**Modul** – unitate de code ce contine clase si metodele din ele.

* Daca Design de noi este prea complex, va fi foarte greu de inteles si modificat.

Iata de ce, e necesar cumva sa evaluam cat de bun este desig, adica deisgn complexity

* Complexitatea la Design poate fi determinata si imbunatatita cu:

- Coupling

- Cohesion

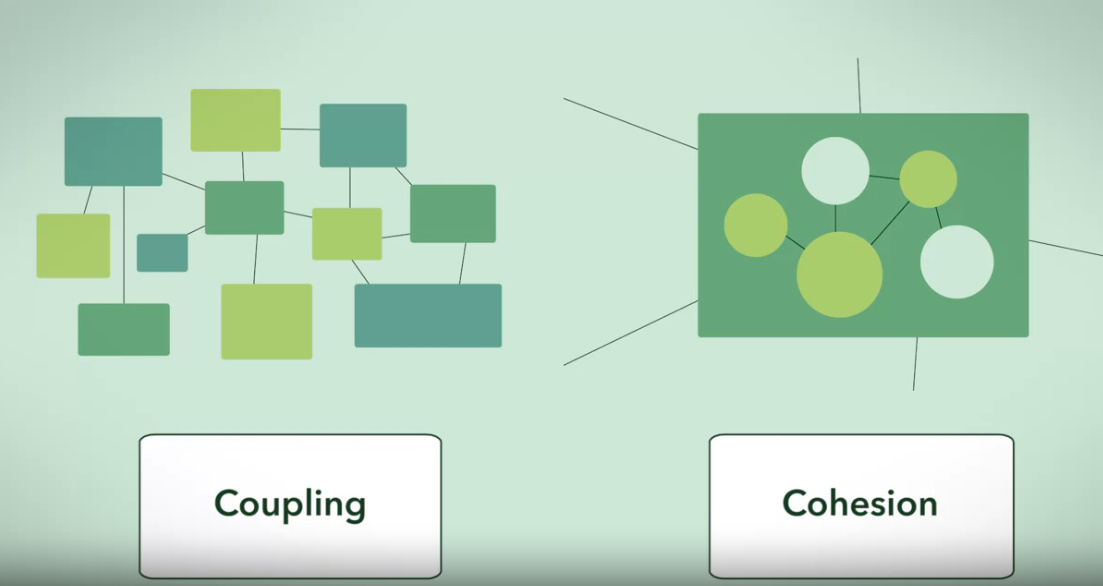
- Separation of concerns

- Information Hiding

- Conceptual Integrity

**Coupling and Cohesion**

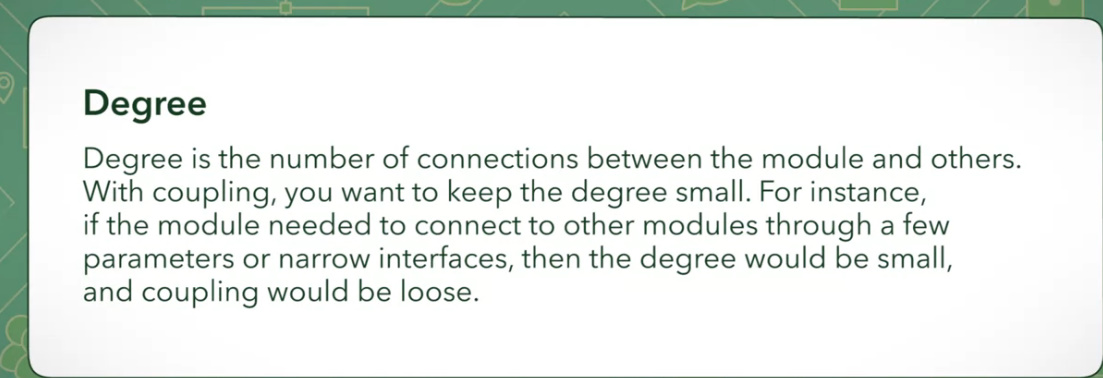
* Sunt unitati de a masura cat de complex este design
* **Coupling** – descrie complexitatea dintre un modul si alte module.
* **Cohesion** – descrie complexitatea dintr-un modul.



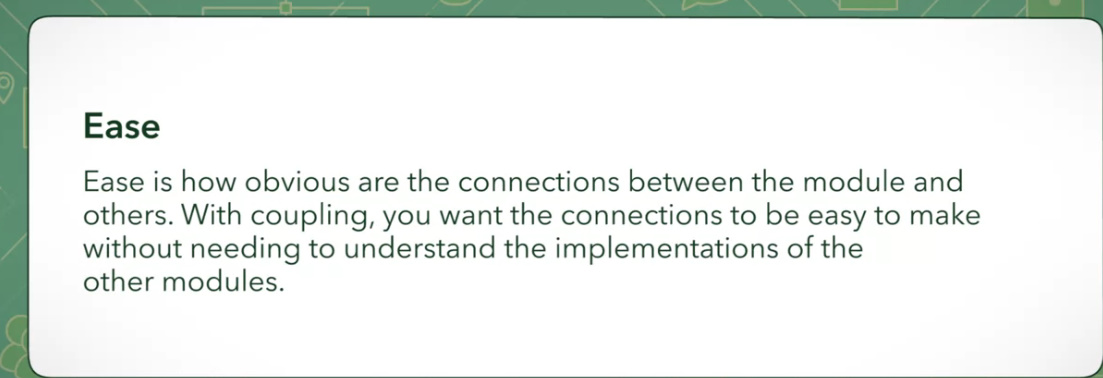
**Coupling**

* **Coupling** – indica complexitatea de a conecta un modul la alte module
* Daca un modul este strans legat de anumite module, spunem ca el este **Tightly Coupled**(strans cuplat). De ex, o clasa are ca fields nu interfete, ci clase concrete, si aici pot aparea probleme.
* Daca modulul nu este strans legat de anumite module, si poate fi conectat usor la oricare altele, el este **Loosely Coupled**(lejer cuplat). Daca o clasa are ca field interfete, ele pot fi conectate usor la orice implementari.
* Scopul nostru este de a face ca module sa fie Loosely Coupled
* Cand evaluam coupling la un module, consideram urmatoarele unitati:

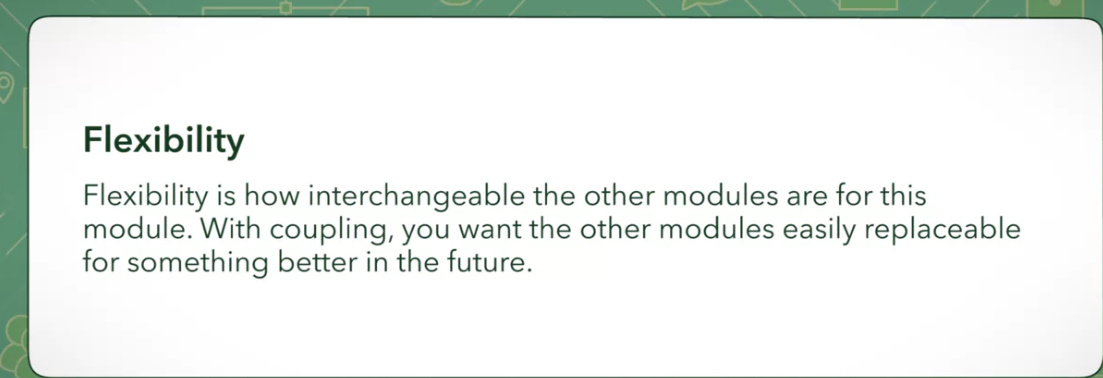
- **degree –** este numarul de conexiuni dintre un modul si altele. Degree trebuie pastrat mic.



- **ease**(usurinta) – arata cat de evidente sunt conexiunile dintre modul si celelalte. Conectiunile trebuie sa fie usor de facut, fara a trebui sa intelegem cum alte module functioneaza.



- **flexibility –** arata cat de interschimbabile sunt alte module pentru acest modul. Cel mai bine este ca modulele sa poata fi inlocuite cu altele in viitor, pentru acesta. Iata de ce, folosim interfete la fields.

****

**Cohesion**

* **Cohesion** – arata complexitatea dintr-un modul si reprezinta gradul de claritate al responsabilitatilor unui modul.

