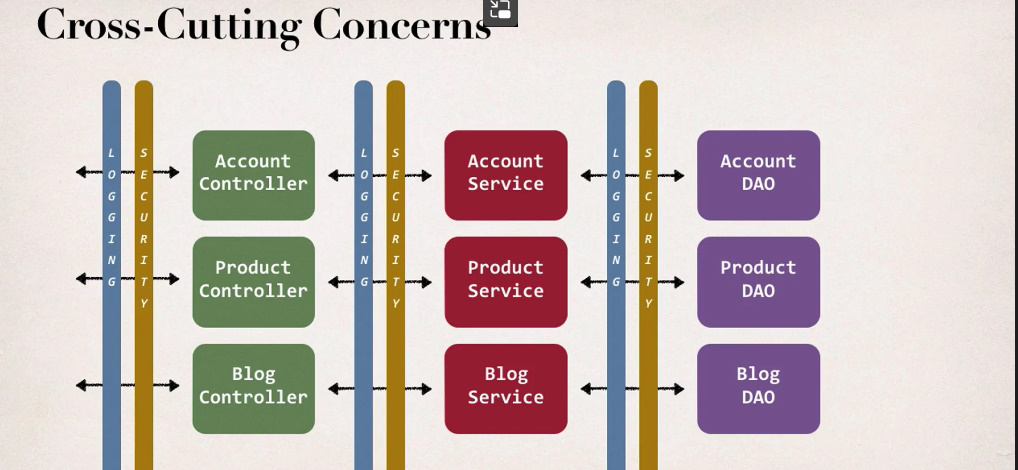
concern/logic – basic infrastructure code de care toate aplicatiile au nevoie

cross – a traversa

cut – a taia

cross-out- transversal

transversal – care taie de-a curmezisul.



Deci, aici avem un cod pentru Logging si Security, dar care poate fi incapsulat in fiecare clasa, asa fiecare clasa il poate folosi cand e necesar, deci e ca si cum taiam layers cu acest service, de aia se cheama cross-cutting, adica taiam calea intre ele prin acest logic code.

**Aspects**

* Un aspect poate fi folosit in multiple locatii. Aspectul e doar o clasa, ce poate fi aplicata oriunde, in dependenta de configuratie

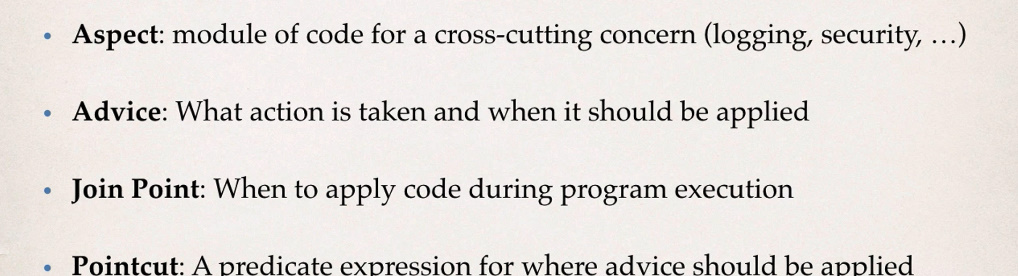
**AOP Terminology**

**Aspect** – modul de cod pentru cross-cutting concerc/logic

**Advice** – ce actiune e luata si cand se aplica

**Join Point** – cand sa se aplice codul la rularea programului,la ce nivel(Method level, constructor level etc.)

**Pointcut** – expresie predicat pentru unde trebuie sa fie folosit advice



Advice type:

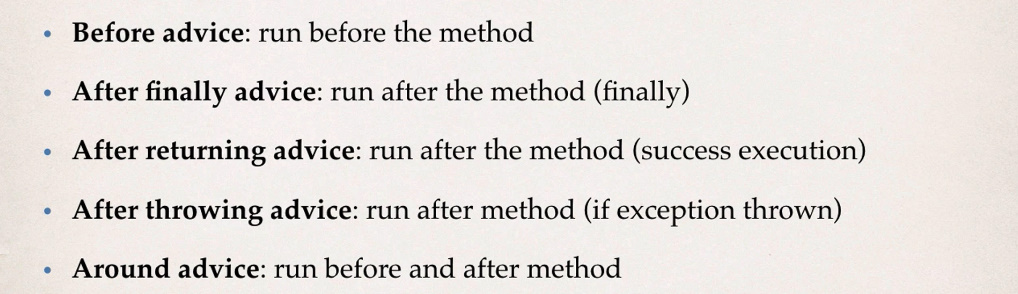
**before advice** – ruleaza inaintea la metoda

