



Software Design & Quality Assurance

Johan van den Broek

Unified Modeling Language

UML

- The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

Use cases



Use Cases

- Use cases

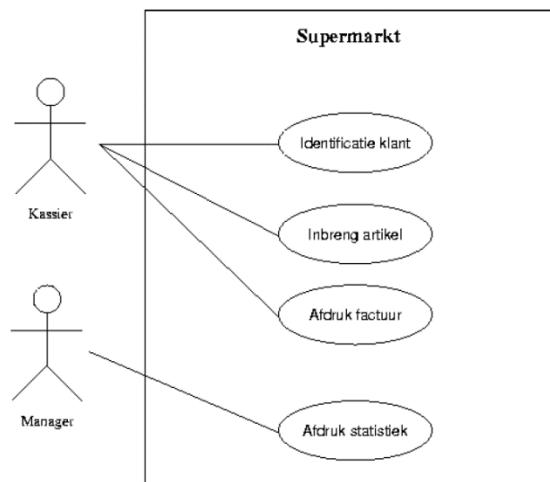
= “verhaaltjes” over een actor die op het systeem een bepaald doel verwezenlijkt

- Voorbeeld

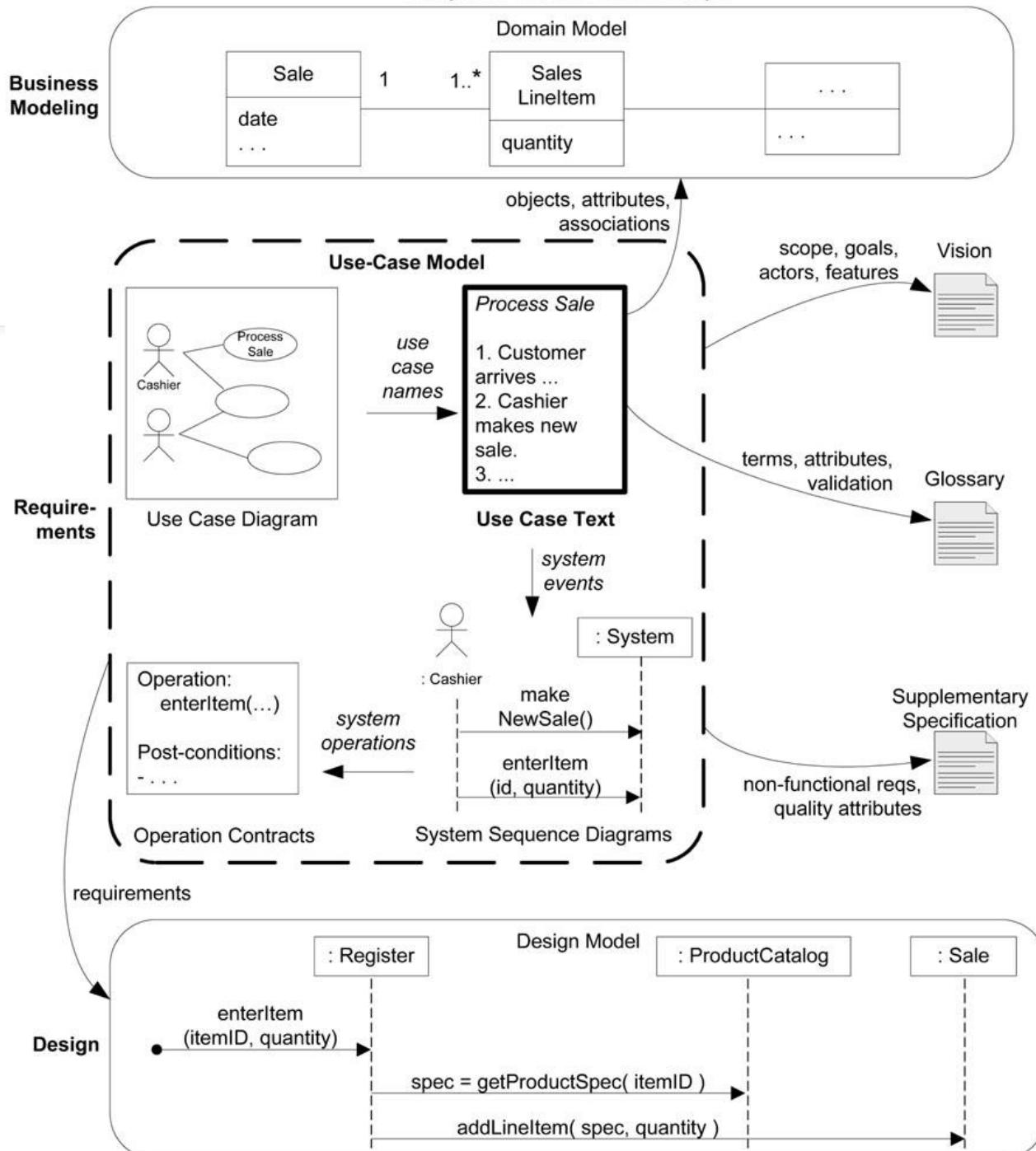
- Een klant komt aan de kassa met zijn gekozen producten. De kassier gebruikt het systeem om elk gekocht product te registreren. Het systeem laat telkens het totaalbedrag en informatie over het product zien. De klant geeft zijn betalingsinformatie op, dit wordt opgeslagen in het systeem. Het systeem past de stock informatie aan. De klant krijgt zijn ticket en verlaat de winkel met de gekochte producten.

Use Cases

- Bij uitwerken van use case
 - Nadruk leggen op tekst
 - Visuele voorstelling (use case diagram) is ook handig maar minder belangrijk



Sample UP Artifact Relationships



Use Cases

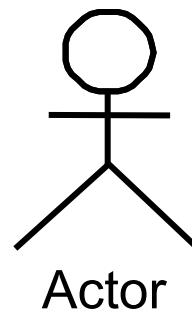
- Focus van systeem naar gebruikers
 - Niet focus op functie van het systeem
 - Wel focus op toegevoegde waarde van systeem voor belanghebbenden



Use Cases

- Actor

- De actor stelt een rol voor die een mens, hardware-apparaat of een ander systeem kan spelen in relatie tot het systeem
- Een actor is extern tov. het systeem
- UML notatie voor actor



Use Cases

- Use case is
 - De specificatie van een aantal acties,
 - uitgevoerd door het systeem,
 - wat een duidelijk resultaat tot gevolg heeft
 - dat typisch van belang is voor één of meerdere actors of andere belanghebbenden.
- Use case wordt gebruikt om gedrag van systeem te beschrijven, niet interne werking



Use Case

Use Cases

- Voordelen van use cases
 - Context voor vereisten schetsen
 - Systeemvereisten in logische sequentie gieten
 - Illustreren waarom systeem nodig is
 - Gemakkelijk te begrijpen
 - Terminologie die gebruikers/klanten begrijpen
 - Concrete verhaallijnen die het gebruik illustreren
 - Vergemakkelijken overleg met klanten
 - Vergemakkelijken het aanmaken van test cases, documentatie en design
 - Vergemakkelijken hergebruik van vereisten

Use Cases

- Use cases vs. andere vereisten

FURPS

- Functionality
- Usability
- Reliability
- Performance
- Supportability



Design constraints

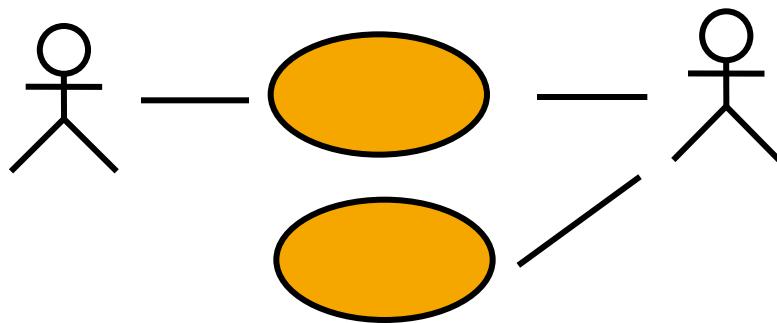
- OS
- Omgeving
- Compatibiliteit
- Applicatie-standaarden

Wettelijke vereisten

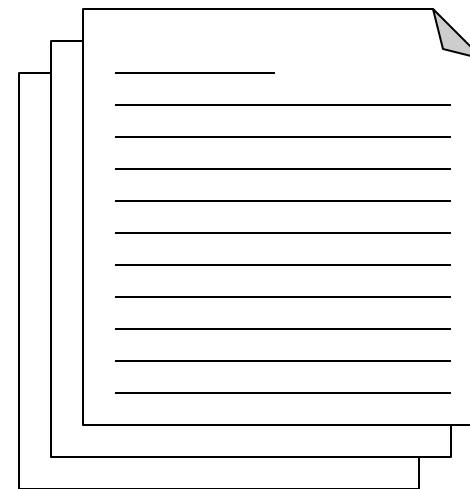
- Specifieke wetgeving
- Bv. belastingen

Use case model bestaat uit

Use-case diagrams (visuele voorstelling)

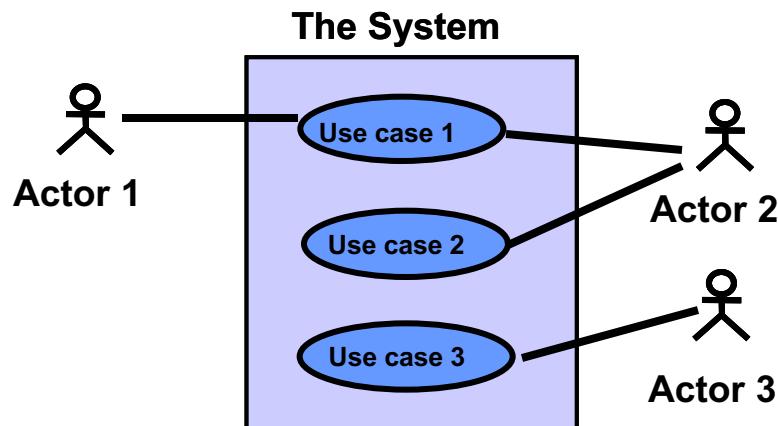


Use-case specifications (tekstuele voorstelling)



Use cases

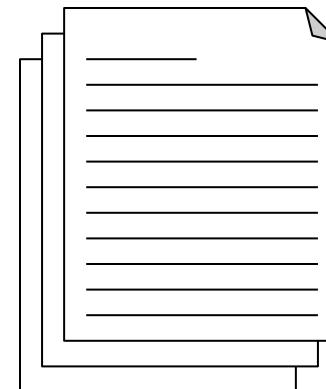
- Use case diagram
 - Stelt een aantal use cases voor met hun actoren en de onderlinge relatie
 - Definieert duidelijk de grens van het systeem
 - Identificeert wie/wat interageert met systeem
 - Vat het gedrag van het systeem samen

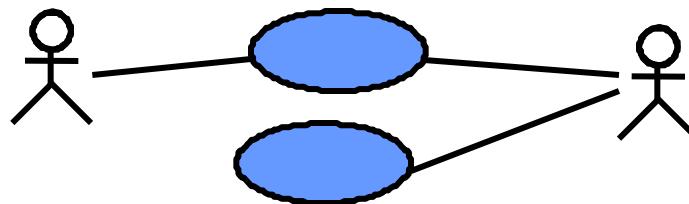


Use cases

- Use case specifications
 - Tekstuele beschrijving van de flow van events die de interactie tussen actor-systeem voorstelt
 - Extra informatie zoals
 - Precondities
 - Postcondities
 - Speciale vereisten
 - Belangrijke scenario's
 - Subflows

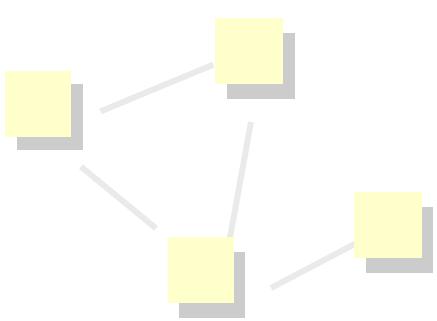
Use-case specification





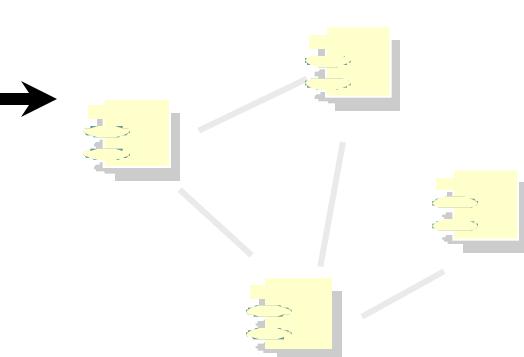
Use-Case Model

Realized by **Implemented by** **Verified by**



Design Model

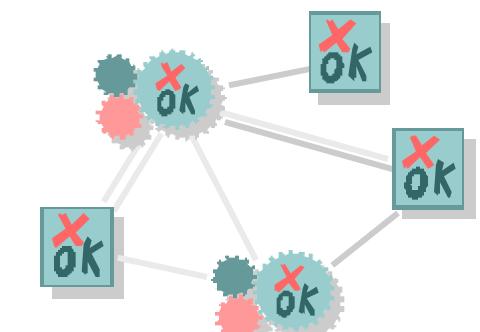
Implemented
by



Implementation
Model

Verified by

Implemented by



Test Model

Use cases

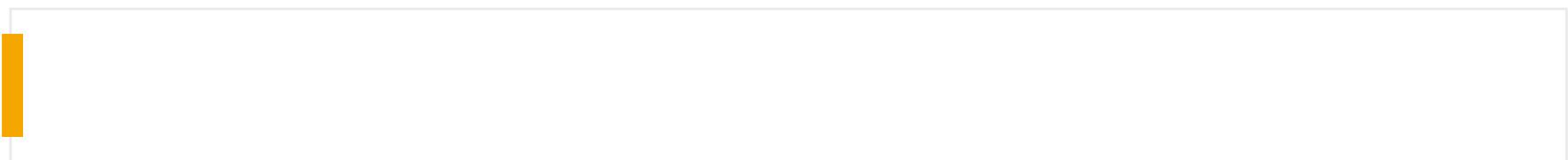
- Verschillende formats
 - Brief
 - 1 paragraaf, enkel main success scenario
 - Casual
 - "Informele paragraaf" stijl, meerdere scenario's
 - Fully dressed
 - Alle stappen en variaties in detail, met precondities, ...
 - Voorbeelden...

Fully dressed template

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	“user-goal” or “subfunction”
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, <i>and</i> worth telling the reader?
Success Guarantee	What must be true on successful completion, <i>and</i> worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.

Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Activity diagrams



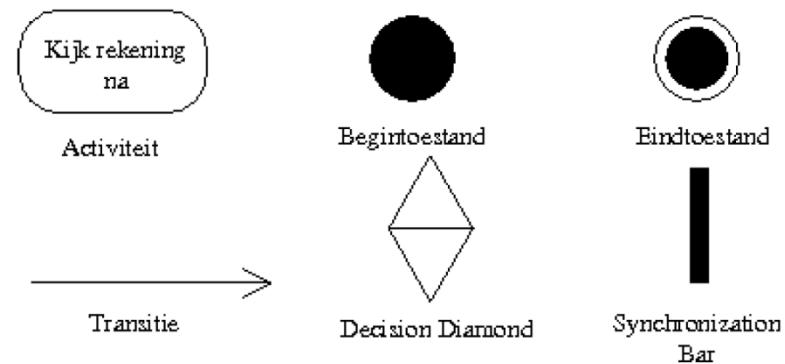
Activity diagrams

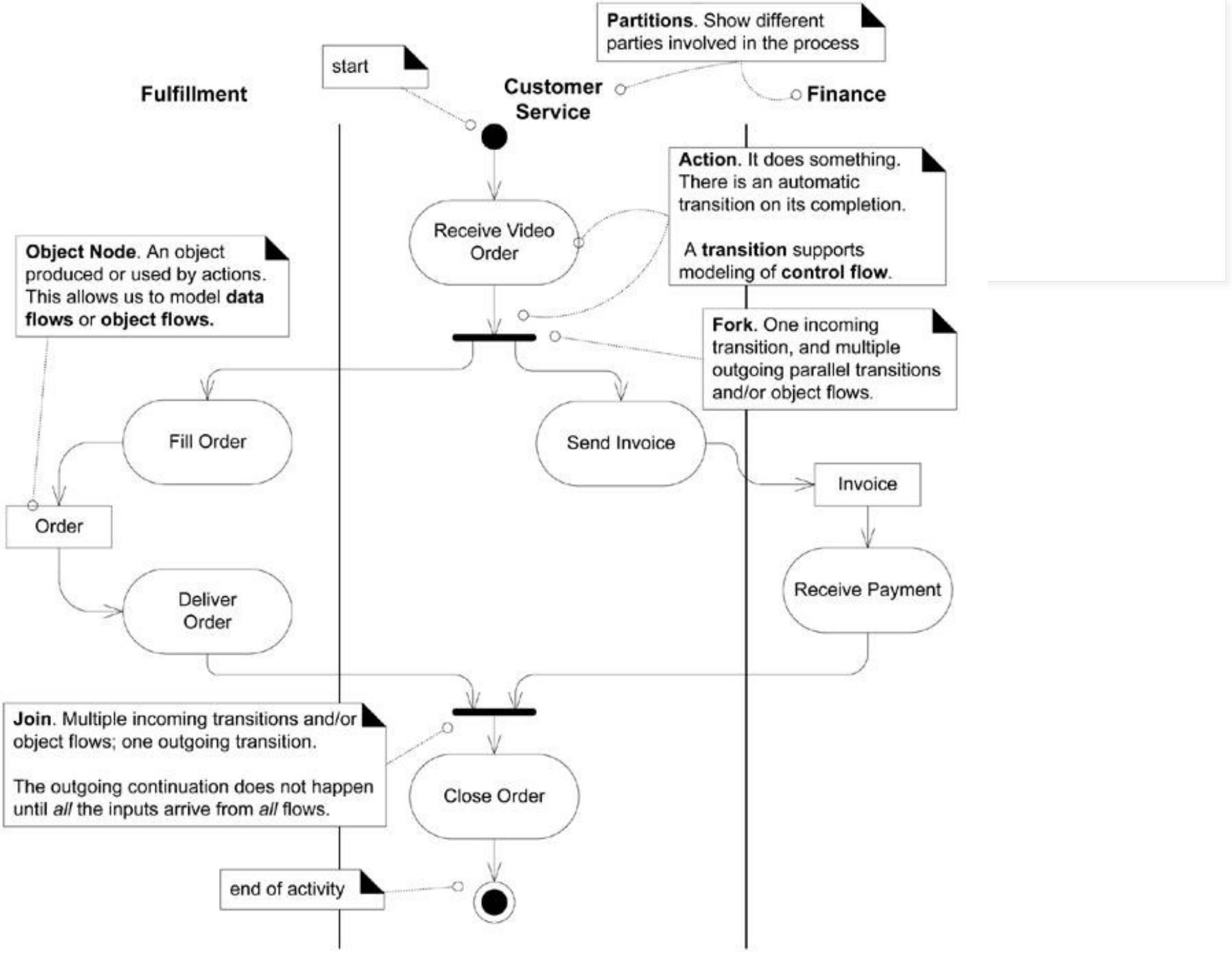
- Voor verduidelijking van workflow
- Geen methoden van klassen, maar business methoden
- Vervollediging van use case diagrams