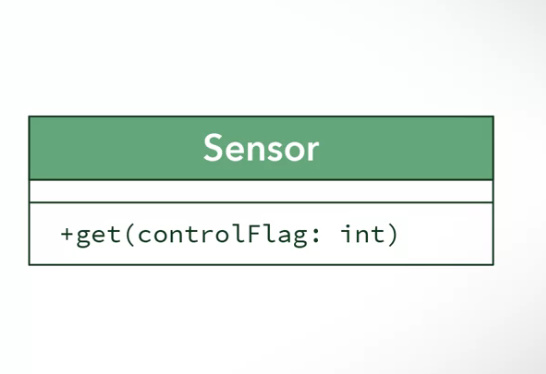
****

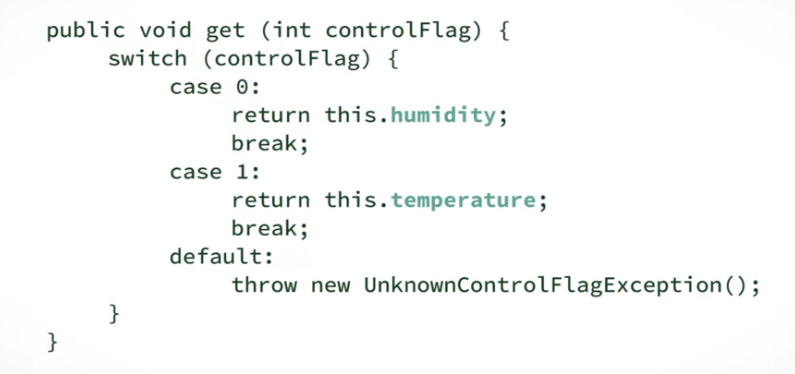
Daca modulul face un singur task, clar, atunci modulul are High Cohesion

Daca face mai multe taskuri sau neclare, atunci are Low Cohesion

**Exemplu**

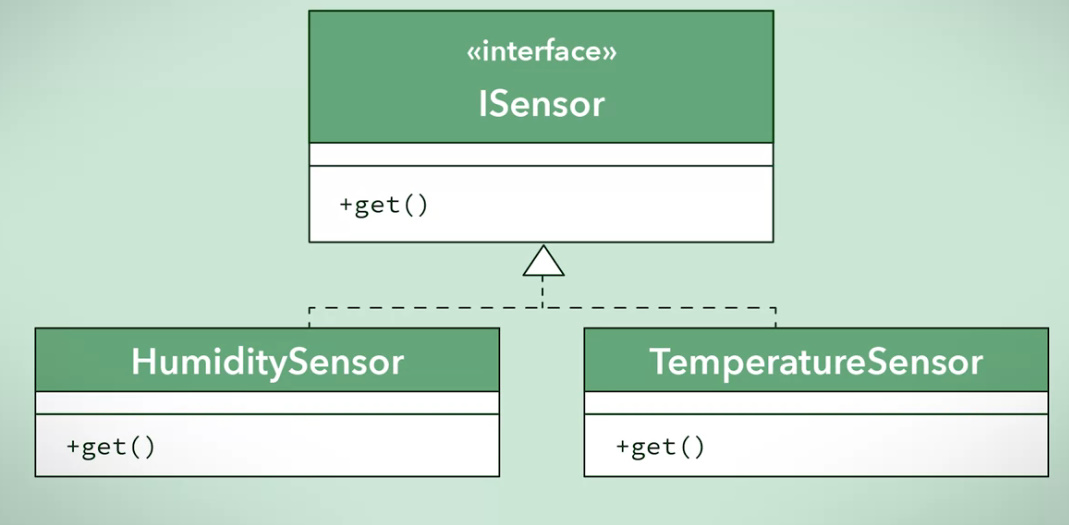
Fie clasa Sensor





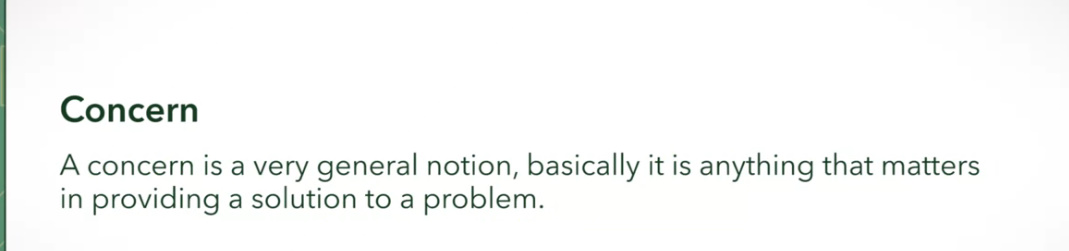
Acum, evaluam clasa in dependenta de Cohesion si Coupling

* Deaorece clasa nu are un singur task, si clar, are **Low Cohesion**
* Deoarece nu este clar cum anume functioneaza metoda get(), caci trebuie sa deschidem clasa, sa vedem ce returneaza 0 si 1, avem **Tightly Coupling**
* Asa deja, avem high cohesion si loosely coupling



Fiecare sensor are deja metoda lui get(), si ele implementeaza interfata Isensor

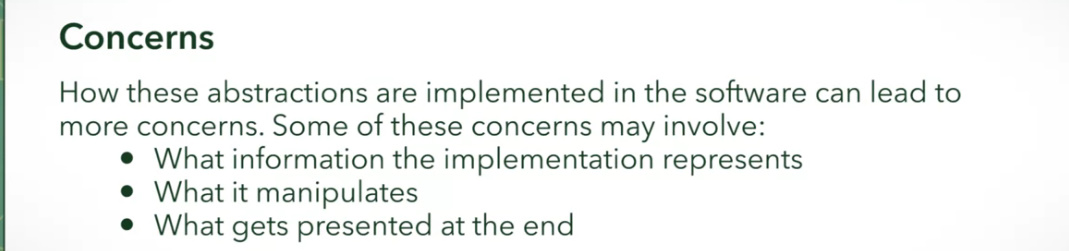
**Separation of Concerns(probleme)**



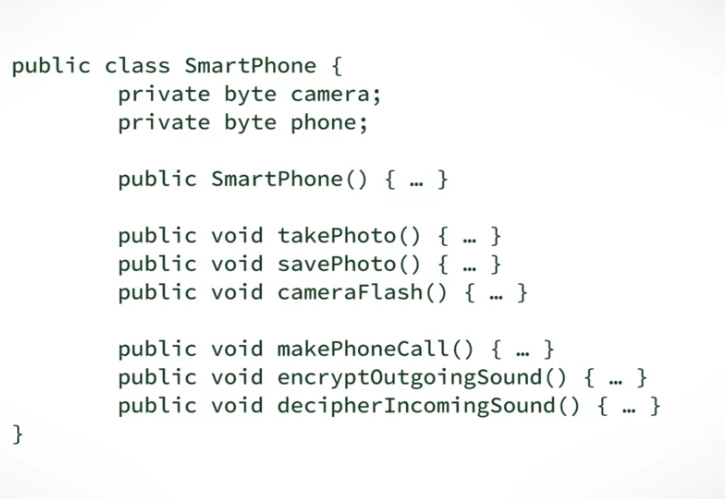
Sunt probleme concrete ce trebuie rezolvate.

De ex, un supermarket trebuie sa se ocupe de multe probleme, cum ar fi cum sa cumpere marfa, cum sa pazeazca marfa, cum sa comunice cu clientii etc. Pentru fiecare problema, exista un Departament axat pe un concern. Fiecare problema e formata din subprobleme, pe care Department corespunzator stie cum sa le rezolve ca sa rezolve in asamblu problema

* In software e la fel. Un system poate rezolva mai multe probleme si le rezolva pe toate, deci rezolva mai multe concerns.



* Intr-un software cu un design bun, fiecare concern e rezolvat separat, nu toate deodata impreuna.
* **Decomposition** este cea care participa in S**eparation of Concerns**
* **Separation of Concerns asigura principiul de Single Responsability**
* In general, pentru fiecate problema sau concern cream o clasa separata, care sa se ocupe de rezolvarea ei, si apoi unim rezultatele la toate ca rezolvam intreaga problema intr-o clasa.
* De ex, fie un smartphone. El are multe functii disponibile, ca a face poze cu camera, a suna pe cineva. Daca clasa ar fi asa:

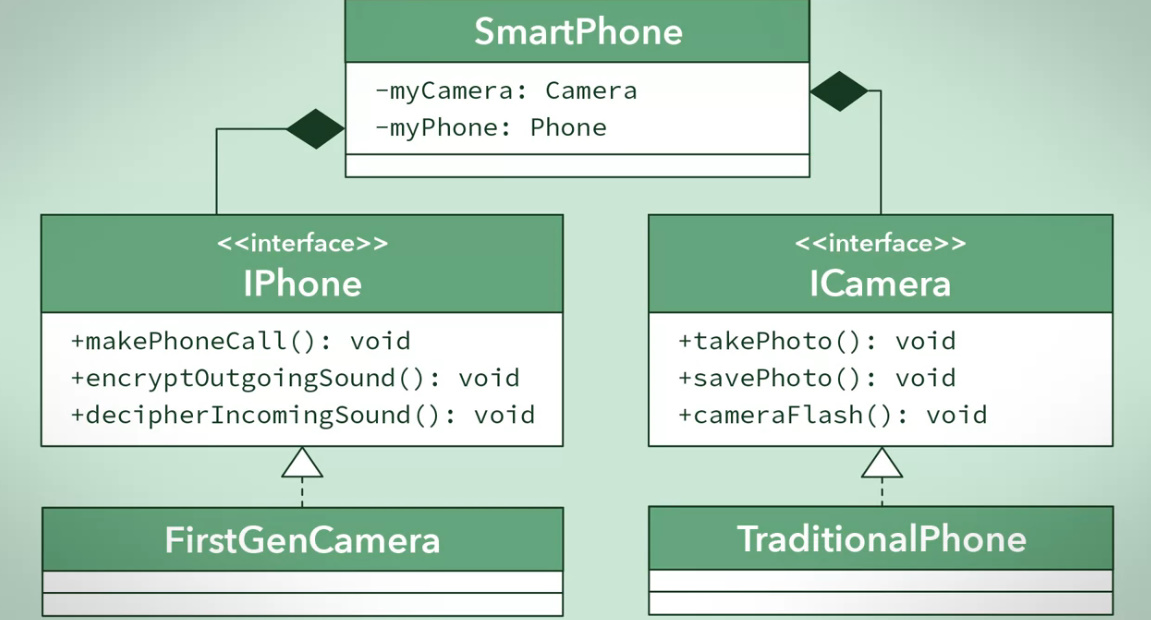


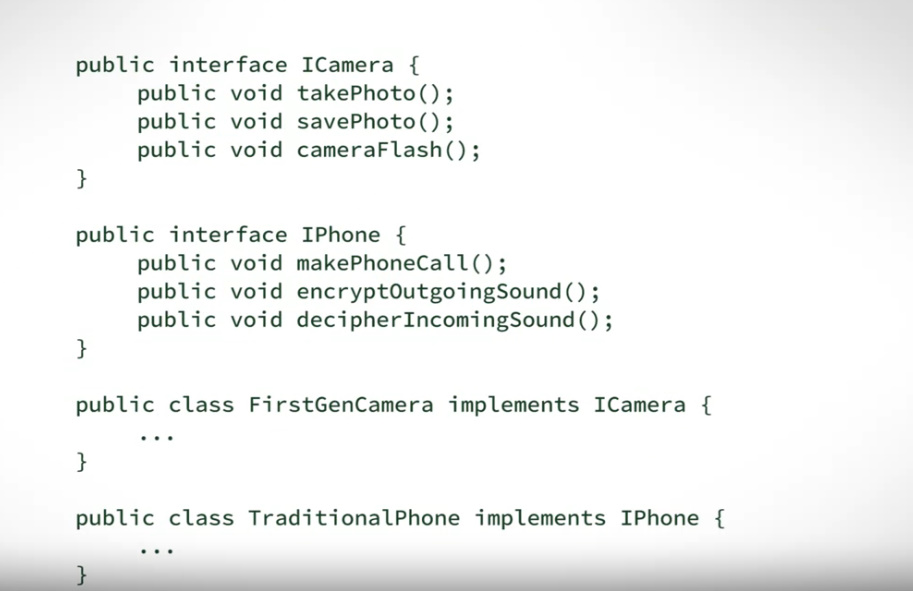
Cohesion e low, deoarece clasa are multe responsabilitati diferite, nu e flexibila si nu evident ce si cum face. Nu putem inlocui camera sau phone, si incapsulam metode ce nu au legatura de loc unele cu altele.

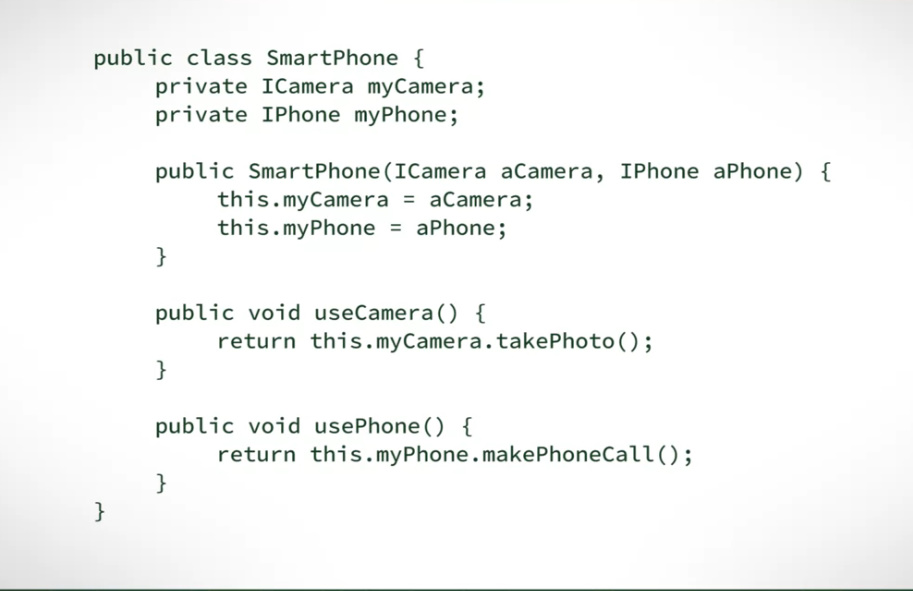
Cel mai bine e sa cream chiar o clasa pentru Camera si Phone, sau chiar interfete si mai bine si ele sa se ocupe de functiile lor.

* Clasa SmartPhone are 2 concerns:



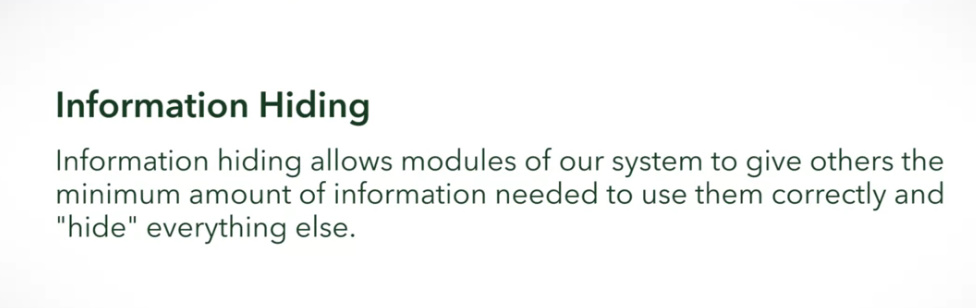






SmartPhone acum are clare responsabilitati, este flexibil cu interfete si foloseste putine fields. In plus, are doar 2 responsabilitati clare.

**Information Hiding**

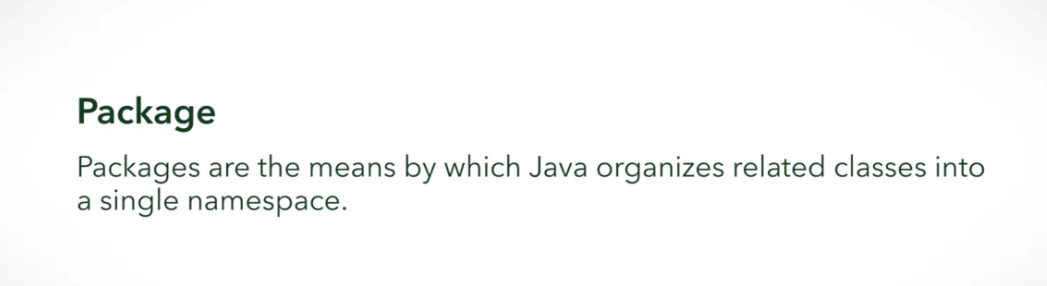


Deci, lucram cu module fara a sti concret detaliile de implementare. Asadar, lucram cu interfetele sau lucram fara a sti anumite chestii ascunse, ca cele private

* Information Hiding este asociata cu Encapsulation



Deci, interfetele permit sa ascundem implementarea, iar implementarile pot si ele ascunde detaliile de implementare



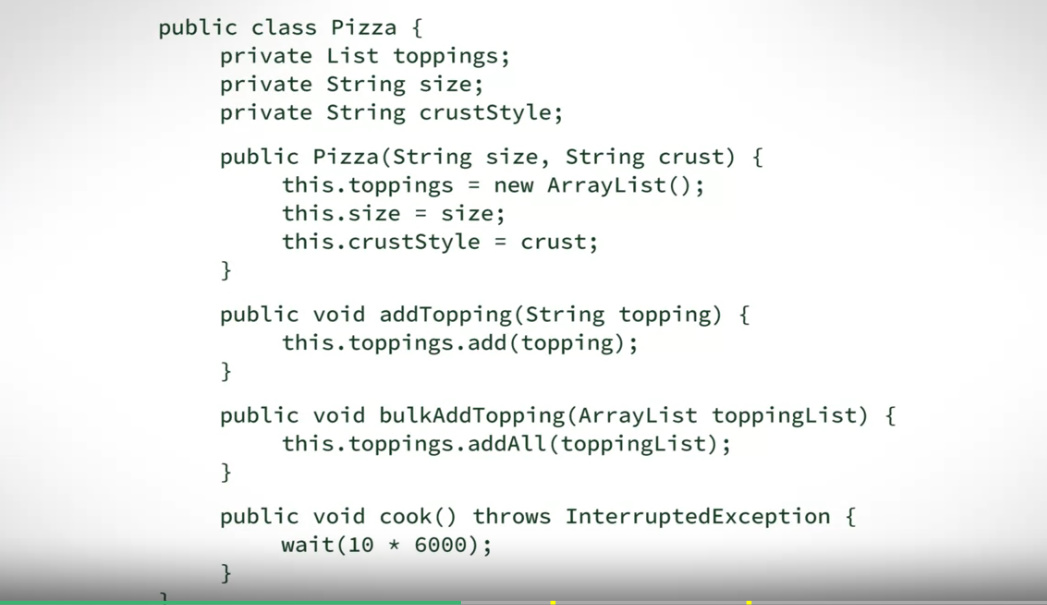
**Conceptual Integrity**

* **Conceptual Integrity** – este despre a crea consistent Software prin a crea un design si o implementare la system cat mai buna, si sa fie doar una, nu fiecare sa o aiba pe a lui.
* Development team este cel care decide cum sa faca design. De ex, toti decid sa foloseasca o anumita conventie pentru a numi clasele sau metodele sau pentru importuri. E foarte important ca toti sa lucreze cu acelasi design si principii alese si stabilite de team.
* Modalitati de a atinge conceptual integrity:

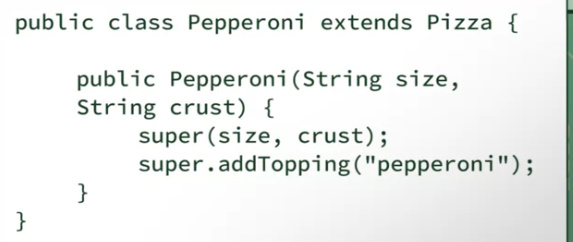
1. **Communication** – ca daily standup. sprint retrospective etc.
2. **Code Reviews** – revizuiri sistematice ale codului. Fiecare evalueaza si codul altora.
3. **Design si arhitecture bine definite**.
4. **Sa avem un grup mic care accepta commiturile in cod**

**Inheritance issues**

* Generalization si Inheritance sunt foarte greu de folosit in OOP, caci e greu uneori de inteles care din ele ar fi mai bune.
* Inheritance, daca e folosita gresit, poate crea probleme multe
* **Un semn ca inheritance e folosit gresit**, este,De ex, daca folosim inheritance pentru a crea o clasa comuna pentru altele, dar cele ce mostenesc de la ea nu adauga absolut niciun atribut sau metoda noua, atunci inseamna ca inheritance e cam deprisos. E deprisos sa folosim inheritance pur si simplu pentru a crea o clasa ce contine niste atribute comune ale claselor copii doar asa ca sa nu le repetam, si clasele copii sa nu adauge nimic nou. Inseamna ca clasele copii nu are sens sa existe cand deja este clasa parinte identica cu ele.
* De ex, cream niste Pizzas. Fie clasa Pizza

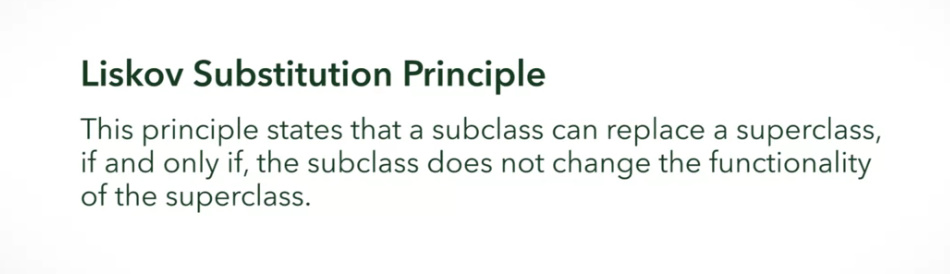


Si, de ex, cream o Peperoni pizza:



Pepperoni class nu face nimic diferit fata de Pizza. Nu are sens sa facem asa inheritance.

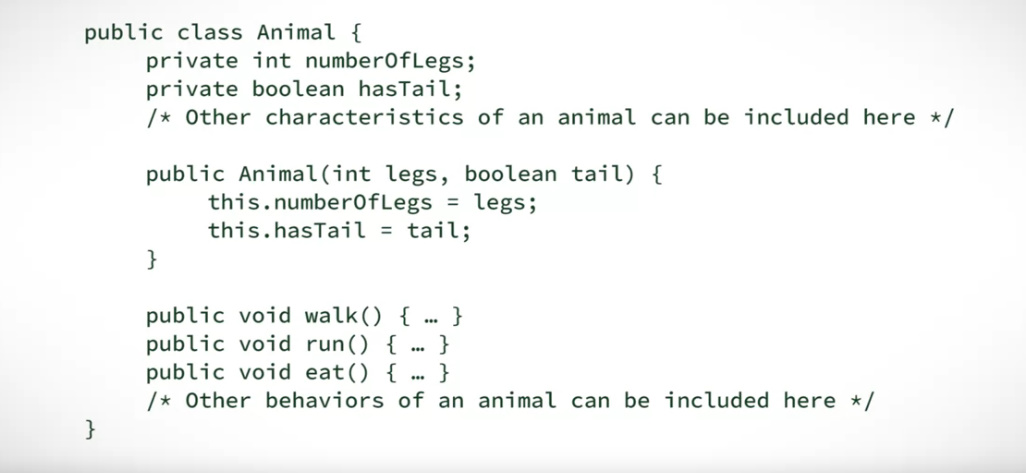
* **Un alt indiciu al inheritance gresite** este cand stricam Liskov Substitution



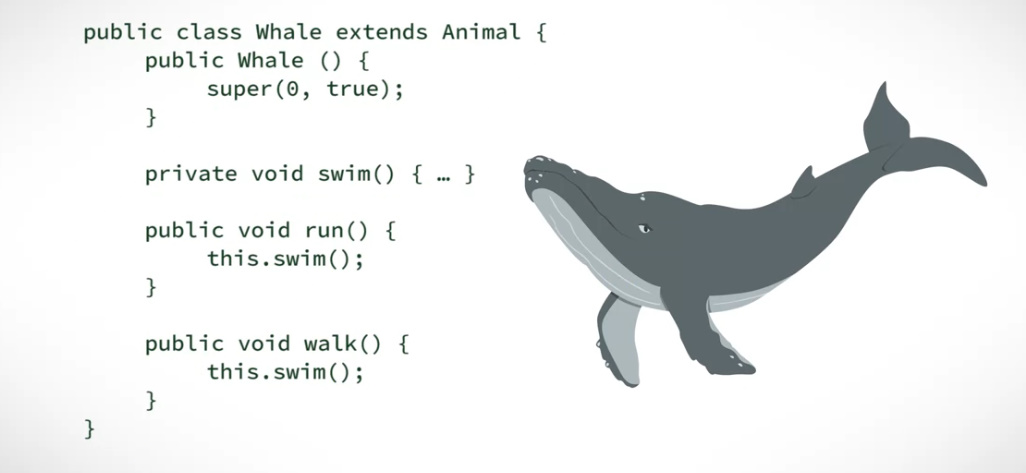
**Deci, Liskow Substitution inseamna ca o Subclass poate inlocui o superclasa doar daca subclasa nu modifica functionalitatile superclasei.**

**Daca o subclasa inlocuieste o superclasa, dar inlocuieste toate behaviours la superclasa, atunci acest principiu e incalcat.**

De ex, fie clasa:



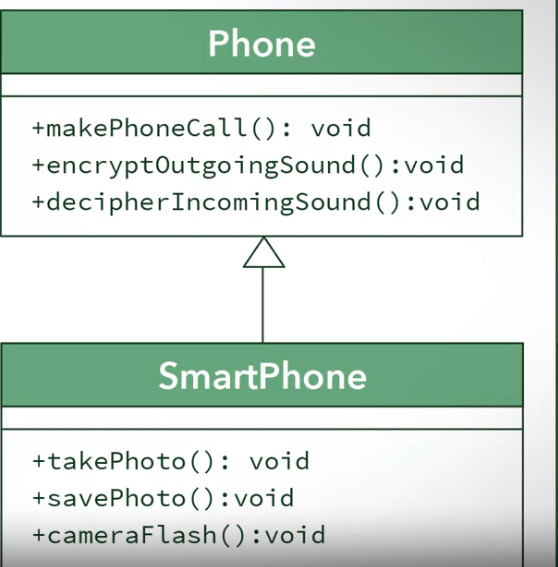
Dar, daca avem de ex o balena:



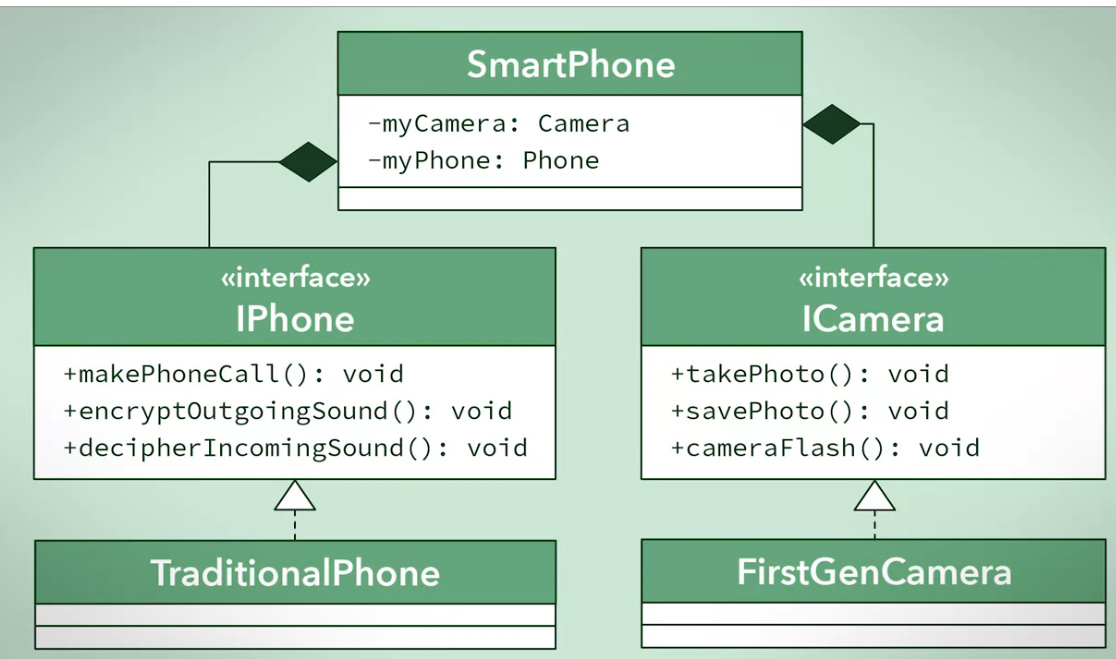
Ea nu poate merge, deci run() e deprisos. Nu avem incotro decat sa folosim swim() in run(), dar asa stricam Liskov Substitution principle. Clasal Whale nu mai poate inlocui clasa Animal si strica rolul ei, modificand scopul ei principal.