**after final advice** – ruleaza cand metoda termina, ca finally

**after returning advice** - ruleaza cand metoda se executa cu succes, dupa return adica

**after throwing advice** – dupa executarea metodei daca apare o exceptie

**around advice** – ruleaza pana si dupa metoda



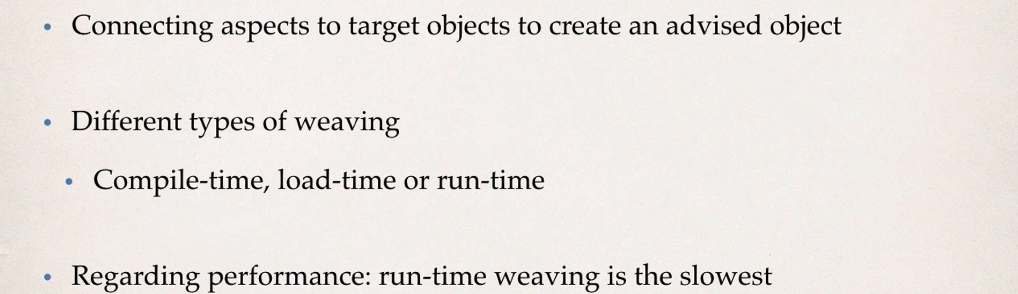
**weaving** – proces de conectare a aspectelor cu alte obiecte pentru a crea un proxy object potrivit.

Tipuri:

**compile time**

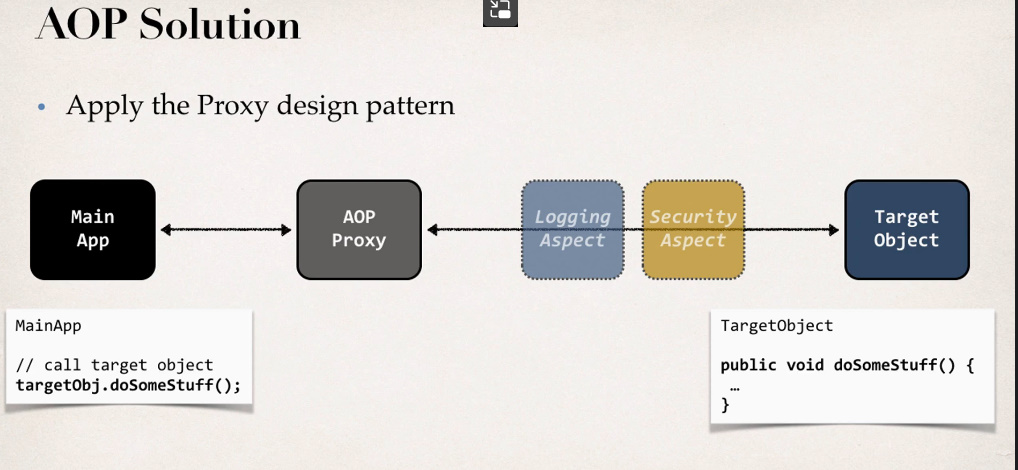
**load time**

**run time**



**AOP Solutions**

* O solutie pentru AOP este Proxy Design Pattern





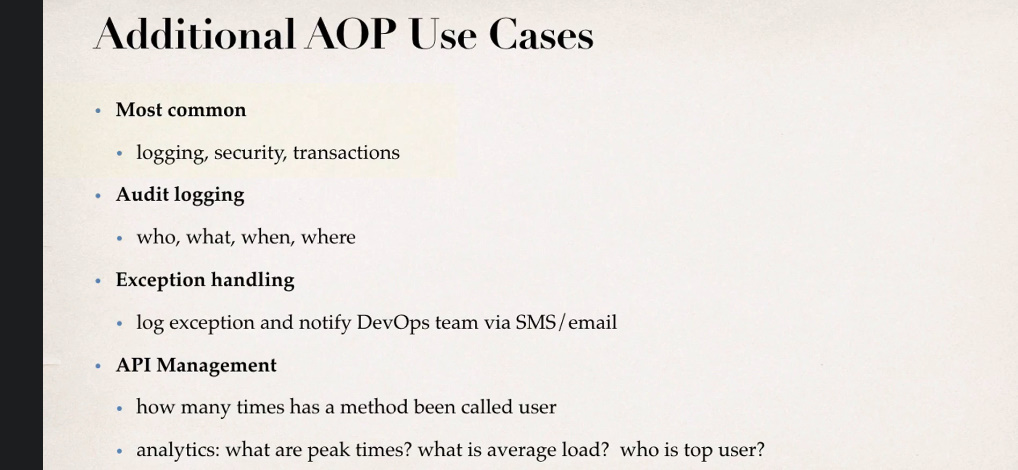
Deci, folosim un TargetObject ca el sa faca diferite chestii. E un obiect simplu, nu are nimic cu AOP.