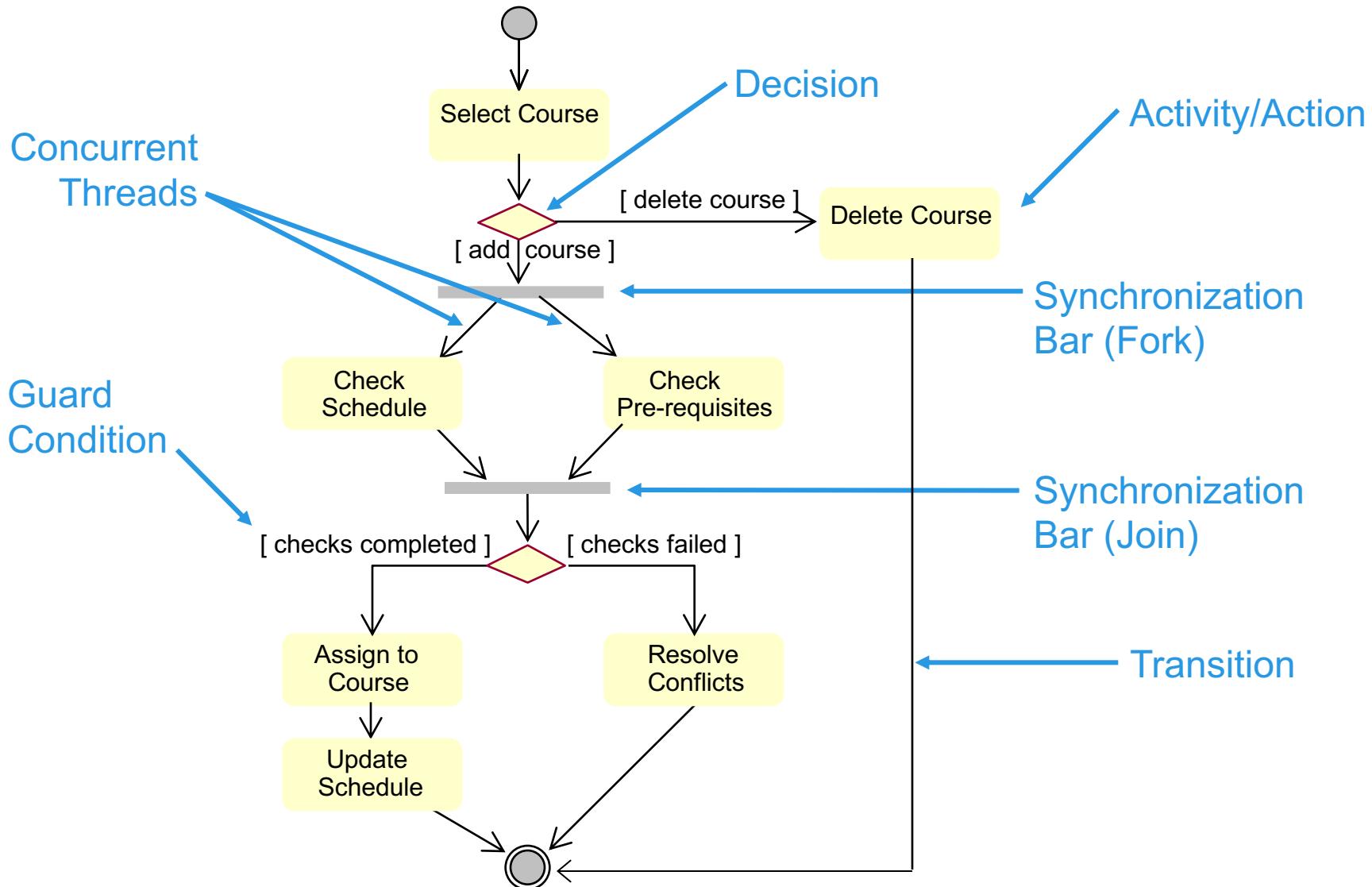
Activity diagrams

- **Symbolen**

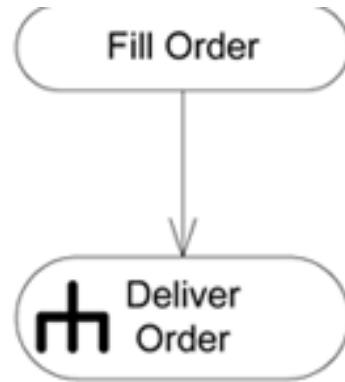
- Decision diamond
 - Voorwaarde op de flow
- Synchronization bar
 - Synchroniseren van transacties
 - Zowel binnenkomend (join) als uitgaand (fork)
- Swimlane (partitions)
 - Indeling van een activiteit
 - De baan waarvoor een bepaalde actor verantwoordelijk is





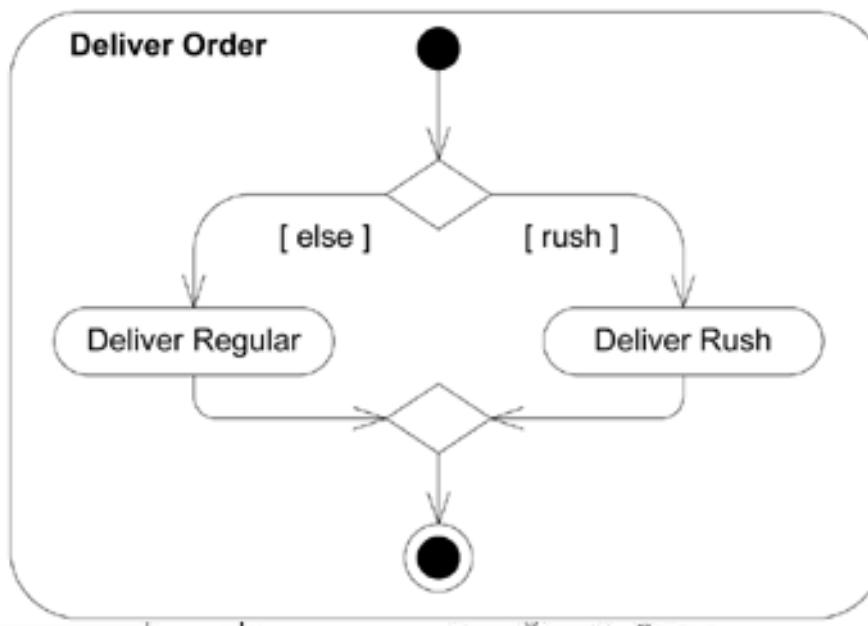


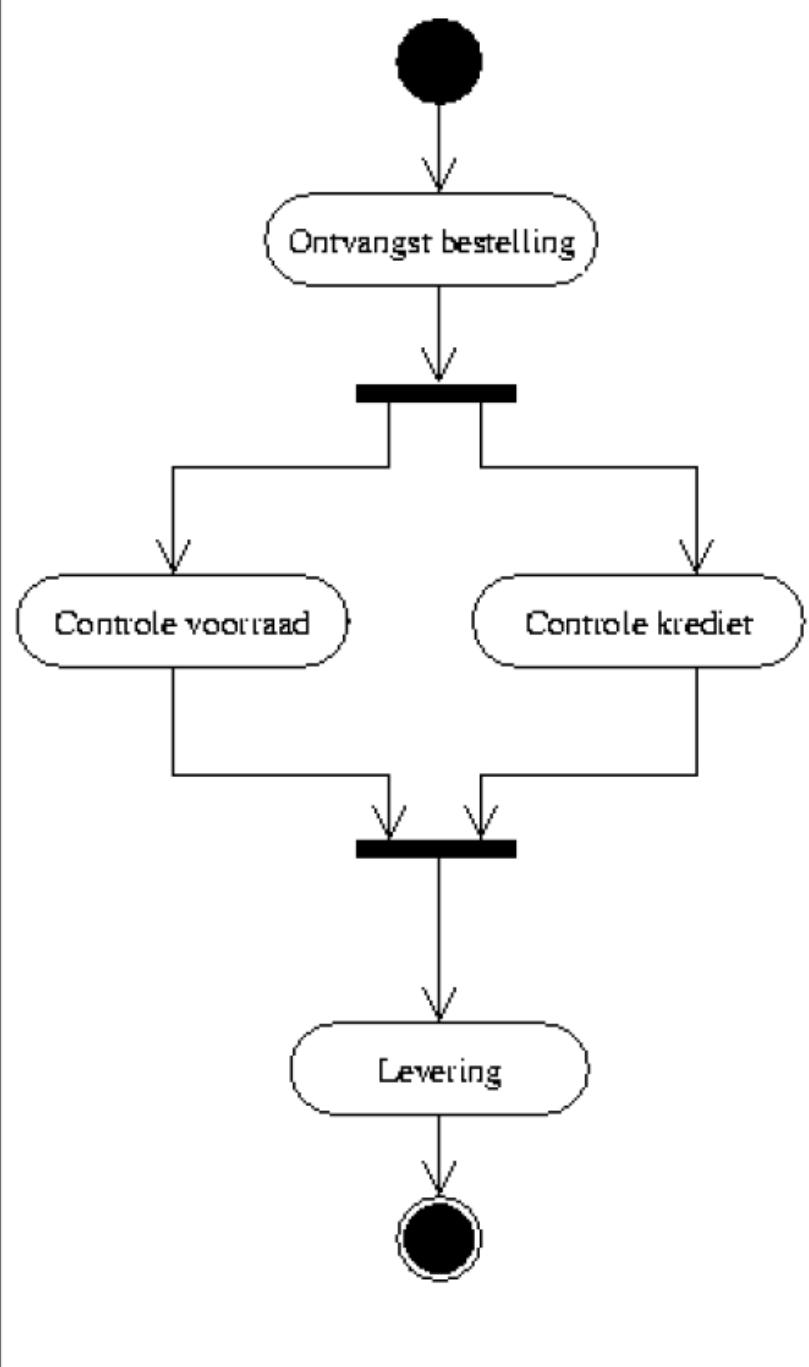
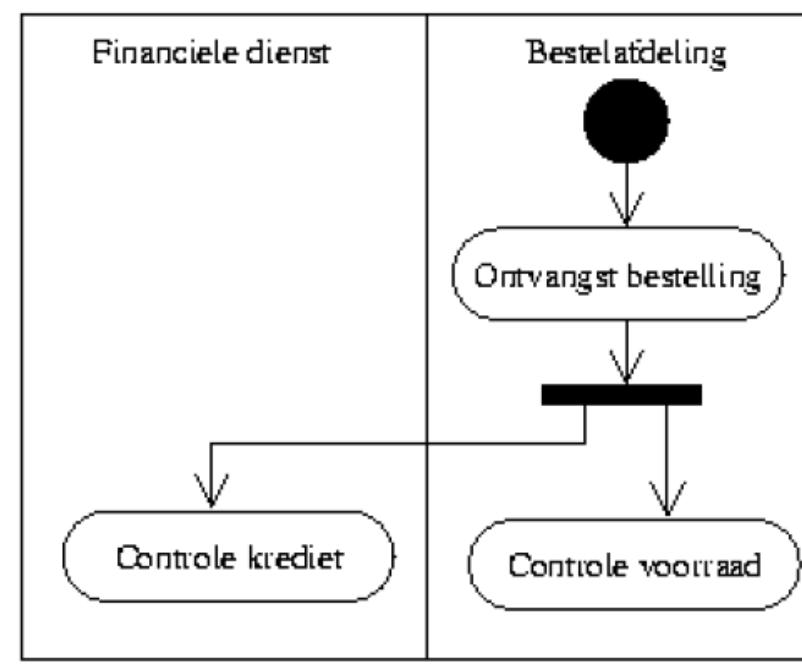
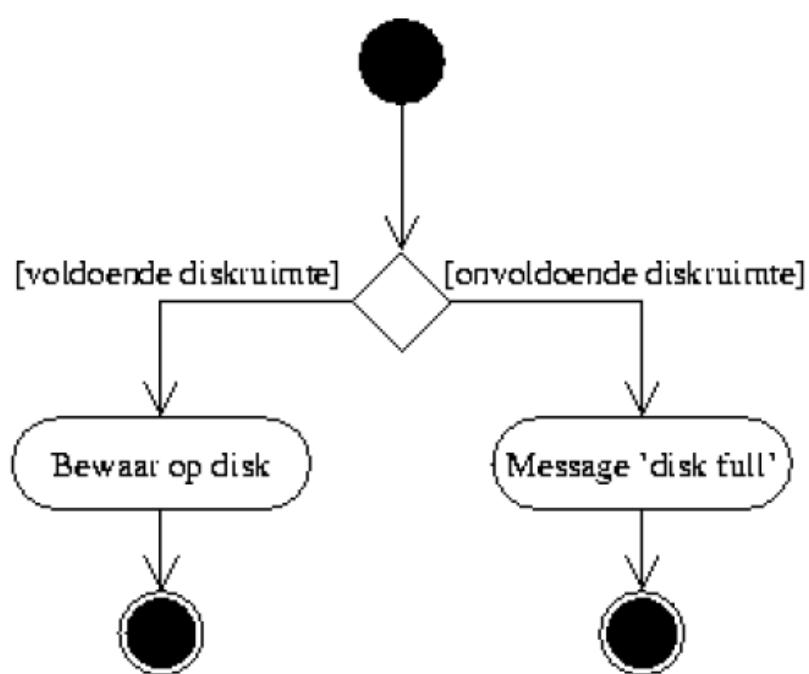
the “rake” symbol (which represents a hierarchy) indicates this activity is expanded in a sub-activity diagram

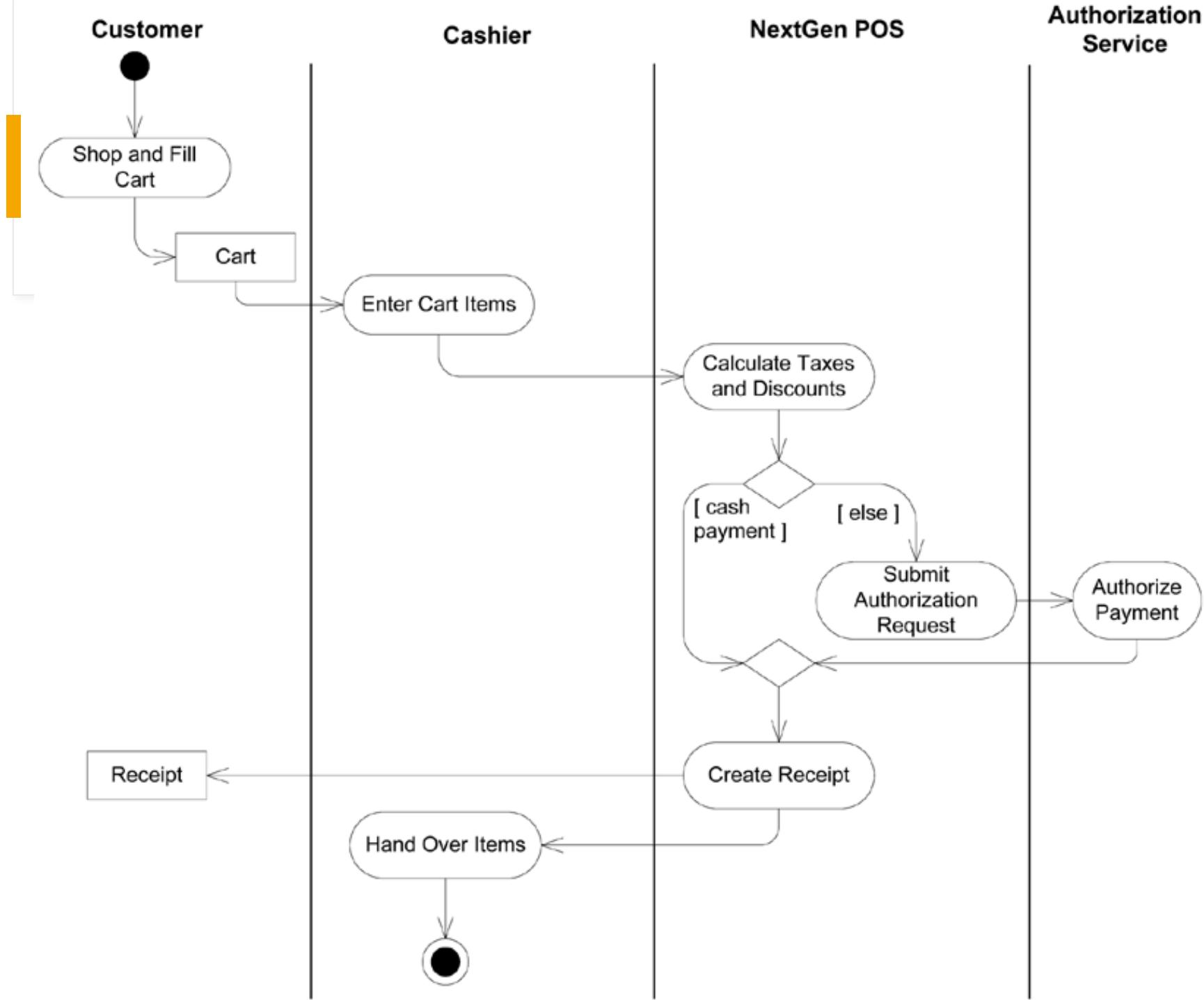


Decision: Any branch happens.
Mutual exclusion

Merge: Any input leads to continuation. This is in contrast to a *join*, in which case *all* the inputs have to arrive before it continues.