* Un exemplu rau de bad inheritance e chiar in Java Collections Framework. Stack class ar trebui sa permita doar inserarea itemelor la urma si stergerea si obtinerea lor tot de la urma. Dar, deoarece ea extinde Vector, putem adauga oriunde, putem obtine oricare. Deci, stricam cam rolul lui Vector si folosim ceea ce nici nu e necesar pentru un Stack
* Daca inheritance nu prea merge, mai bine folosim **Decomposition**

De ex, asta e un ex de rau inheritance



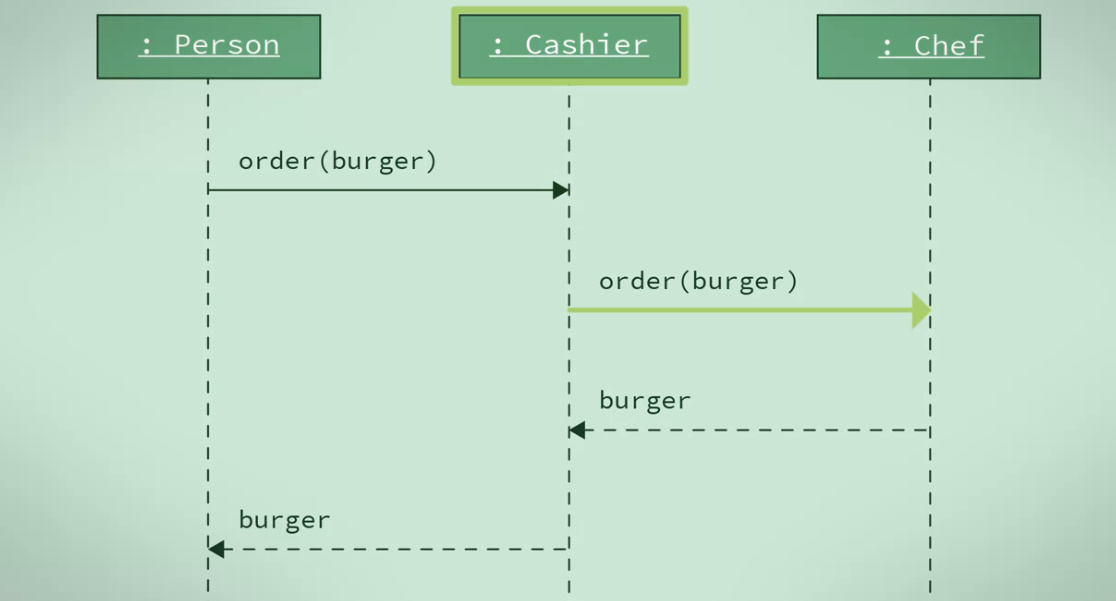
Mai bine folosim Decomposition



mai bine SmartPhome sa aiba fielduri, decat sa aiba mostenire. Deci, am folosim Composition din Decomposition

**UML Sequence diagram**

* **Sequence diagrams** – Arata cum obiectele interactioneaza unele cu altele pentru a indeplini taskuri

****

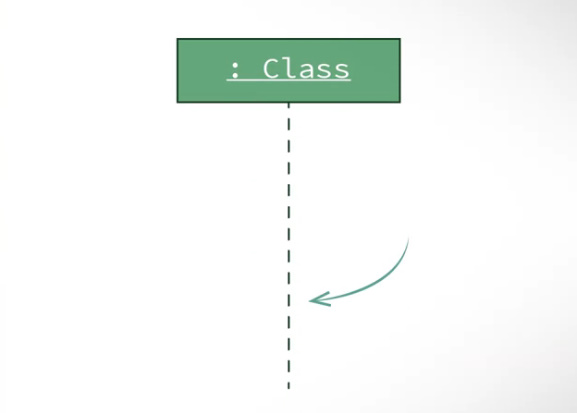
Person comanda un burger, Casierul comanda burger de la Chef, apoi Chef ii da burger lui Casier si Casier lui Person

* **Sequence Diagram** este un alt tip de **UML Diagram**
* Elemente ale Sequence Diagram:

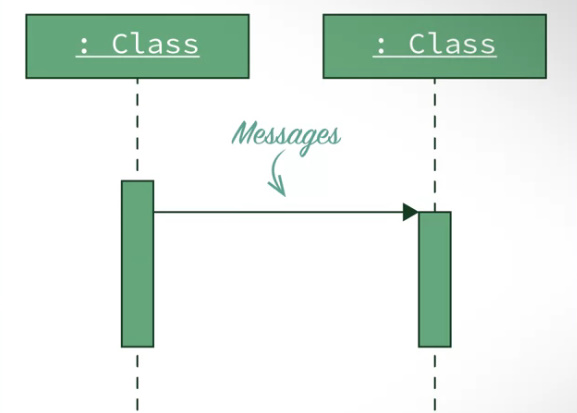
Punem intr-un dreptunghi numele la clasa obiectului cu :



Folosim o linie verticala intrerupta ca sa aratam existenta la obiect

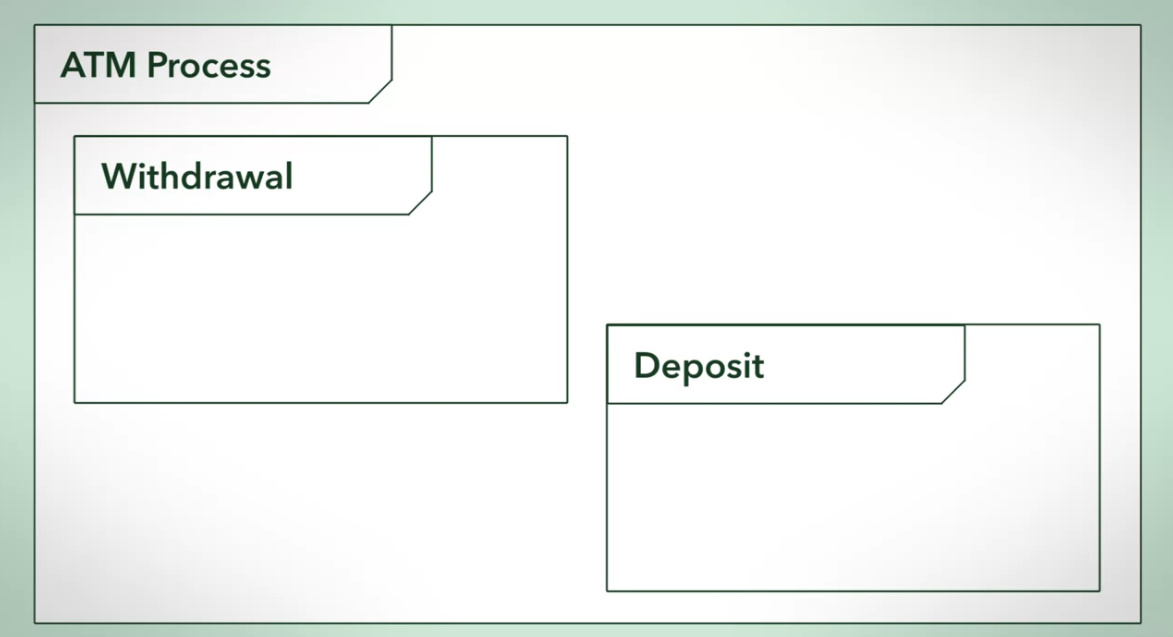


Folosim o sageata si un mesaj pentru a arata interactiunea dintre 2 obiecte



**Sageata dreapta inseamna ca trimitem date de la obiectul tinta la cel final, si sageata intrerupta cand trimitem raspuns de la obiectul final la cel tinta**

* Un Sequence Diagram poate avea mai multe sequence diagrams in ea



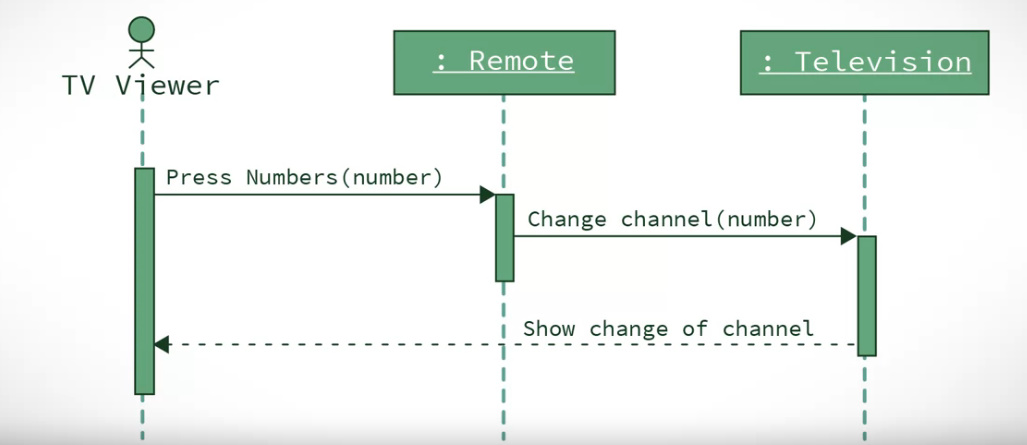
Avem un Big Sequence Diagram cu mai multe Sequence Diagrams