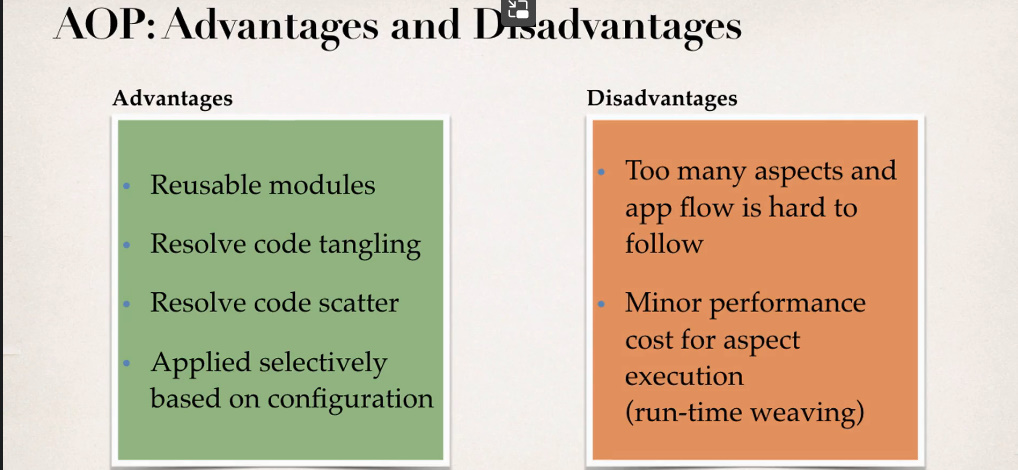
Aplicatia habar nu are de proxy sau AOP, asa cum ea doar face apeluri la metode din alt obiect.

* E ca si cum am suna un prieten, asa cum face in MainApp, dar telefonul cu care facem asta e monitorizat, de aceste Loggin si Security aspectcare pot lua decizii in dependenta de ce spunem.
* Proxy e obiectul folosit de AOP pentru a verifica orice metoda ce ruleaza. Adica toate metodele trec prin el.

**Beneficii ale AOP**

1. Codul pentru aspecte e definit intr-o clasa
2. Nu trebuie sa punem codul in toate clasele
3. Codul e usor de modificat
4. Poate fi aplicat doar unde cerem noi in proiect

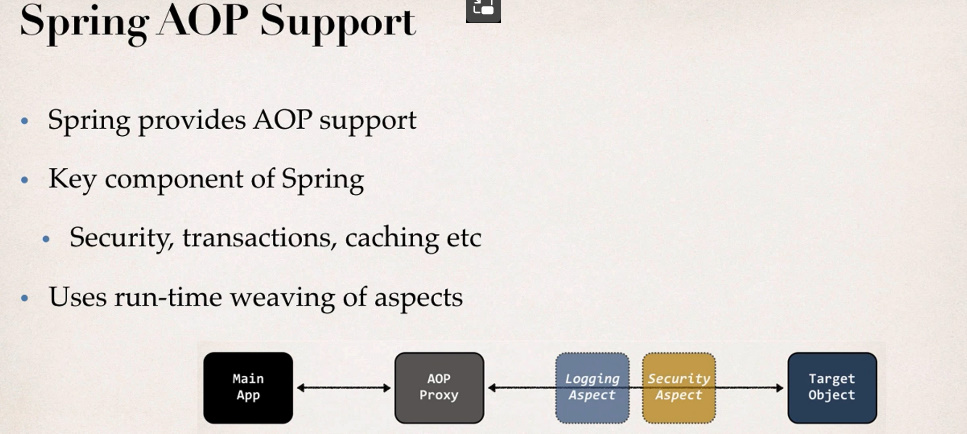




Prea multe aspecte pot incetini programul

**AOP Frmeworks**

* Spring AOP
* AspectJ
* **Spring AOP** este deja implementat automat in Spring framework, cand facem tranzactii



* **AspectJ** – framework