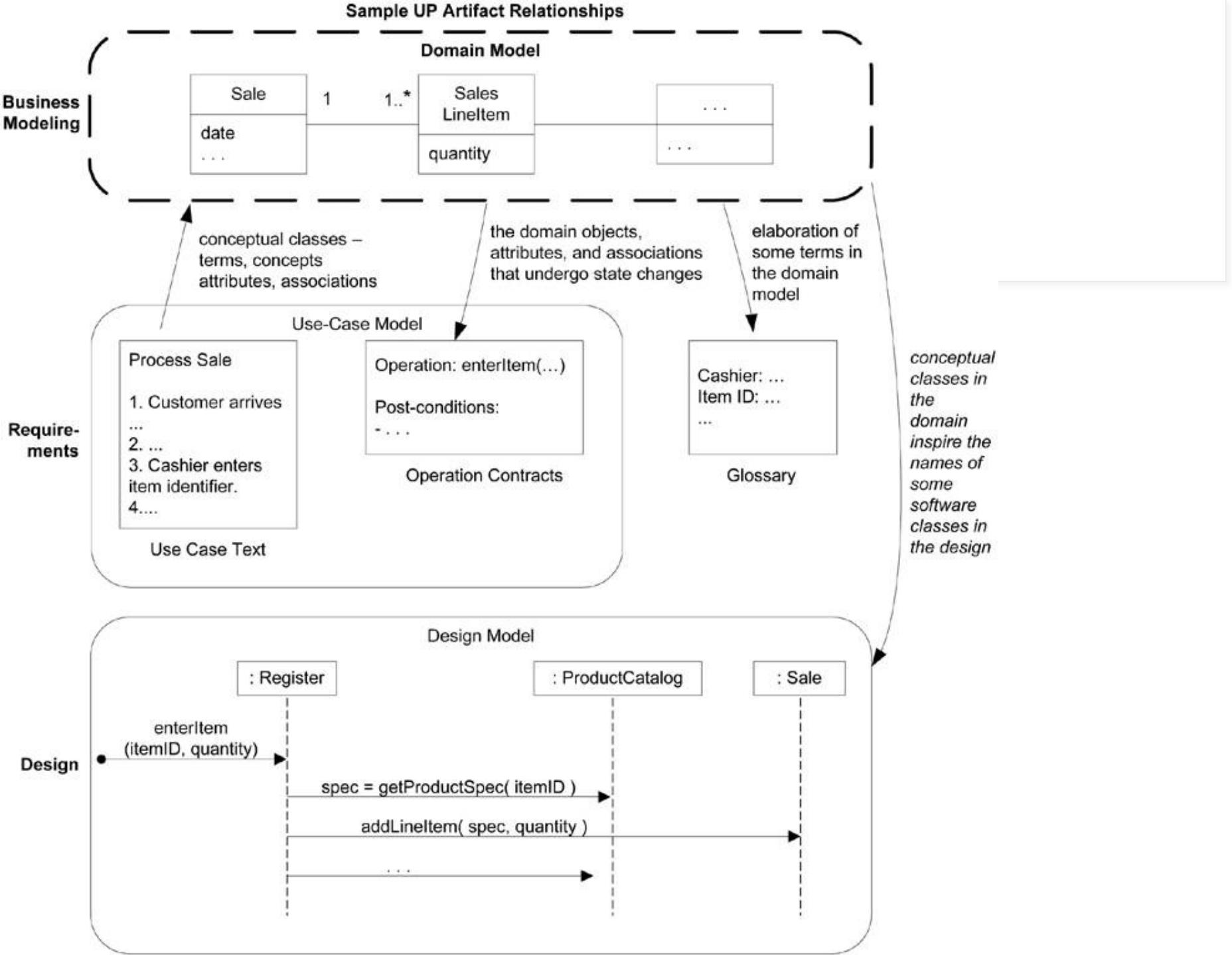




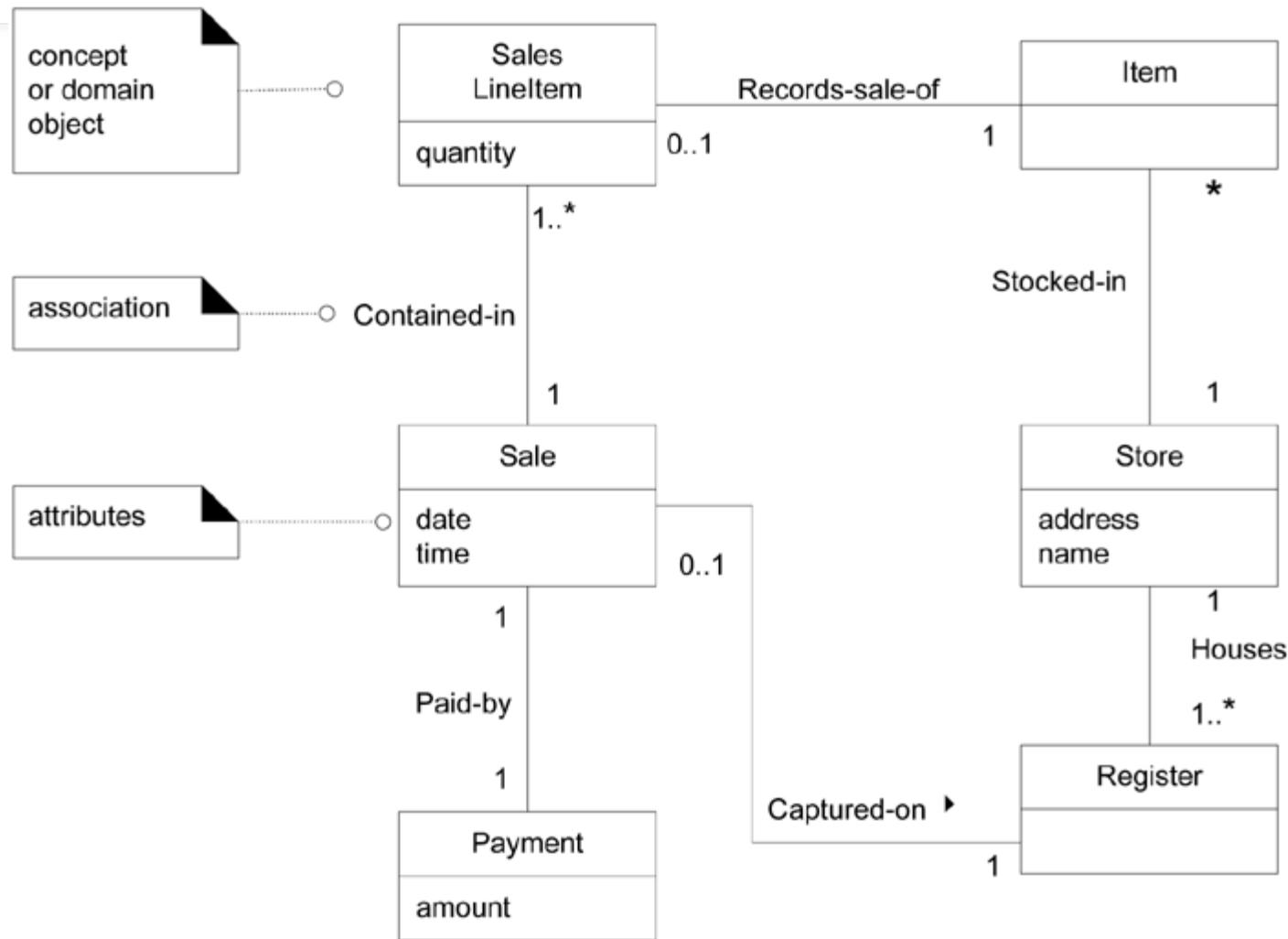


Domain models





Domain models



Domain models

- Bedoeling
 - Beter verstaan van de (begrippen uit de) probleemwereld
 - Alle termen uit de use cases in een samenhangend model brengen
 - Geeft een *conceptueel perspectief*
- Voordeel
 - Grafisch schema kan veel sneller geïnterpreteerd worden dan doorlopende tekst
 - Geen dubbelzinnigheden over begrippen

Domain models

- Wat een domain model **NIET** is
 - Geen klasse-diagram
 - Een klasse-diagram bevat ook operaties, implementatiedetails, ...
 - Het is wel zo dat een klasse-diagram dikwijls kan afgeleid worden uit een domein model

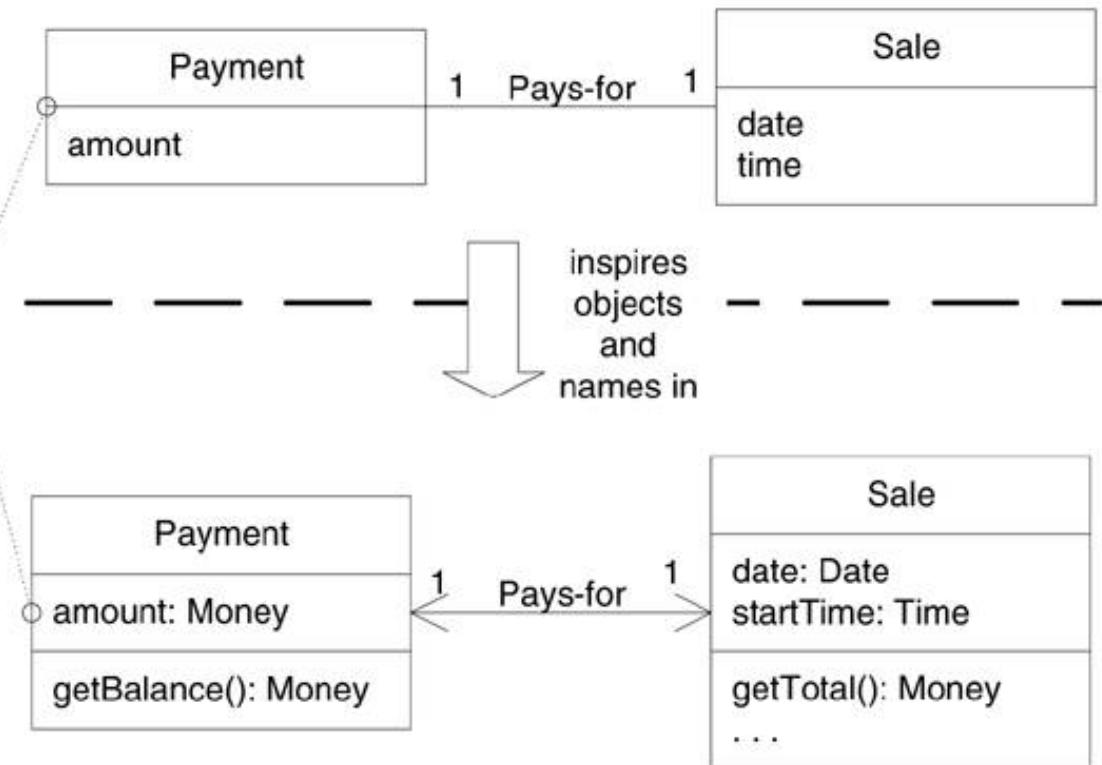
UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Domain models

- Hoe domain model creëren?
 1. Zoek de conceptuele klassen
 2. Teken deze klassen in een UML diagram
 3. Voeg associaties en attributen toe

System Sequence Diagrams



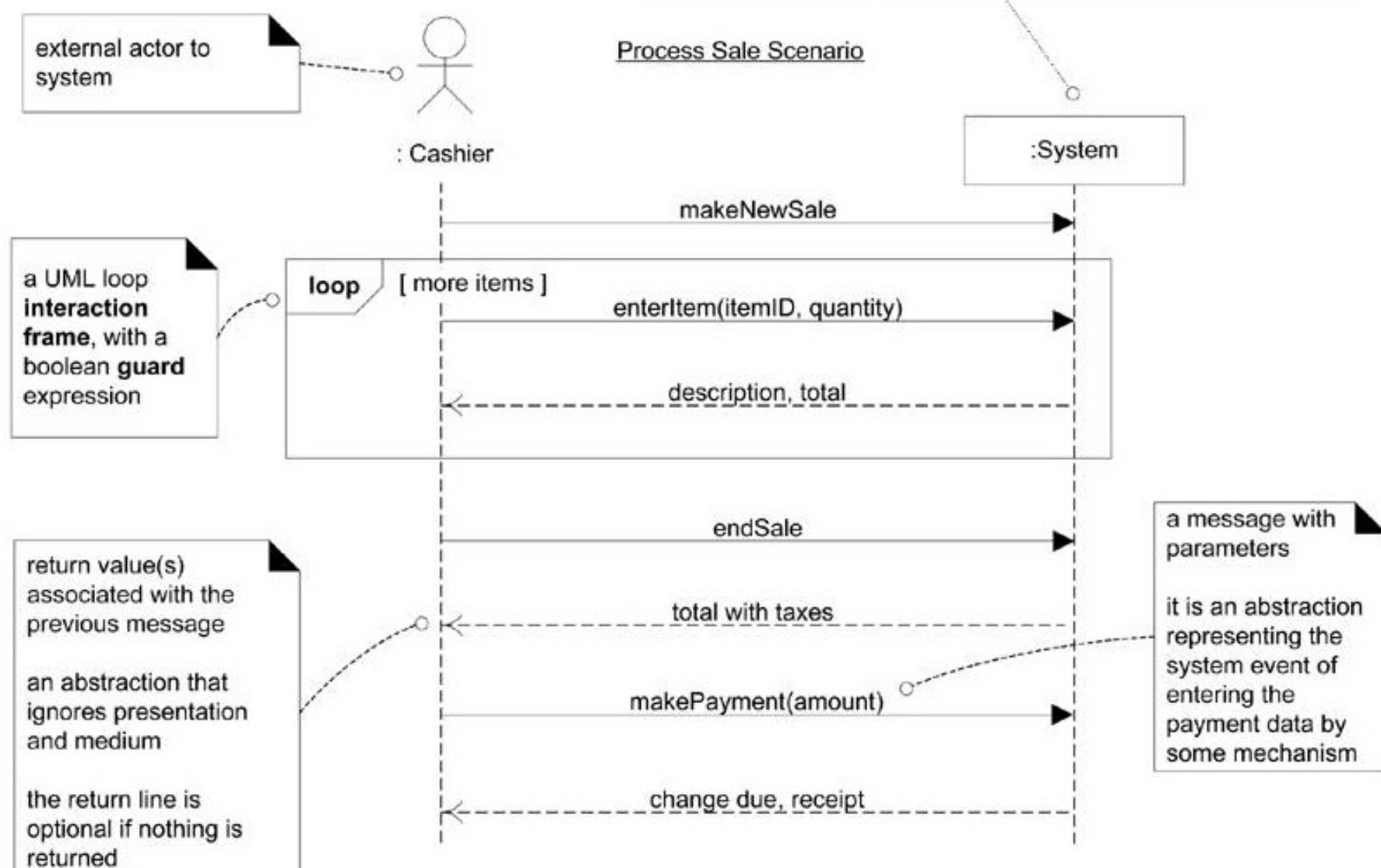
System Sequence Diagrams

- Wat is een system sequence diagram?
 - SSD geeft interactie tussen de gebruiker en het systeem weer op basis van system operations
 - SSD geeft voor een specifieke uitvoering van een use case (dus een use case scenario) de actor, het systeem en de system events die de actor op het systeem genereert