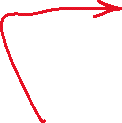
Titlul trebuie sa fie sugestiv

* Daca avem persoane ca Obiecte, le desenam cu semnul la om. Ei se numesc actors



Ex, Telespectator si Televiztor. Schimbarea canalului cu telecomanda(Remote)

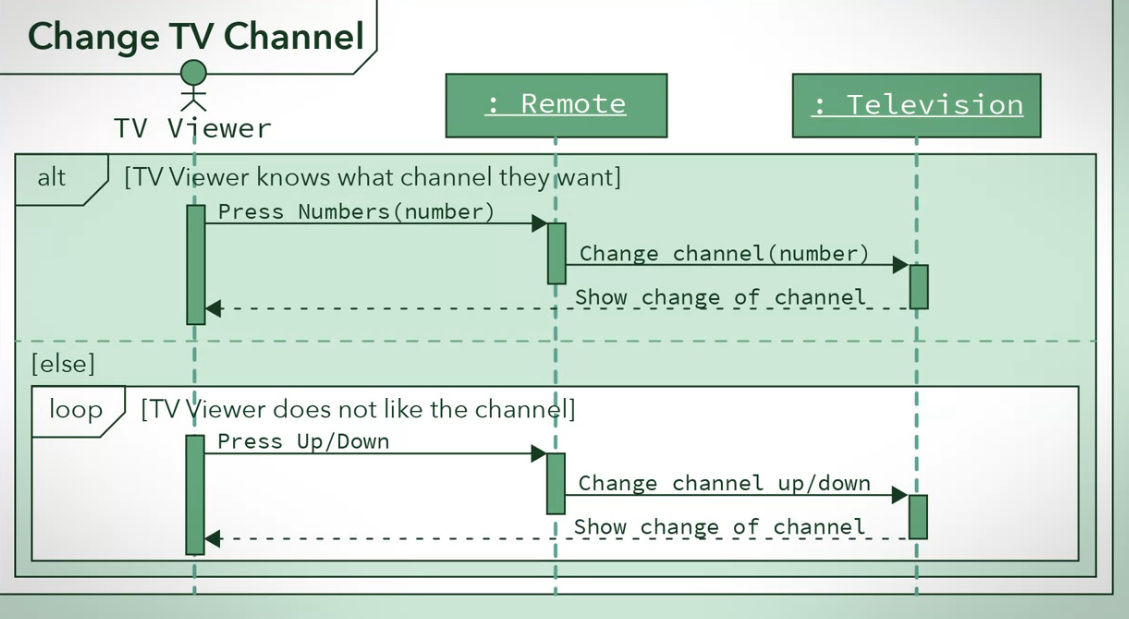


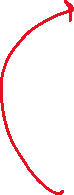


**Dreptunghi alungit** – inseamna ca obiectul este activ. Activam un obiect oricand acesta trimite ,primeste sau asteapta mesaj. Totusi, daca activarea unui obiect este facuta de un alt obiect(deci se face mai tarziu), atunci punem dreptunghiul mai jos de cat cel al obiectului sursa. De ex, vedem ca dreptunghi de la Remote e mai jos deca al Tv Viewer, deci Remote a fost activata dupa ce TV Viewer a fost activat si deja dupa ce a apast pe buton. Television tot a fost activat(a inceput a modifica canalul) dupa ce Remote s-a activat, deci iar e mai jos. Daca dreptunghiurile stau intr-o linie, inseamna ca obiectele deja erau activate si asa.

**Avem o sageata de la Television direct la Tv Viewer deoarece doar Television trimite raspuns la Tv Viewer, nu si Telecomanda deja**.

* Loop si if/else



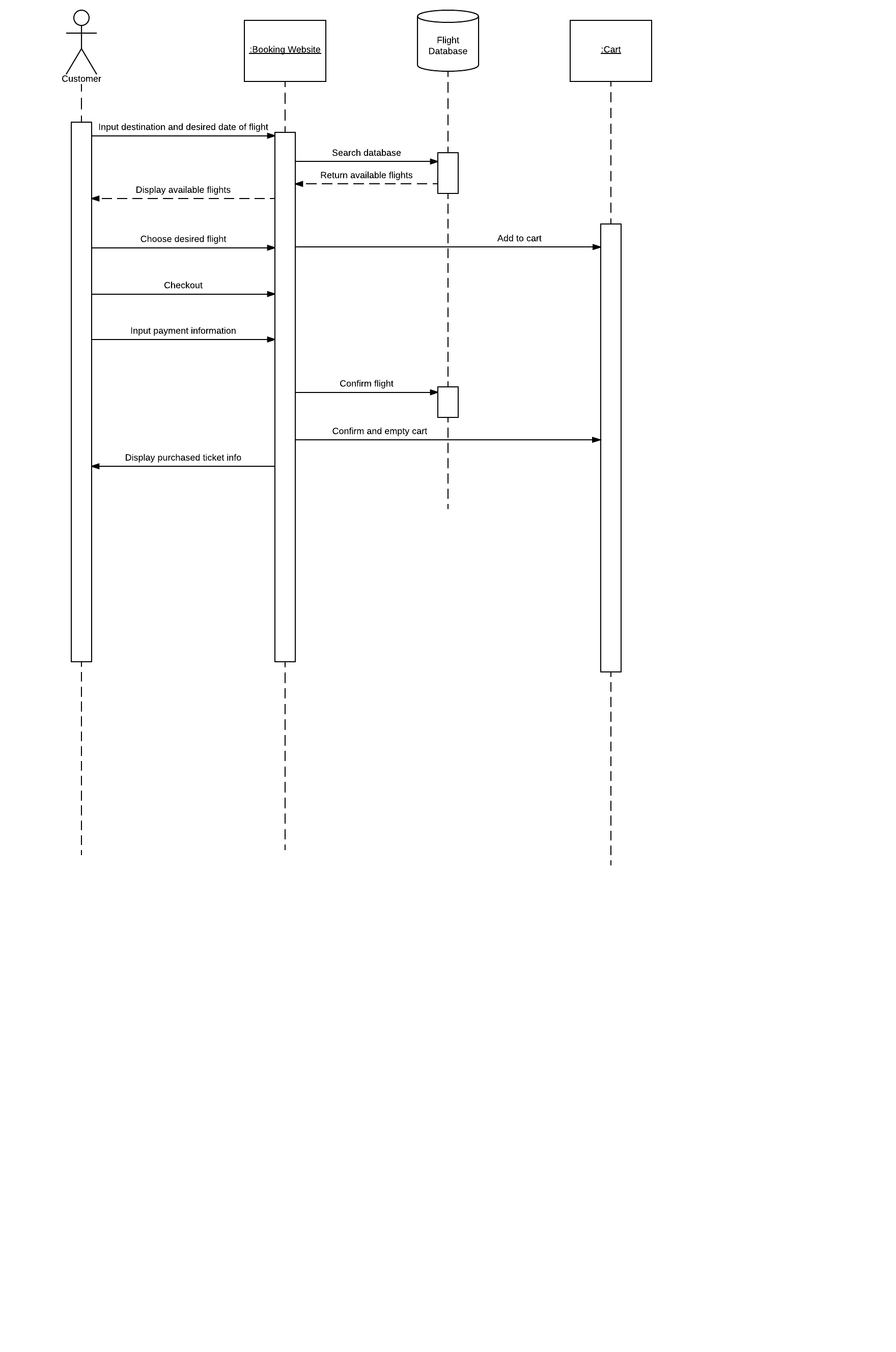


Putem avea si mai multe variante posibile. Gen, TV Viewer fie apasa un numar concret, fie apasa pe inainte/inapoi pana gaseste canal.

* If se pune intre [] si aici punem ce actiune sa fie true
* Loop se indica aici si arata ca procesul este iterativ, adica poate fi executat in continu de mai multe ori, si punem conditia ce sa fie true in []
* alt – alternative. Se foloseste cam avem mai multe sequende diagrams in una si fiecare va fi alt

**UML Sequence Exemplu**

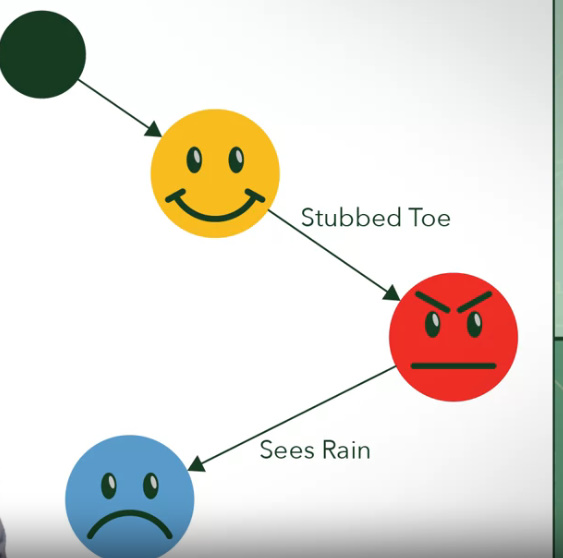
Un Customer comanda un ticket pentru un zbor de un website. Website verifica zborurile in baza de date, le returneaza, Customer alege flight, el e pus in Cart al sau, apoi confirma plata si WebSite returneaza ticker cu informatiile necesare i goleste Cart



**UML State Diagram**

* UML State Diagram – este o tehnica pe care o putem folosi pentru a descrie cum se comporta si raspunde sistemul la events.