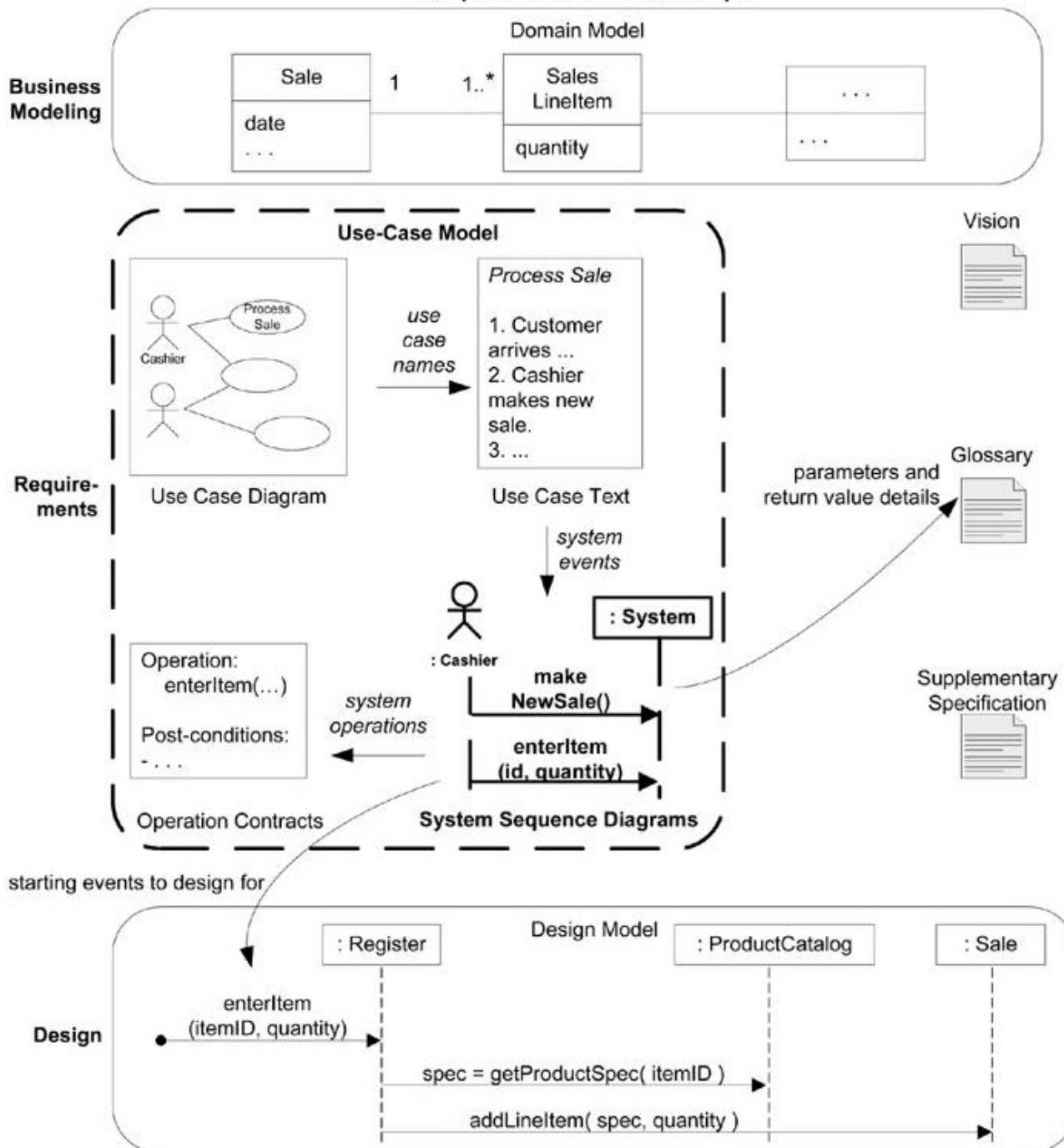
system as black box

the name could be "NextGenPOS" but "System" keeps it simple

the ":" and underline imply an instance, and are explained in a later chapter on sequence diagram notation in the UML



Sample UP Artifact Relationships



System Sequence Diagrams

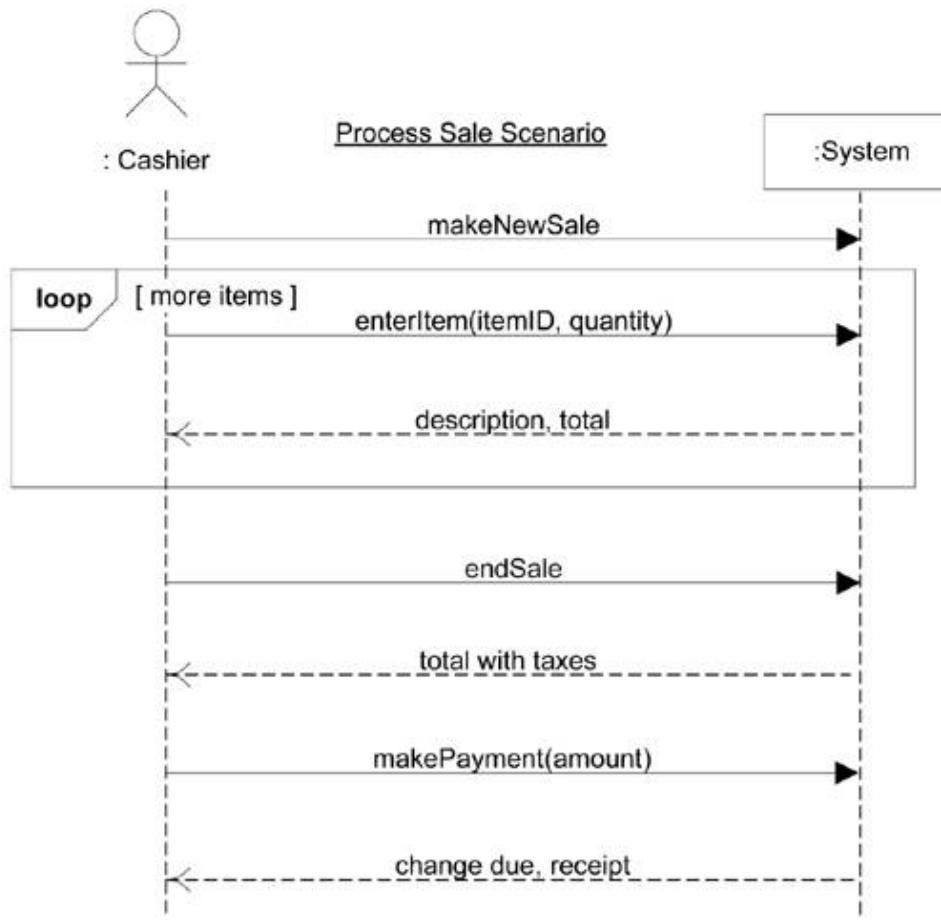
- Wanneer een SSD tekenen?
 - Enkel een SSD voor de main success scenario's van elke use case, en voor frequente of complexe alternatieve scenario's
- Waarom een SSD tekenen?
 - "Welke events komen binnen in het systeem?"
 - Typisch moet de software keyboard, muis, ... input afhandelen

System Sequence Diagrams

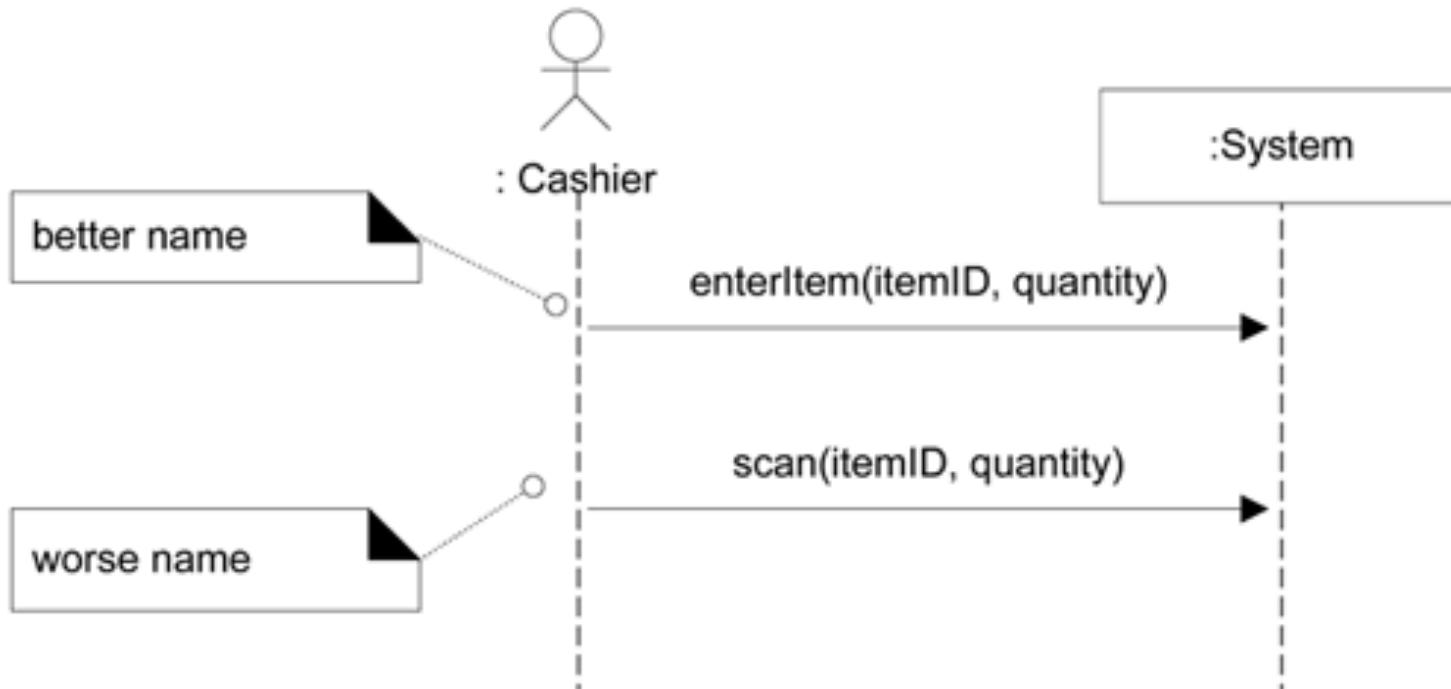
- Waarom een SSD tekenen?
 - Een systeem reageert eigenlijk op 3 dingen
 - Externe events van de actoren (muis, keyboard, ...)
 - Timer events
 - Falingen of exceptions
- SSD tekenen aan de hand van UML
 - UML voorziet een “sequence diagram” notatie

System Sequence Diagrams

- UML seq



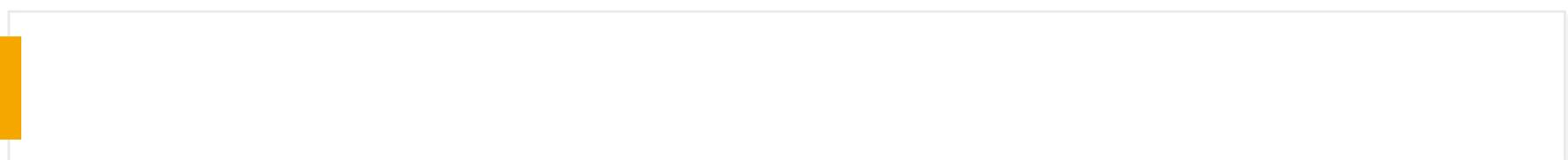
System Sequence Diagrams



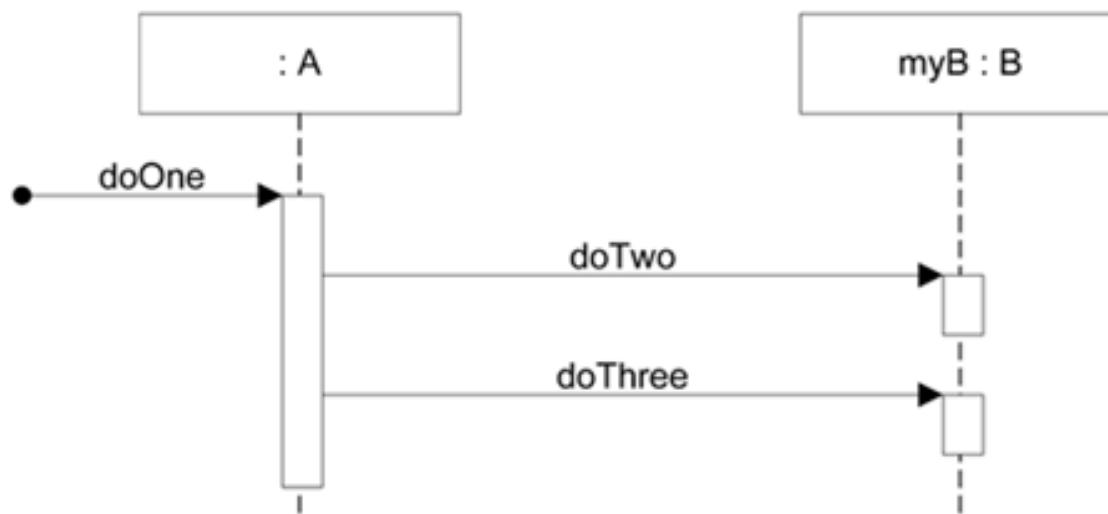
Intermezzo: richting object design

- Statische vs. dynamische modellen
 - Dynamische modellen
 - geven inzicht in de logica, het gedrag van de code of de bodies van de methoden
 - Bv. sequence diagram, communication diagram
 - Statische modellen
 - helpen bij het ontwerp van packages, namen van klassen, attributen en methodesignaturen
 - Bv. class diagram
 - !! Spendeer voldoende tijd aan dynamische!!

Interaction sequence diagram



Interaction sequence diagram



Interaction sequence diagram

- Doel: object interacties weergeven
- Tip: spendeer voldoende tijd aan het opstellen van interaction diagrams!
 - Beginnende programmeurs spenderen meestal te veel tijd aan class diagram

Interaction sequence diagram



lifeline box representing an unnamed instance of class *Sale*

:Sale

lifeline box representing a named instance

s1 : Sale

lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass

«metaclass»
Font

lifeline box representing an instance of an *ArrayList* class, parameterized (templated) to hold *Sale* objects

sales:
ArrayList<*Sale*>

related example

lifeline box representing one instance of class *Sale*, selected from the *sales ArrayList <Sale>* collection

sales[i] : Sale

List is an interface

in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal

x : List

Interaction sequence diagram

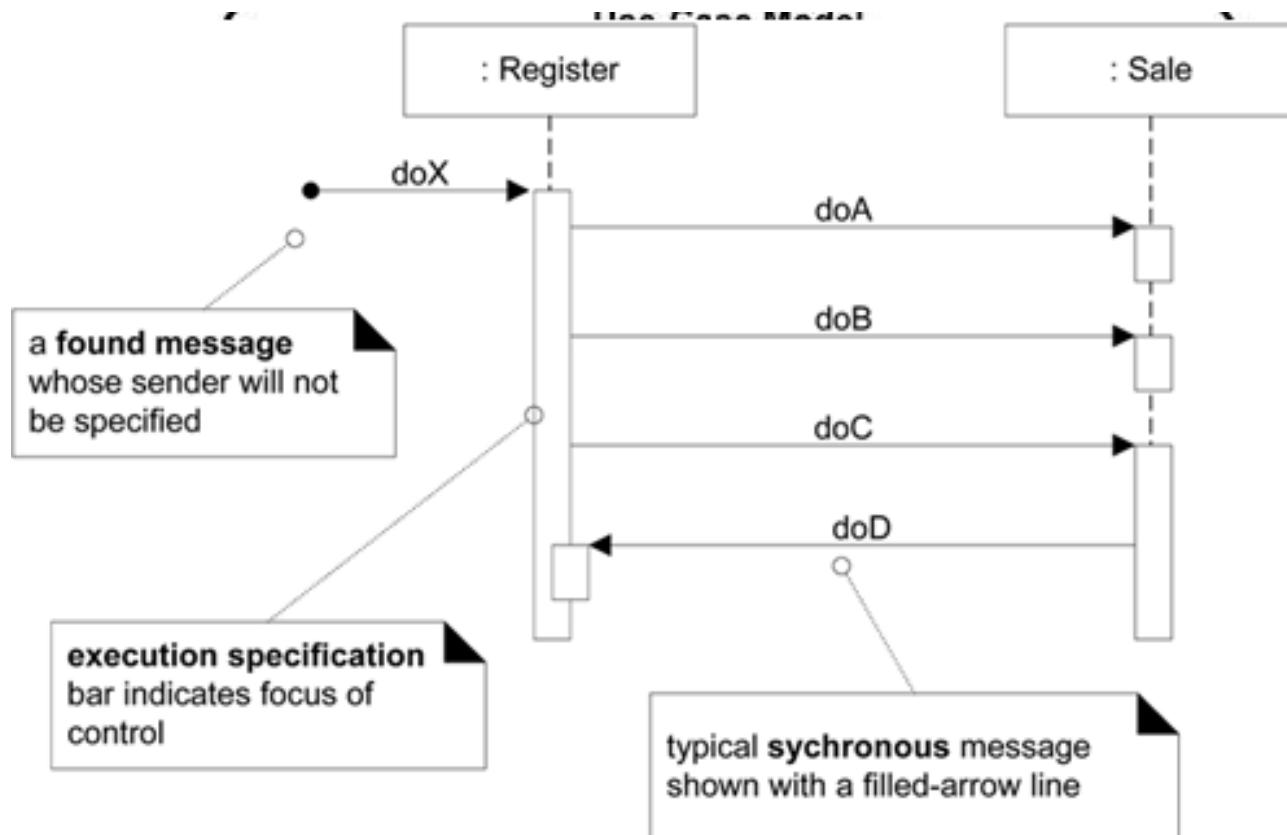
- Algemene message syntax
 - UML standaard:
 - Return = message (parameter:parameterType):returnType
- Voorbeelden
 - initialize(code)
 - Initialize (haakjes mogen weg indien geen param)
 - d = getProductBeschrijving(id)
 - d = getProductBeschrijving(id:ItemID)
 - d = getProductBeschrijving(id:ItemID):ProductDescription