De ex:



vedem cum diferite actiuni produc o stare diferita la o persoana.





De ex, fie un Car. Daca schimbatorul de viteza este in pus pe Back, masina se poate misca doar in spate, adica in Back state este. Daca schimbatorul de viteze este pus pe Forward, masina este in Foward state si se poate misca in fata. Valoarea la schimbator de viteza, sau mai bine zis valoarea la atribut, modifica comportamentul la masina si state al ei.

* Exact asa este si in soft. Cand sistemul sau obiectul este intr-un oarecare state, el se comporta intr-un anumit fel. Anume State Diagram ne arata cum se comporta obiectul/sistemul in diferite states.

**Desenarea la ULM State Diagrams**

* **Incepem cu un punct de pornire, ce e un cerc,mereu!!!!.**
* Fiecare **state** are 3 componente de baza:

- nume

- variabile – data relevante unui state object. De ex, daca un curs are multi studenti, va avea o variabila String cu valoarea “FULL”

- activities



Un state trebuie sa aiba cel putin un State Name

* **Activitatile** sunt actiuni ce au loc in anumit state
* Exista 3 activitati pentru orice stare:

- **Entry** – actiuni care au loc cand state abea intra in unul, adica abea a iesit din unul si a intrat in cel curent.

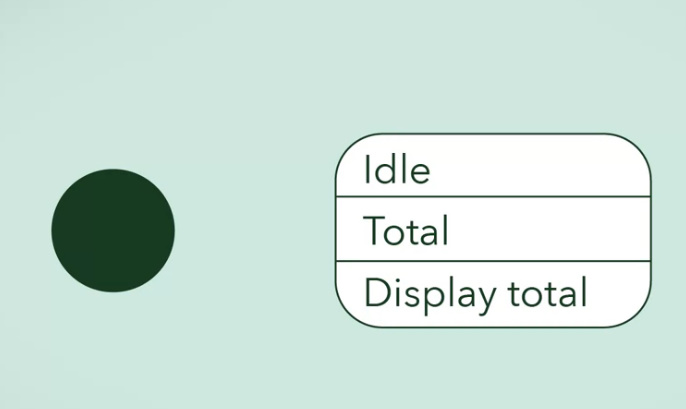
- **Exit** – actiuni care au loc cand state iese si se duce catre alt state

- **Do** – actiuni care au loc odata sau de mai multe ori atat timp cat obiectul e intr-un state



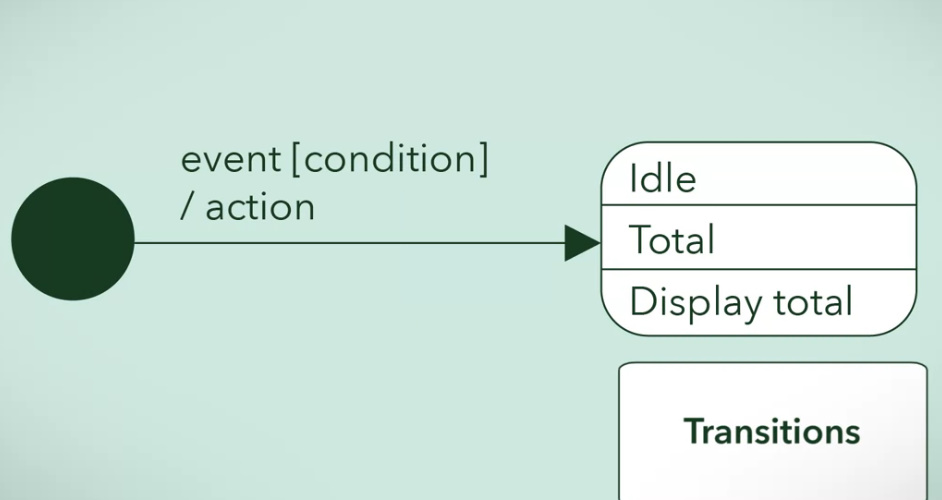
**Exemplu**

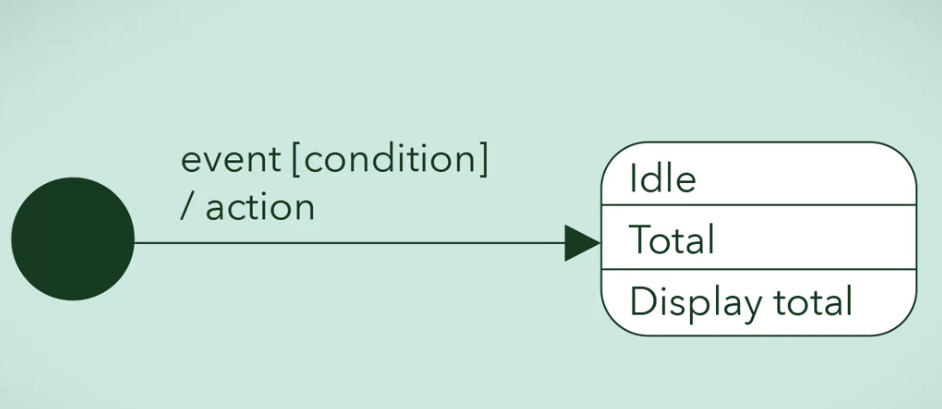
* Fie o masina de unde cumparam cu monede dulciuri



Cand masina e in Idle(inactiv) state, ea arata nr total de monede inserate pana acum. Pentru variable e total,caci abea a intrat in state Idle,desi si asa era, si pentru activity e display total

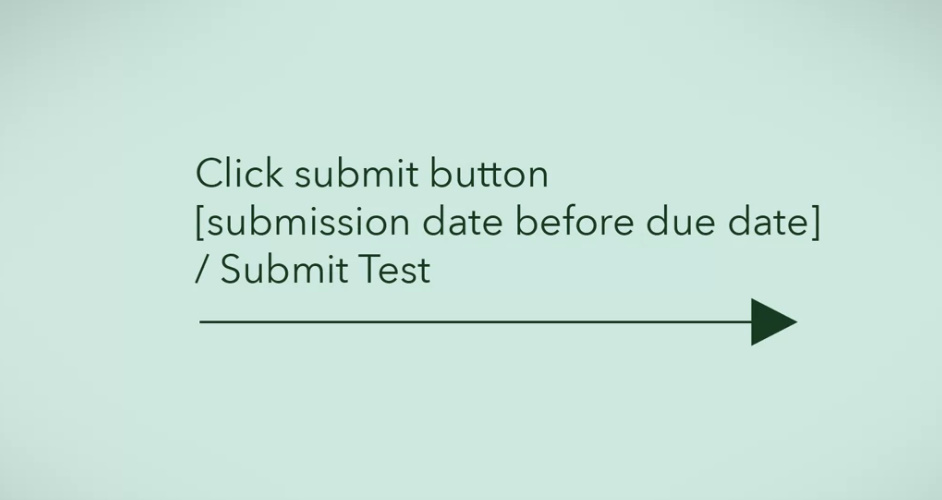
* **Transition** – conexiune intre 2 states.





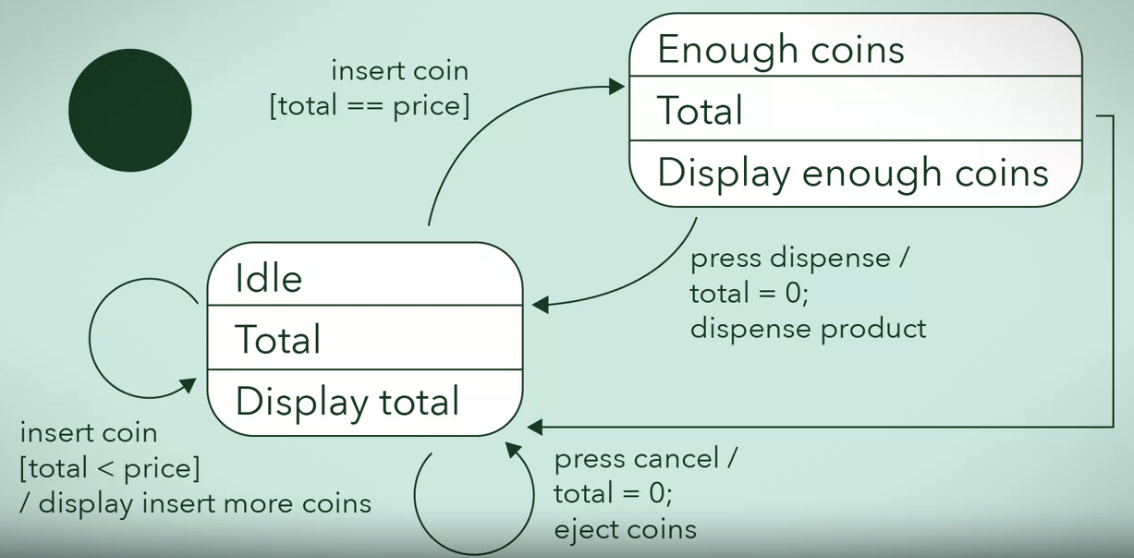
Fiecrae transition are un event si poate avea o conditie si o actiune. Action se intampla daca Event are loc si Condition e TRUE

* De ex, facem un test online si vrem deja sa il trimitem, odata ce am terminat testul:



Click pe button e event,[] avem conditia ce trebuie sa fie true ca actiunea Submit Test sa aiba loc.