Interaction sequence diagram

- Singleton objecten

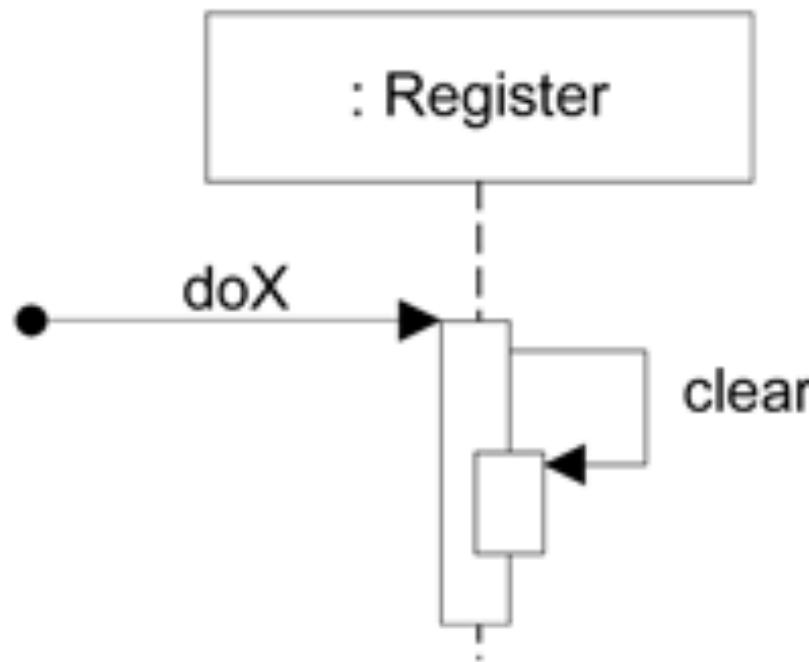


Interaction sequence diagram



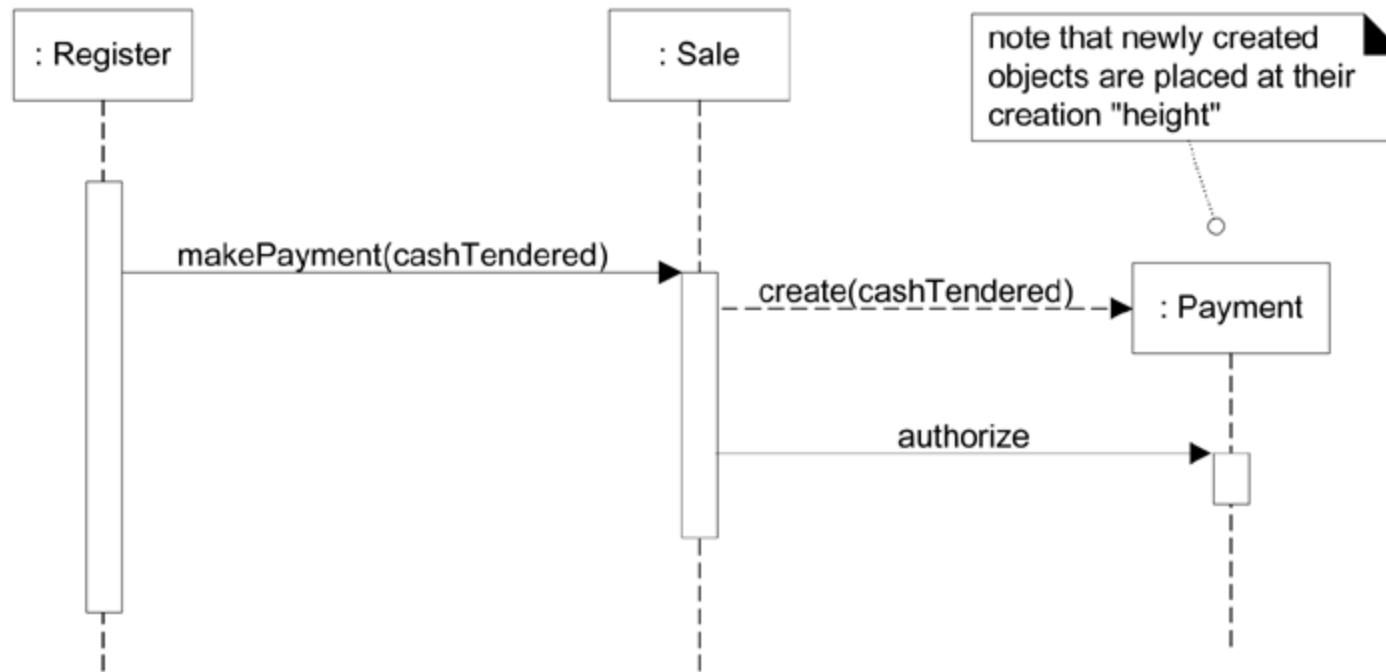
Interaction sequence diagram

- Sequence diagram notatie
 - Bericht naar eigen object



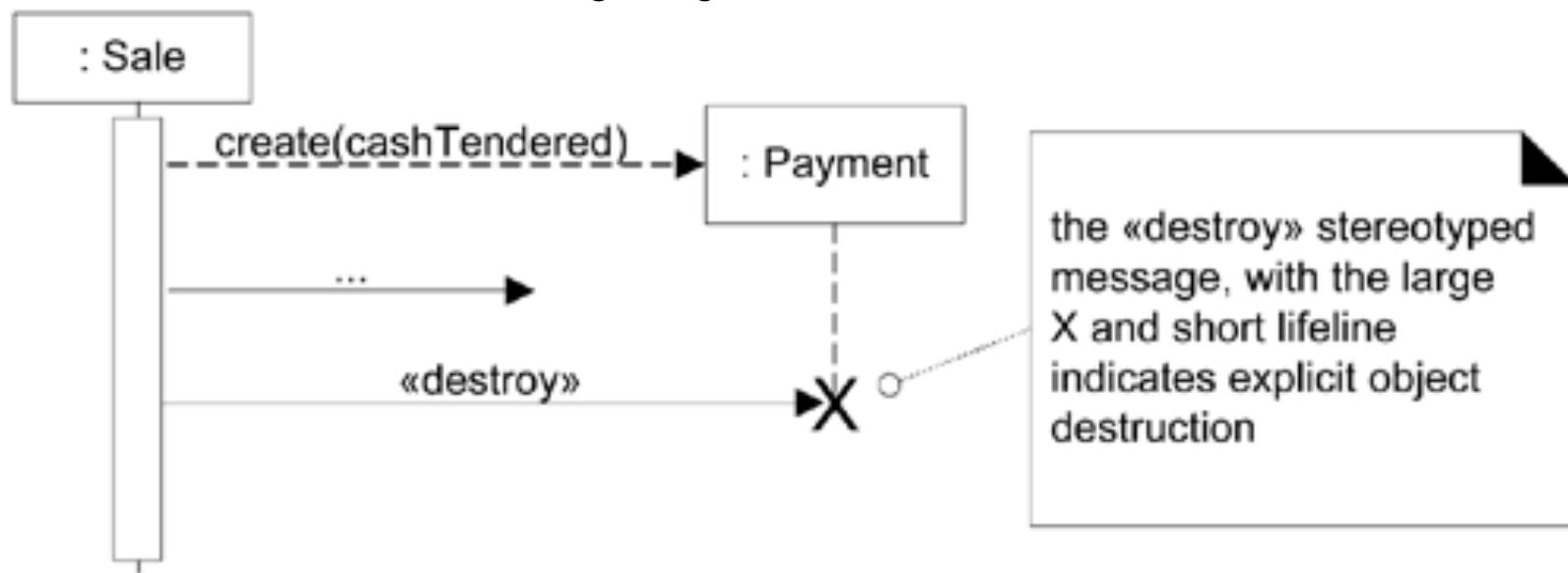
Interaction sequence diagram

- Sequence diagram notatie
 - Object creatie ("new")



Interaction sequence diagram

- Sequence diagram notatie
 - Object destructie
 - Enkel in talen zonder garbage collector