* **Ciclu** – un event care se poate repeta de mai multe ori, in mod consecutib, se arata cu o sageata in forma de cerc:

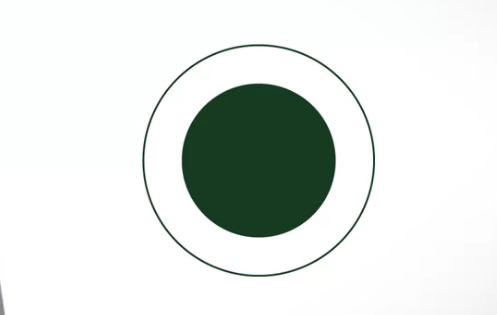


Deci, daca [nu am introdus suficiente monede], actiunea va fi sa arate un mesaj sa mai introducem monede. Cand total==price, trecem in state Enough Coins

Daca apasam pe retragere item, adica Press Dispense, numarul de monede se seteaza 0 si primim produsul. Daca ne razgandim, apasam cancel, total se seteaza 0 si monedele sunt aruncate

* **Termination** – reprezinta distrugerea unui obiect sau completarea unui proces.

**E desenat asa**:

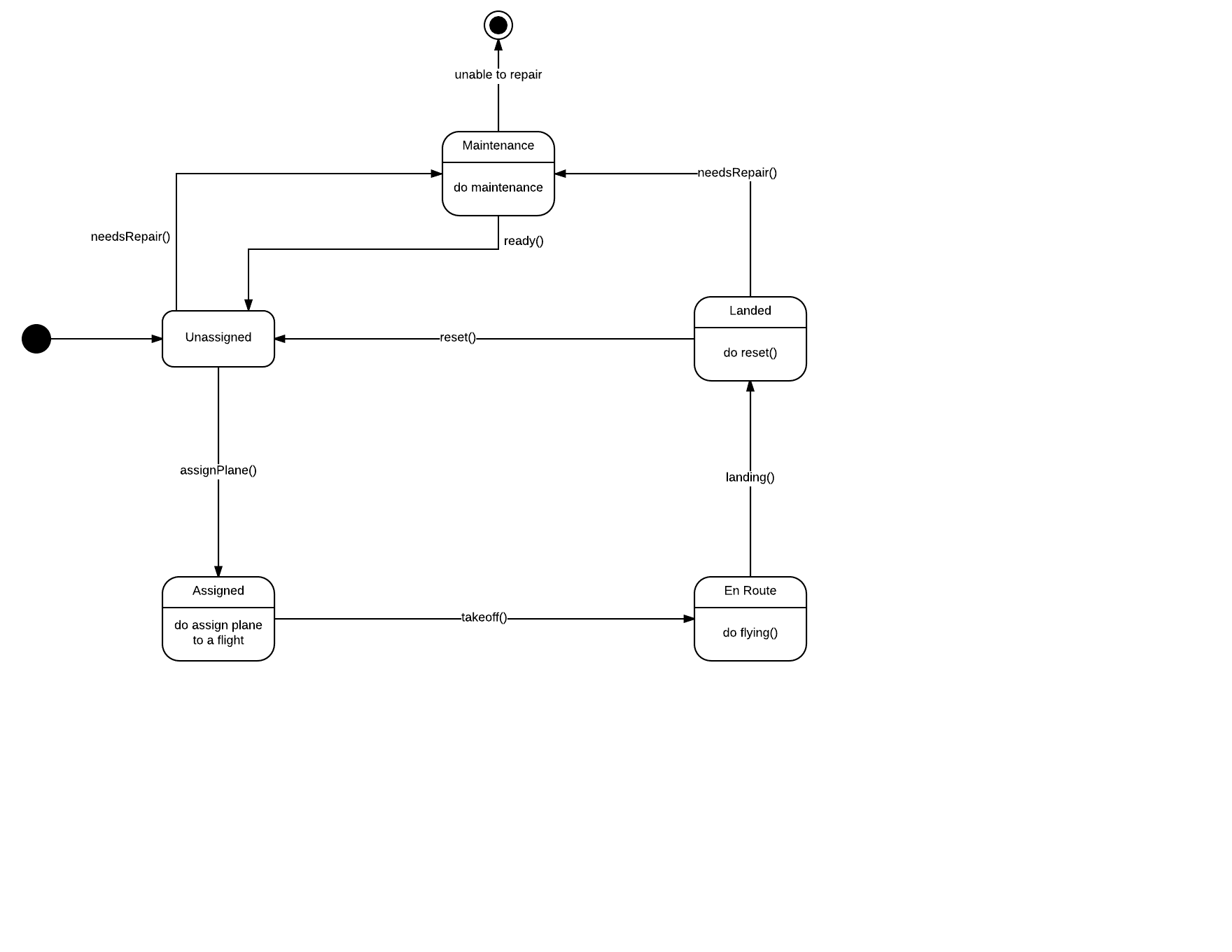
****

De ex, cand primi cardul din cache machine, aia e termination

**Nu toate diagramele au mereu un termination**

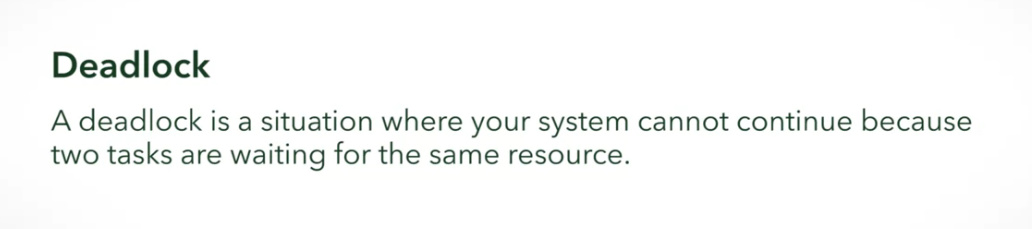
**project**

The airplanes should go through multiple different states. When planes are not in use for a flight they are usually **waiting** to be assigned. Once a plane is chosen to be used for a flight, they are **assigned** to that flight until the airplane is ready for take-off. While the plane is in the air and flying the state is termed **‘en route’**. Once the plane has reached its destination, the plane has to change into a state of **landing** for the airport to prepare for its arrival. Finally, once the plane has successfully landed, the plane is checked to see if it is ready to be assigned to a new flight or if maintenance is required. If **maintenance** is required the plane is unusable and if a mechanic decides that the plane cannot be repaired it is removed from the airport and disposed.

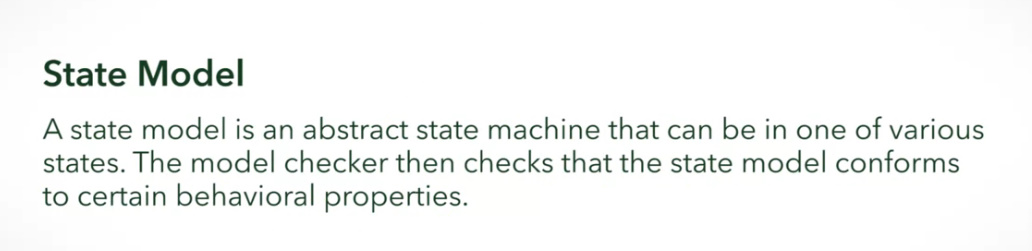


**Model Checking**

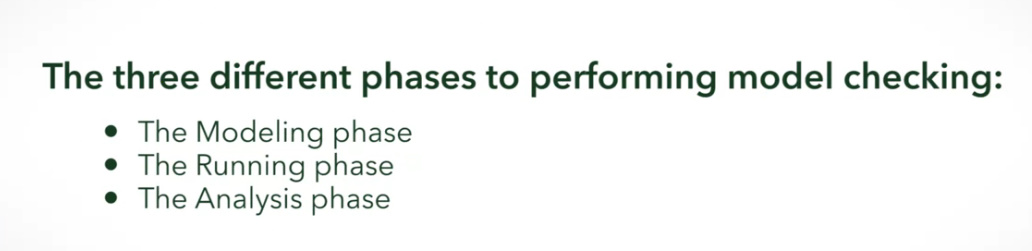
* Este necesar sa stim daca systemul creat si desenat de noi in UML Sequence si state este corect gandit
* **Model checking** – verificare sistematica a **state model** a sistemului in t oate states posibile in care se poate afla.
* El consta in faptul ca verificam toate states posibile ale software pentru a verifica si identifica posibilele erori, simuland diferinte evenimente care ar schimba states si varibilele la software
* Facem Model Checking dupa ce am scris codul deja, nu doar am desenat ULM Diagramas.
* Model Checking nu se aplica la orice code. Uneori poate fi mult prea anevoios sau chiar inutil.



* Model Checking creaza un State Model pentru code.



* Sunt 3 etape pentru a face Model Checking:



**Modeling Phase** – introducem model description. O facem in orice limbaj de programare. Putem face anumite verificari la program chiar.

**Running Phase** – rulam Model Checker pentru a vedea cum model se conforma la design properties scrie in etapa de modelare

**Analysis phase** – verificam daca design properties sunt respectate si daca sunt violate. Violarile sunt chemate **counterexamples**

* Model checking ne ajuta sa depistam erori ce nu pot fi gasite de ex cu Unit Tests