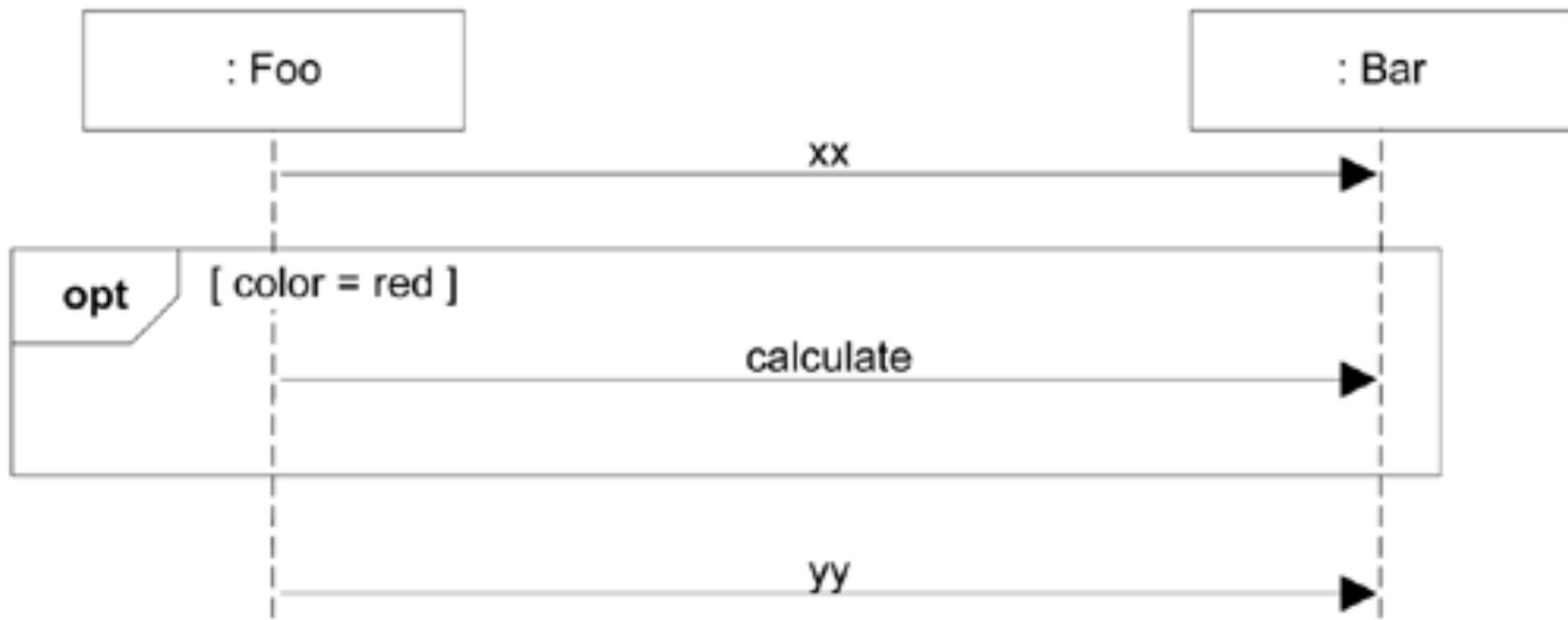
Interaction sequence diagram

- Sequence diagram notatie

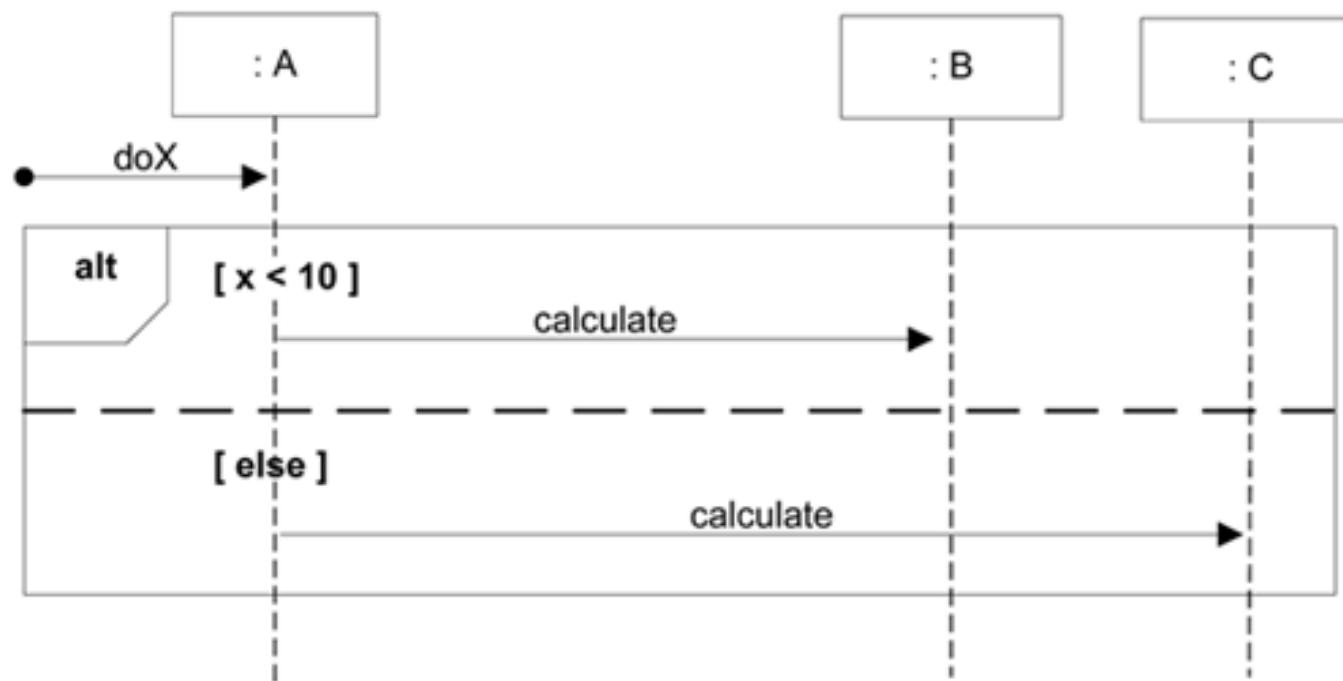


Interaction sequence diagram

- Condities

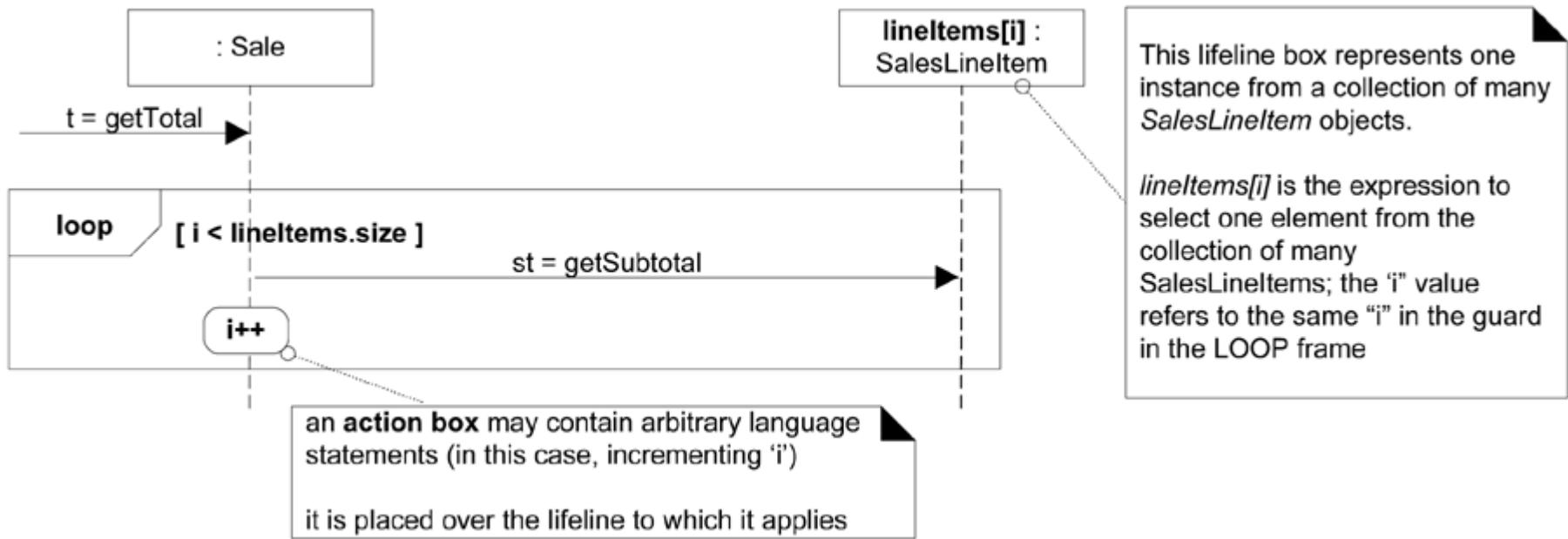


Interaction sequence diagram



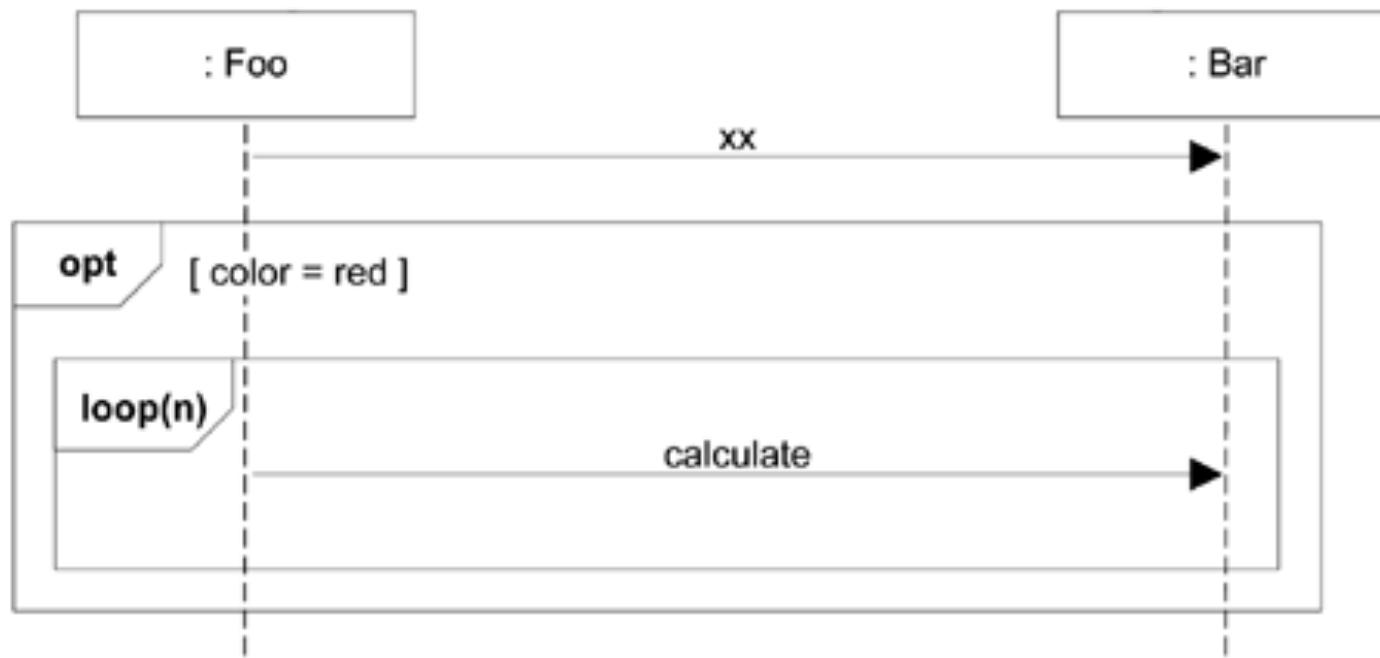
Interaction sequence diagram

- Iteratie over collecties



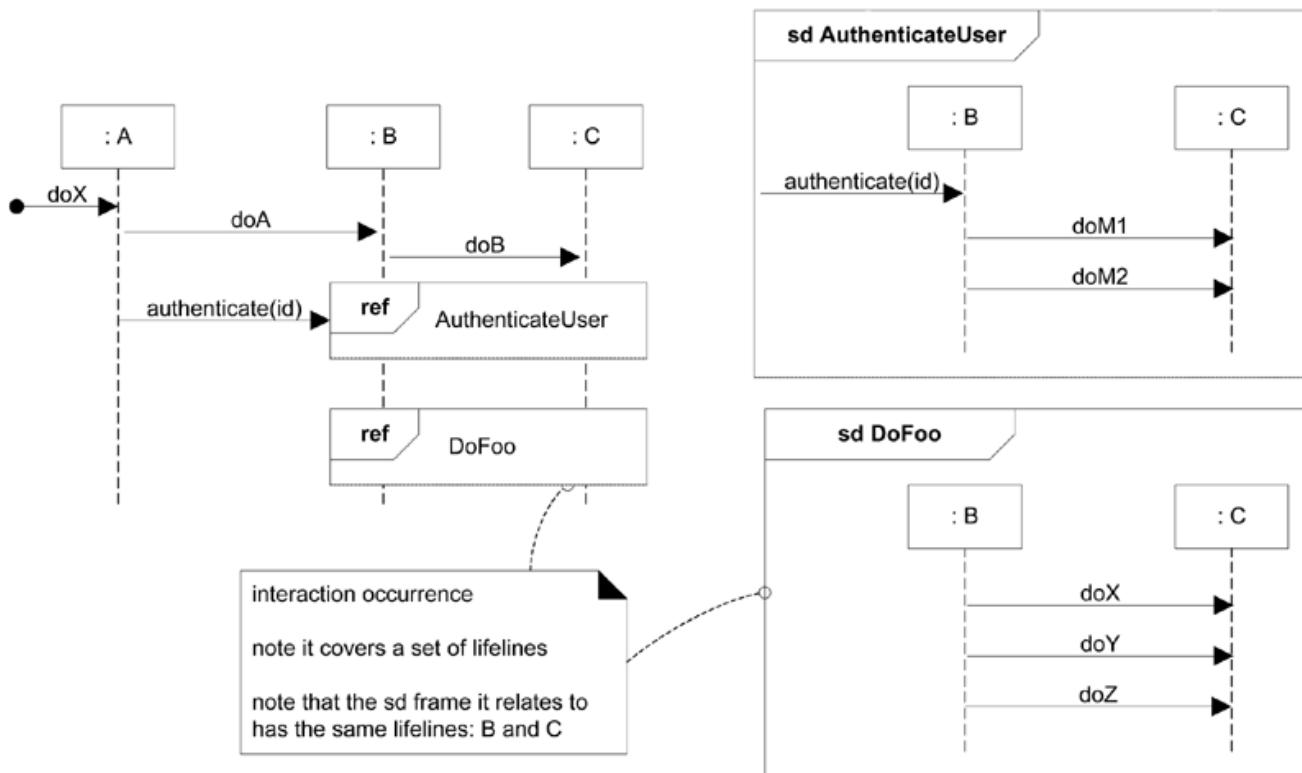
Interaction sequence diagram

- Nesten van frames



Interaction sequence diagram

- Referenties

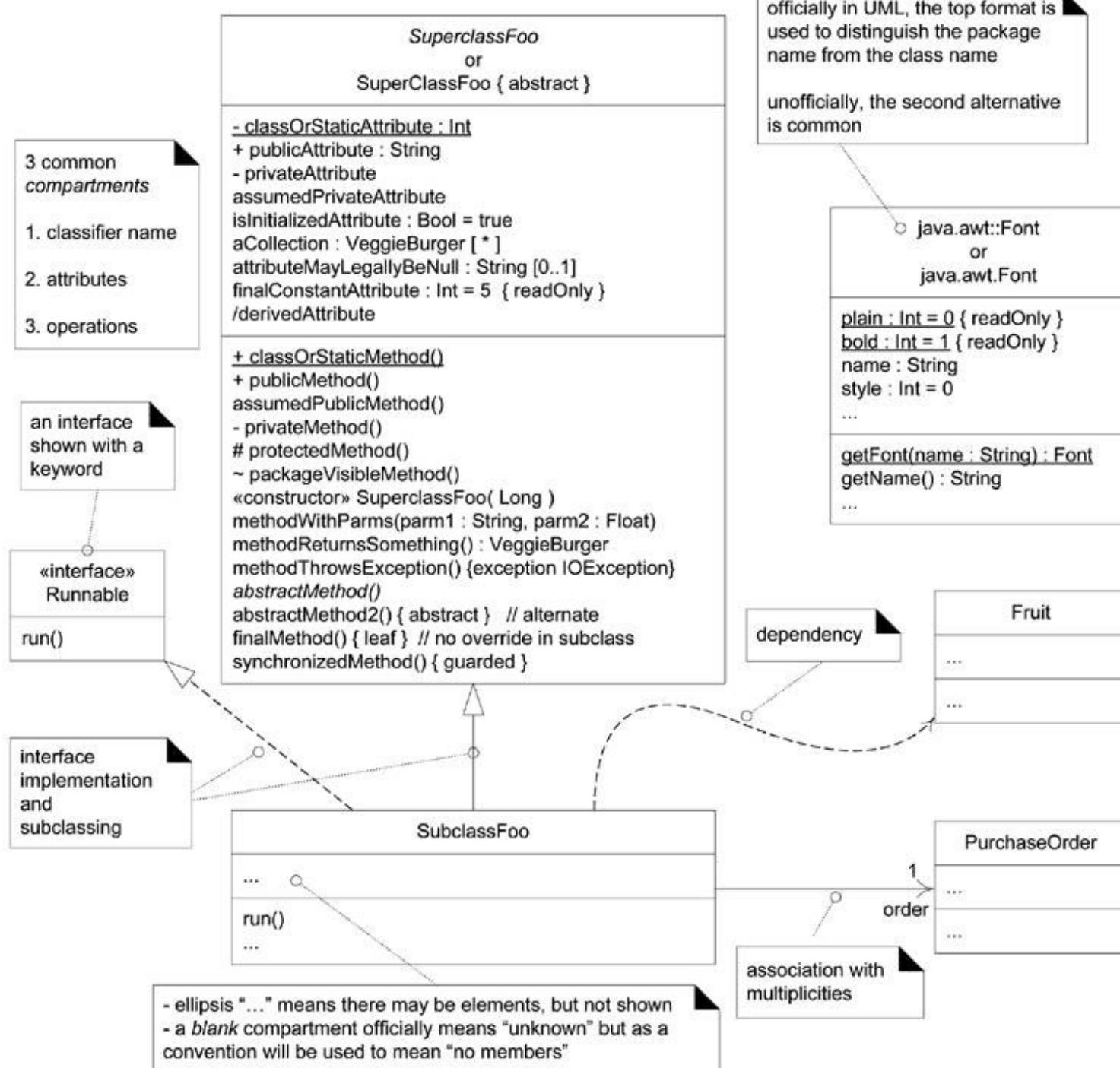


Class diagram



Class diagrams

- Wat is een class diagram?
 - Illustreren van klassen, interfaces en hun onderlinge associaties
 - Statische manier van modelleren
- UML notatie komt op 2 plaatsen voor
 - Domain Model (soms ook class diagram genoemd)
 - Design class diagram (DCD)



Class diagrams

Domain Model
conceptual perspective

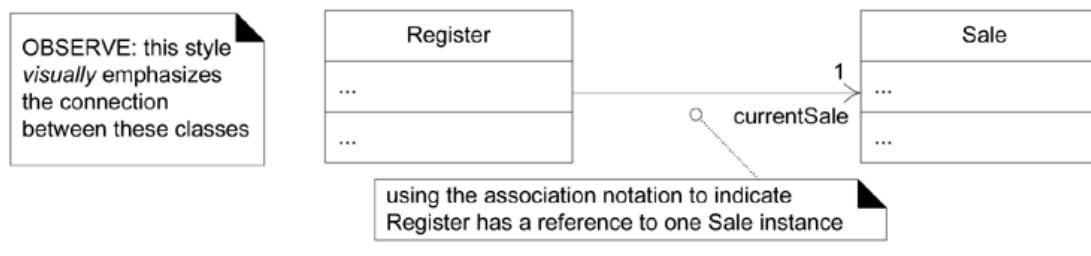


Design Model
DCD; software perspective



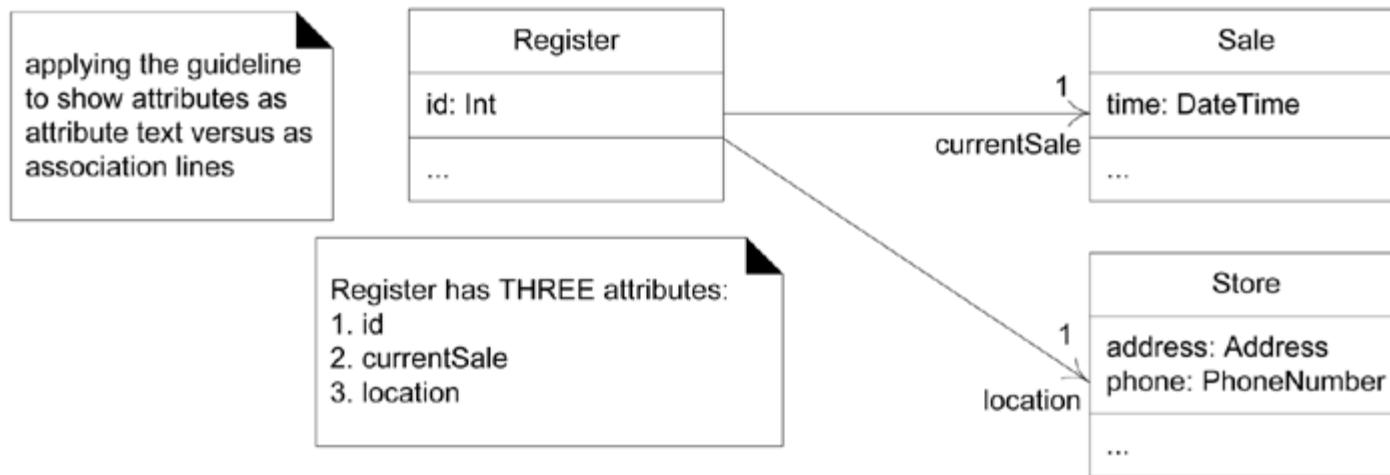
Class diagrams

- UML attributen
 - Attribute text vs. association lines



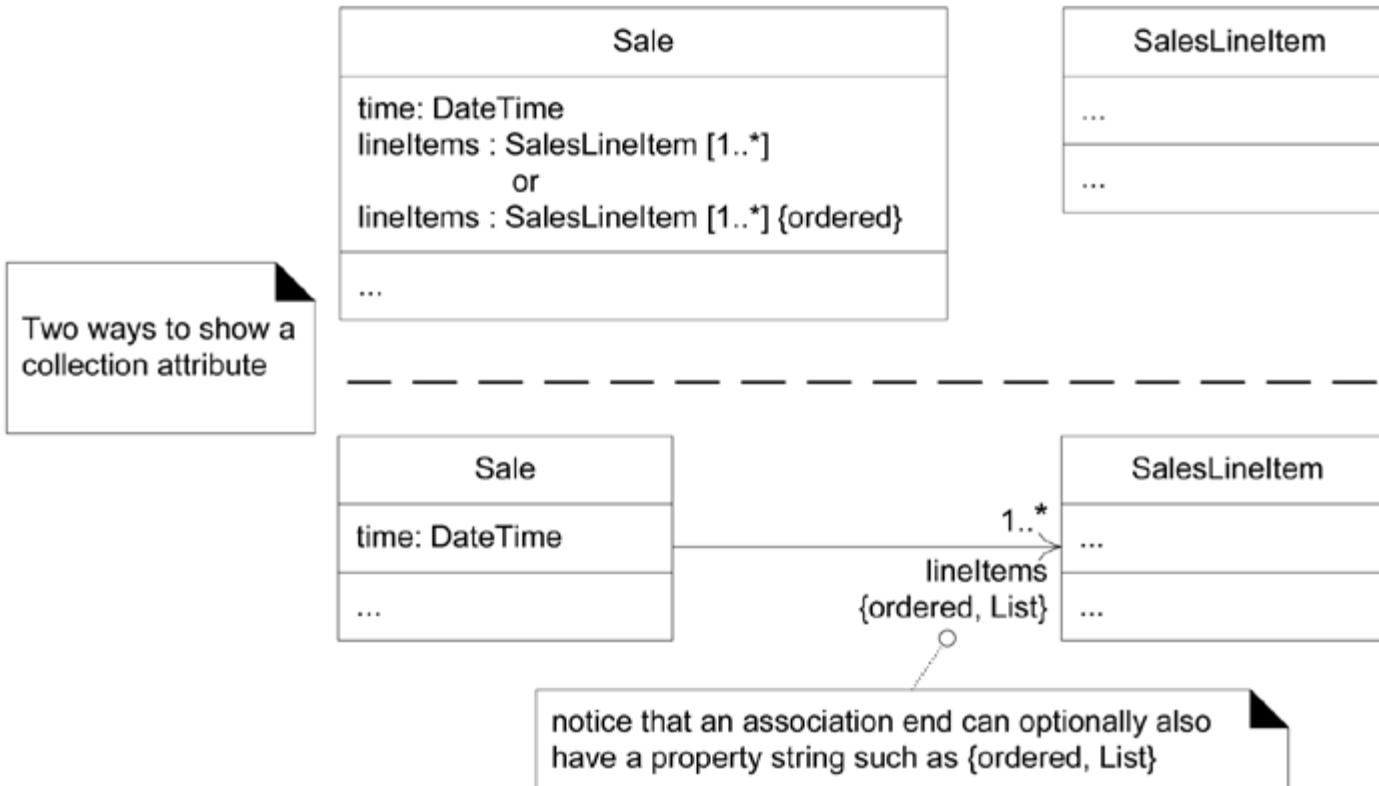
Class diagrams

- UML attributen
 - Attribute text vs. association lines
 - Tip
 - attribute text notatie voor data type objects (Boolean, Date, Time, Number, String, ...)
 - association lines voor alle andere attributen



Class diagrams

- UML attributen



Class diagrams

- UML attributen
 - Typische notatie
 - visibility name : type multiplicity = default {property-string}
 - Bv.
 - - aantalWielen: int = 4 {frozen}

Symbol	Verklaring
+	Public
-	Private
#	Protected
~	Package

Indien geen visibility dan
private

Taalspecifiek

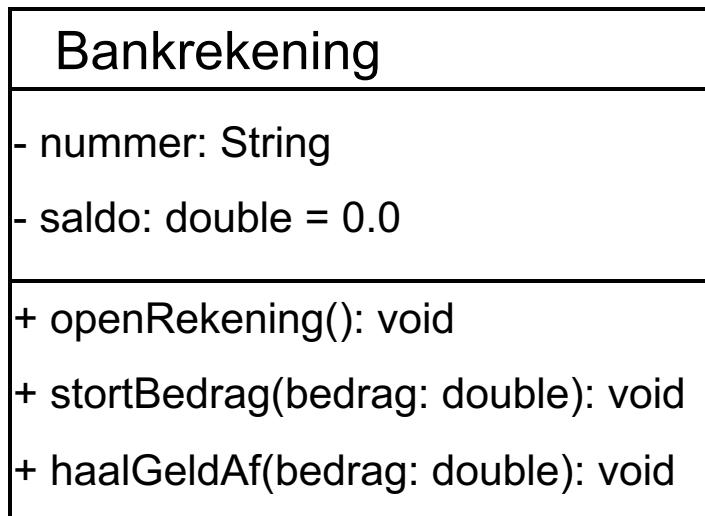
Bv. Java:

Protected: alleen zelfde package
+ vanuit subklassen

Package: alleen zelfde package

Class diagrams

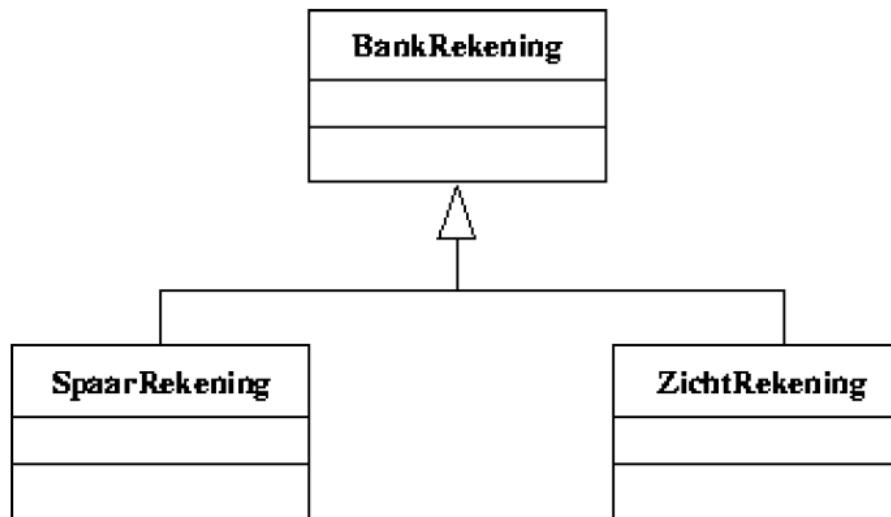
Code generatie



```
Public class Bankrekening{  
    private String nummer;  
    private double saldo;  
    public void openRekening() {  
        //code  
    }  
    public void stortBedrag(double bedrag) {  
        //code  
    }  
    public void haalGeldAf(double bedrag) {  
        //code  
    }  
}
```

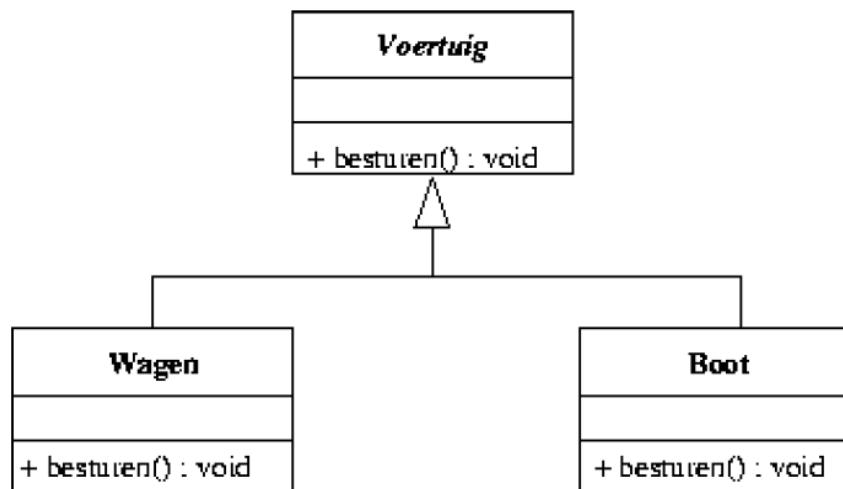
Class diagrams

- UML generalisatie, abstracte klassen, abstracte operaties
 - Overerving



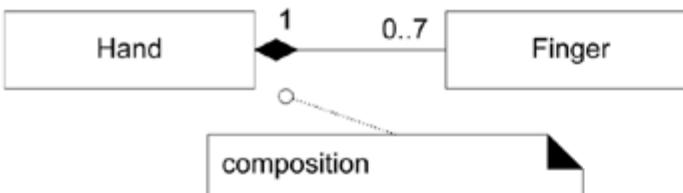
Class diagrams

- UML generalisatie, abstracte klassen, abstracte operaties
 - Abstracte superklasse



Class diagrams

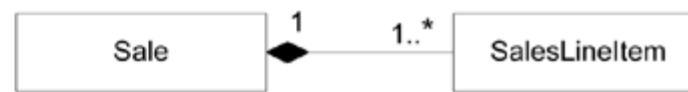
- UML composition relatie
 - Whole-part relatie (geheel-deel)
 - 3 voorwaarden
 - Instantie van deel behoort tot juist 1 geheel op gegeven tijdstip
 - Deel kan niet op zichzelf bestaan
 - Het geheel staat in voor het aanmaken van de delen



composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

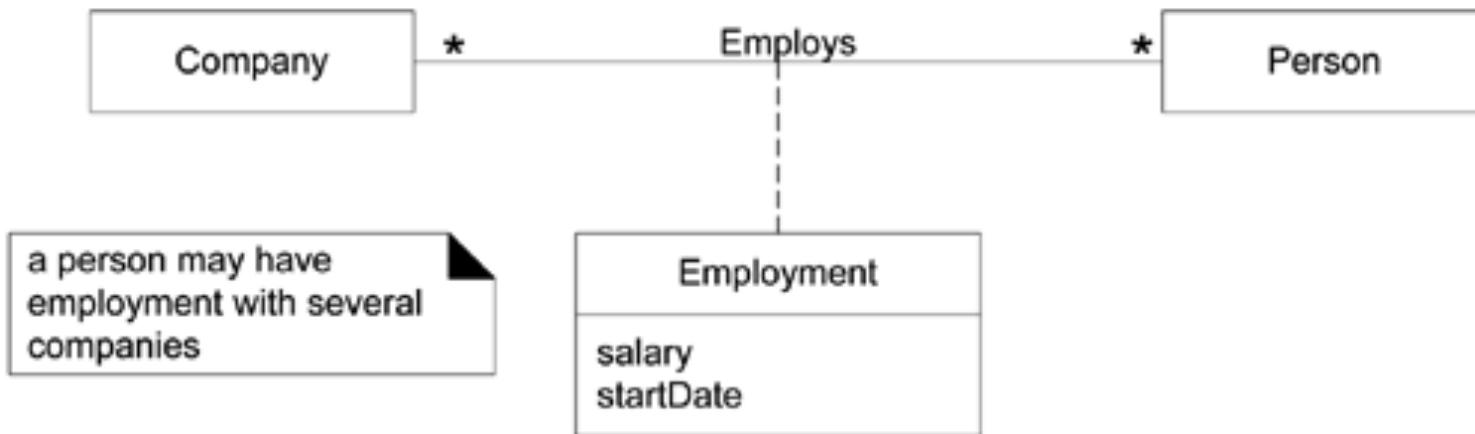
-the composite has sole responsibility for management of its parts, especially creation and deletion

Relatie is altijd **heeft-delen**
(niet noteren)



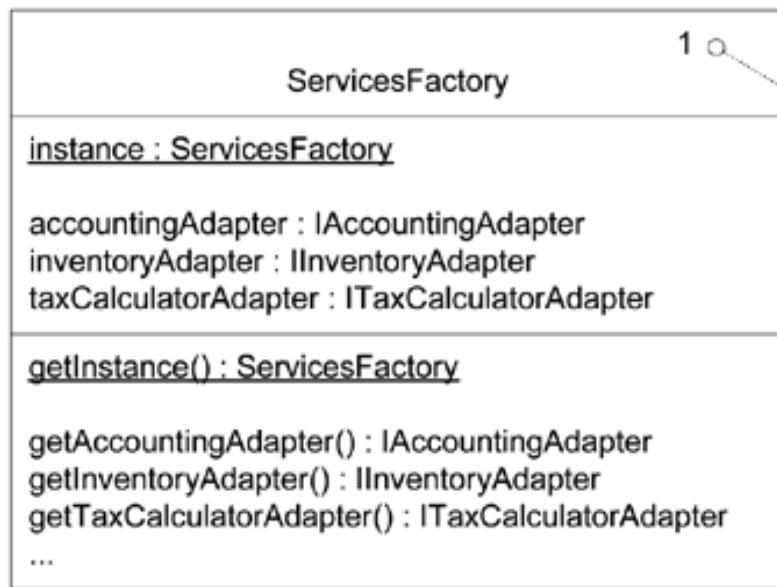
Class diagrams

- UML association class
 - Associatie zelf als klasse bekijken
 - Mogelijkheid tot toevoegen van attributen, operaties, ...



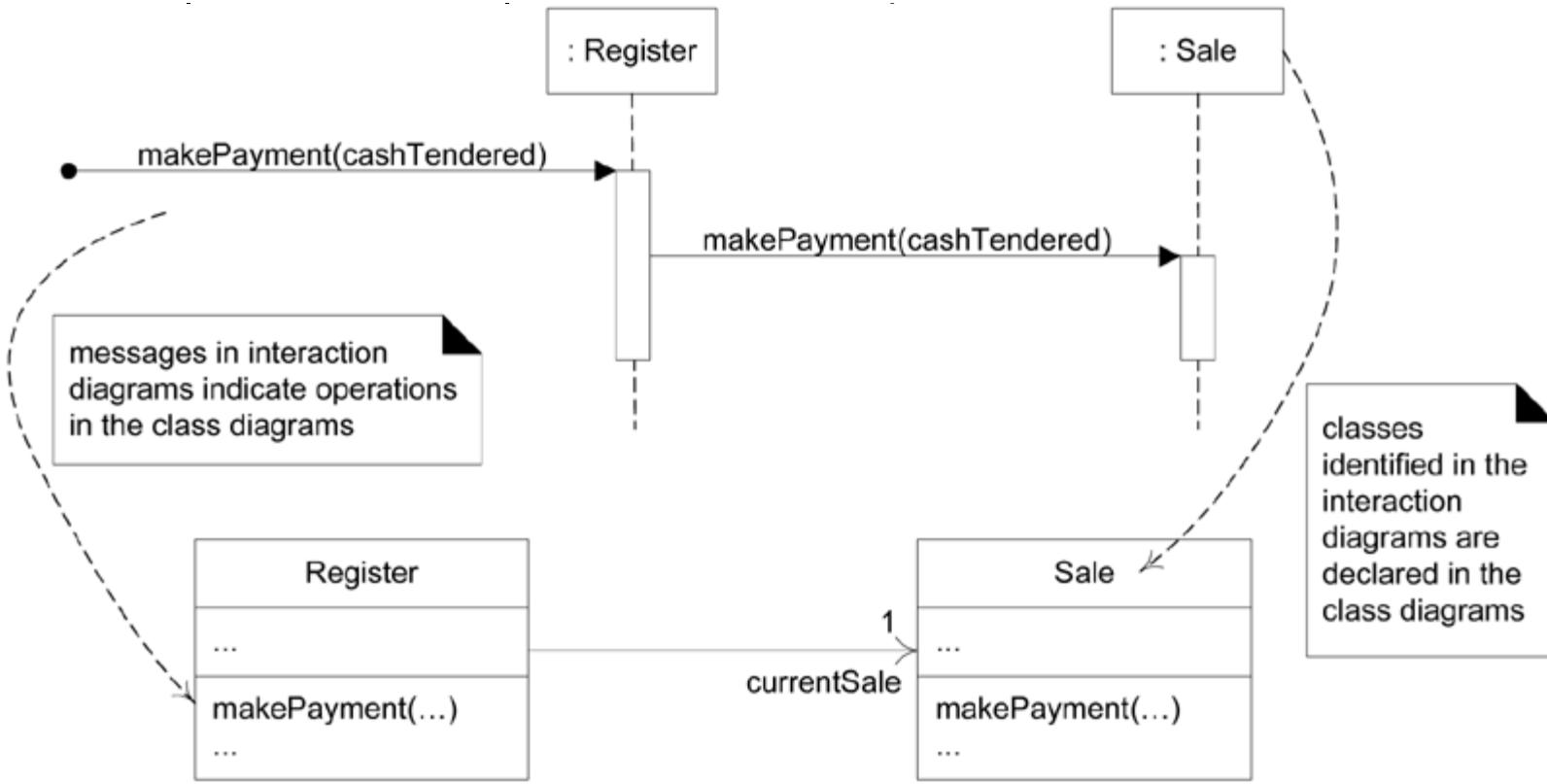
Class diagrams

UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member



UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

Class diagrams



State machine diagram

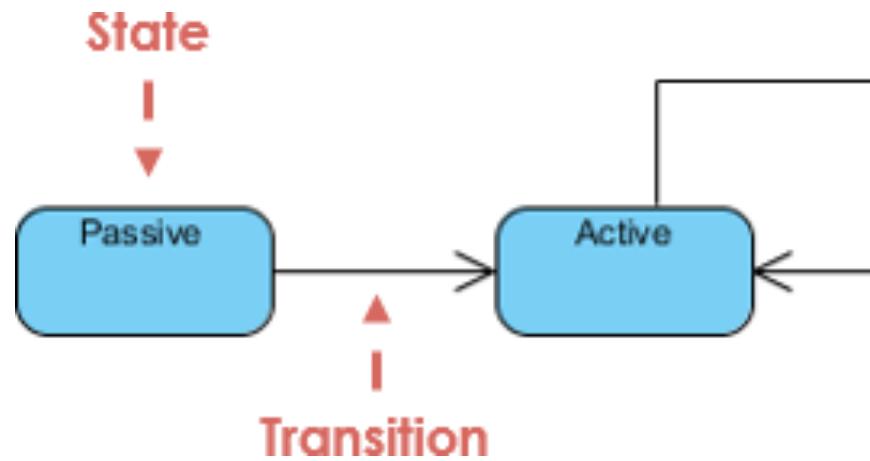


State machine diagrams

Wat is een state machine diagram?

- Een diagram dat de interessante events en statussen van een object weergeeft, alsook het gedrag van het object als respons op een event
- Beschrijving van toestanden en overgangen tussen deze toestanden in
 - Systeem
 - Domeinklassen
 - ...
- Kan gebruikt worden in verschillende fasen van het software-project

State machine diagrams



- State

bezet

- Transitiie

bezet → vrij

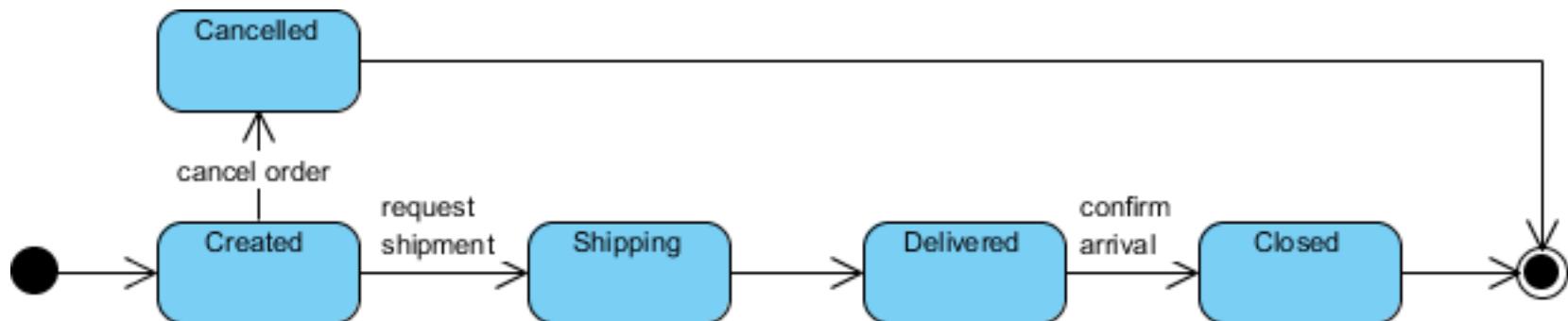
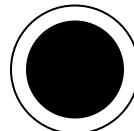
Hoorn afleggen

State machine diagrams

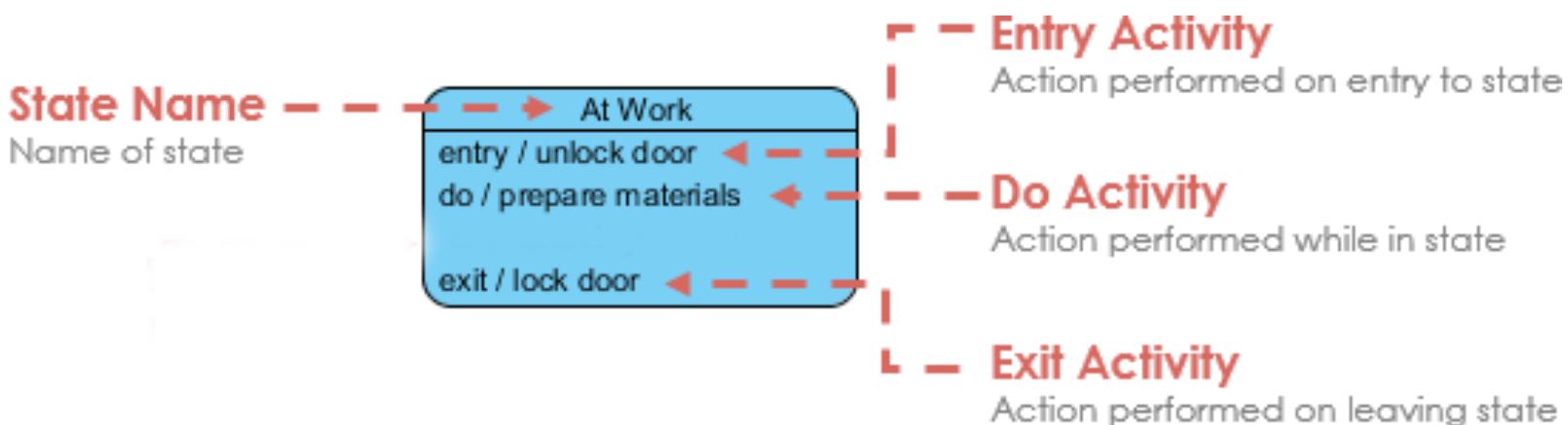
- Begintoestand



- Eindtoestand



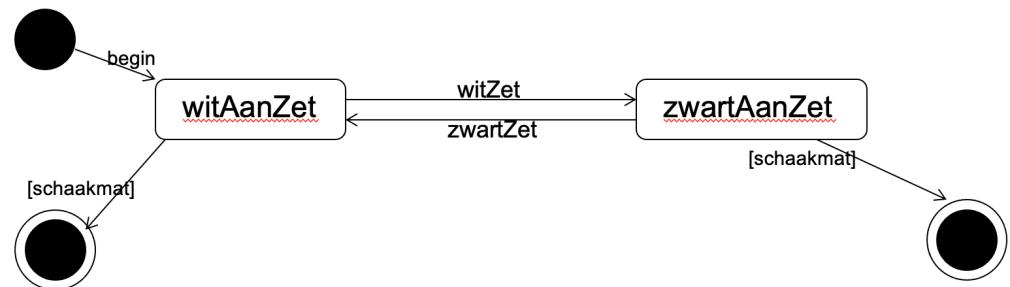
State machine diagrams



State machine diagrams

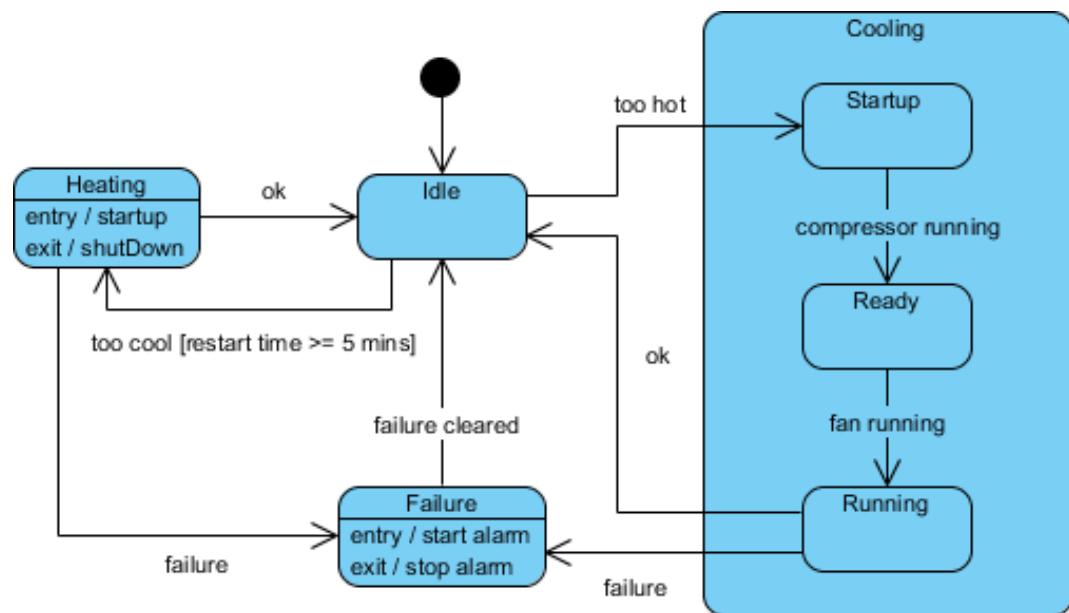
Guard

Voorwaarde op transitie
[voorwaarde]



State machine diagrams

SUBSTATES (NESTED
STATES)



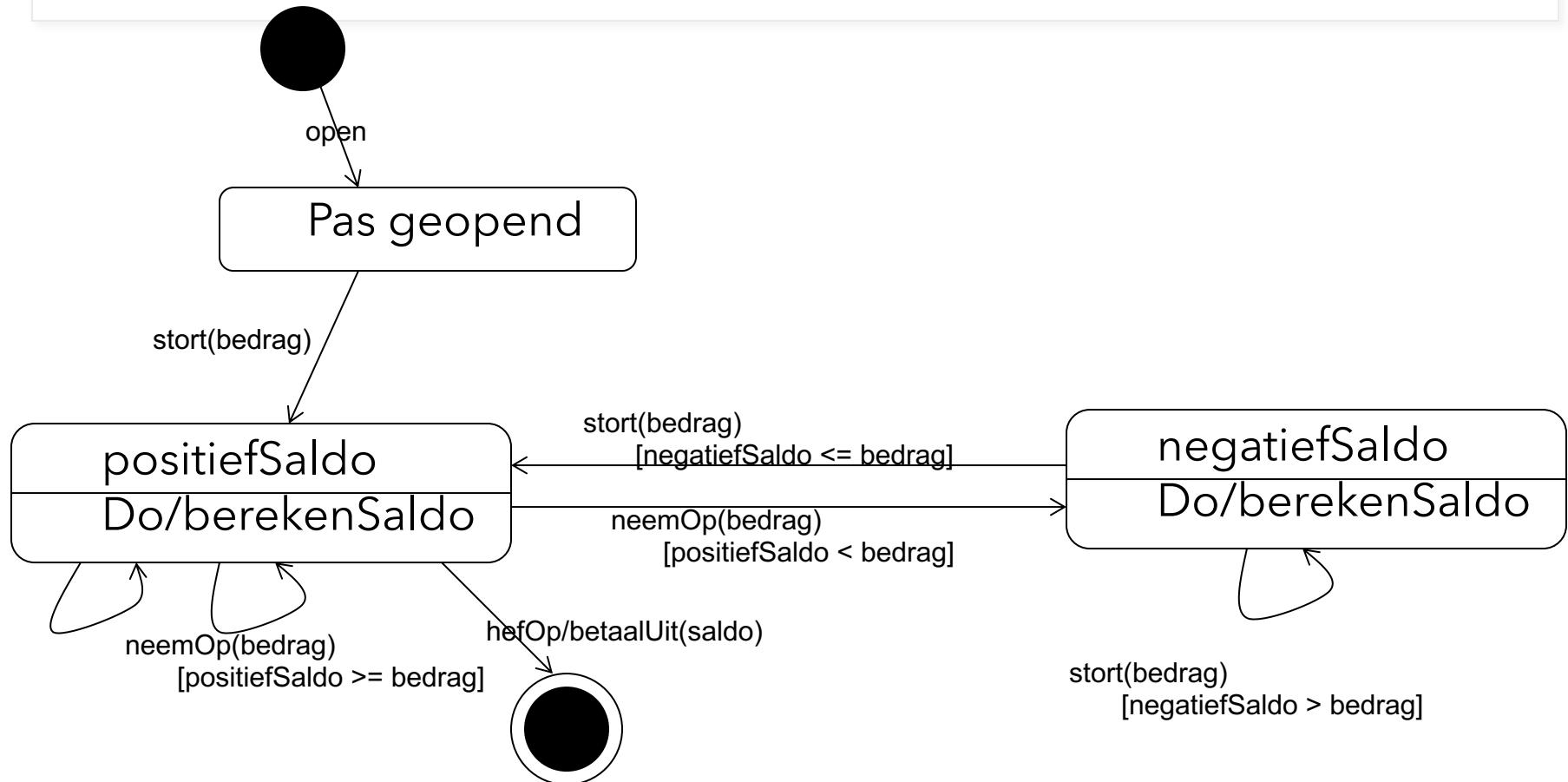
State machine diagrams

Werkwijze

Per (interessante) klasse een state diagram

1. Vind de toestanden van een object
2. Vind voor iedere event de bijhorende transitie
3. Voeg begin- en eindtoestanden toe
4. Voeg acties toe
5. Voeg activiteiten toe
6. Voeg eventueel overige informatie toe
7. Itereer over de uitgevoerde stappen

State machine diagrams





Software Design & Quality Assurance

Johan van den Broek

Overzicht



INLEIDING



GRASP



GOF
PATTERNS

Inleiding

- Wat zijn design patterns?
 - Typische manieren hoe
 - Objecten
 - Klassen
- Wiel niet heruitvinden
 - Patronen zijn gebaseerd op ervaringen van software-projecten over de hele wereld

Inleiding

- Wat zijn design patterns?
 - Patterns moet je leren (studeren!)
 - Om ze te herkennen
 - Om ze technisch te kunnen toepassen
 - Hergebruik van ervaring
 - Hoger niveau dan object-oriented ontwikkeling

Design patterns

- = Gemeenschappelijke vocabulaire
 - Krachtig middel voor communicatie
 - Meer zeggen met minder woorden
 - Discussie op ontwerp niveau
- Denken op pattern-niveau, niet op object-niveau
- Ook bv. Java API gebruikt dikwijls patterns
 - Begrijpen patterns = begrijpen API's

Design patterns

- Design patterns
 - ≠ UML
 - ≠ technologie
- UML = enkel taal voor modellering
 - Eens je weet wat je wil ondersteunt UML modellen
 - Maar we moeten eerst weten wat we willen...
= Object design

Object Oriented Software Design

- Hulpmiddelen
 - GRASP: General Responsibility Assignment Software Patterns
 - GoF design patterns
 - RDD: Responsibility Driven Design
- Output van design fase
 - UML interaction en class diagrams, package diagrams
 - UI sketches en prototypes
 - Database Models

Object Oriented Software Design

- Responsibilities en RDD
 - Belangrijke termen in software design
 - Responsibilities (verantwoordelijkheden)
 - Roles (rollen)
 - Collaborations (samenwerking)

Deze termen horen thuis in een groter framework

= RDD (Responsibility Driven Design)

Object Oriented Software Design

- Responsibilities en RDD
 - Duidelijk onderscheid tussen



- **Doing**

- Als object iets zelf doen, bv. objectcreatie of berekening
 - Een actie in andere objecten opstarten
 - Beheren en coördineren van activiteiten objecten

- **Knowing**



- Private geëncapsuleerde gegevens kennen
 - Gerelateerde objecten kennen
 - Weten hoe zaken afgeleid of berekend kunnen worden

Object Oriented Software Design



• Responsabilities en RDD

- Tip: Het domain model is omwille van zijn attributen en associaties een goede inspiratiebron voor het gedeelte **knowing** van een object
- Vertaling van responsabilities → klassen/methoden hangt af van granulariteit
 - Grote verantwoordlkh → honderden klassen/methoden
 - Kleine verantwoordlkh → 1 methode

Object Oriented Software Design

- Responsabilities en RDD
 - RDD gebruikt ook samenwerkingen
 - Methoden hebben andere objecten/methode nodig
 - Bv. *Sale*: `getTotal()` methode heeft `getSubtotal()` methode nodig uit *SalesLineItem*
 - RDD werkt zoals in het echte leven: bepaalde mensen hebben verantwoordelijkheden, ze moeten dikwijls samenwerken om iets gedaan te krijgen



Object Oriented Software Design

- Responsibilities en RDD
 - Relatie tussen responsibility, GRASP en UML?
 - **Responsibilities** van objecten vinden tijdens coding/modeling
 - **UML** diagram laat ons nadenken over responsibilities
 - **GRASP** helpt ons bij toewijzen van responsibilities aan objecten
 - Je past dus GRASP toe bij tekenen van UML diagrams en bij coderen
 - **GoF patterns** zijn nog meer geavanceerde patronen

Preamble

- Each class should do only one predefined task/responsibility.
- Questions you should ask when writing code
 - Is it the right place to put this here? Is it my job to know that kind of information? Do I have the responsibility to do this? Why does this element have to make that decision? ...

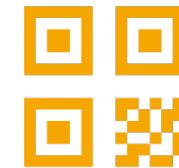
Overzicht



Inleiding



GRASP



GoF patterns

GRASP

- Er bestaan 9 GRASP patronen
 - Creator
 - Information Expert
 - Low Coupling
 - Controller
 - High Cohesion
 - Polymorphism
 - Pure Fabrication
 - Indirection
 - Protected Variations

GRASP: Low Coupling

Probleem

- Hoe "lage" afhankelijkheid ondersteunen?
- Hoe "change impact" minimaliseren?
- Hoe herbruikbaarheid verhogen?

Oplossing

- Deel verantwoordelijkheden uit zodat coupling laag blijft
- Gebruik dit principe om keuzes te maken

GRASP: Low Coupling



Wat is coupling?

Maat die aangeeft hoe sterk een element gekoppeld (afhankelijk) is van andere elementen

Element: klasse, subsysteem, systeem, ...



Nadelen high coupling

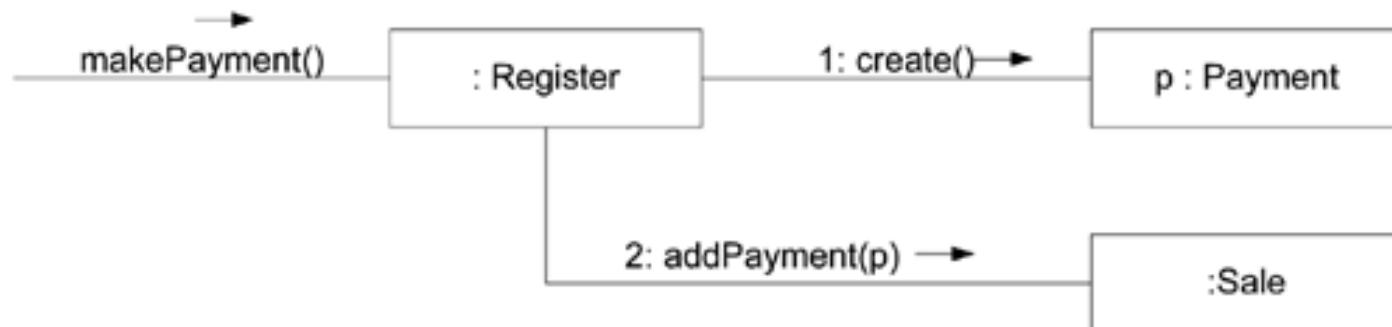
Lokale veranderingen door veranderingen in andere klassen

Klasse moeilijker op zich te begrijpen

Moeilijker herbruik van klassen

GRASP: Low Coupling

- Voorbeeld
 - Probleem: we moeten een Payment-instantie aanmaken en deze aan een Sale associëren
 - Indien we Creator patroon volgen



GRASP: Low Coupling

- Voorbeeld
 - Indien we Low Coupling patroon volgen



- Enkel Sale is gekoppeld aan Payment, Register is hier niet gekoppeld aan Payment dus Lower Coupling!

GRASP: Low Coupling

- Low Coupling is een algemeen principe
 - In verschillende design beslissingen in achterhoofd houden
 - Vooral bij overlopen van alternatieven
 - Komt voor in andere design patronen als basis
- **Coupling** tussen klassen betekent
 - Referentie bijhouden naar (member), methode oproepen, subklasse, implementeren van interface, ...

GRASP: Low Coupling

Overerving

- Overerving is voorbeeld van zeer sterke coupling
 - **Bv.** Bij persistentie: alle klassen laten overerven van persistentie-superklasse
 - Nadeel: domain objects koppelen aan technisch aspect
 - Zeer hoge coupling!

Absolute maat is niet mogelijk

- Er is steeds coupling nodig
- Minimaliseren door overlopen van design mogelijkheden

GRASP: Low Coupling



Niet overdrijven

Beste "low coupling" is geen coupling!

- Geen associaties
- Losstaande klassen die berichten uitwisselen
- Gevolg: grote, complexe klassen met veel te veel functionaliteit



Contra-indicaties

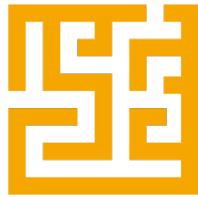
Java libraries: zijn stabiel over lange tijd

- Weinig kans dat hier wijzigingen zullen optreden

GRASP: Low Coupling

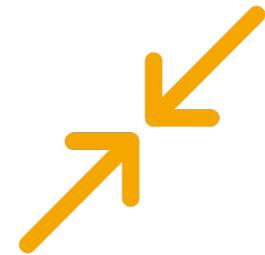
- Voordelen
 - Wijzigingen blijven “lokaal” en beperkt indien er iets moet veranderd worden
 - Eenvoudig te begrijpen indien klasse geïsoleerd wordt
 - Handig voor hergebruik

GRASP: High Cohesion



Probleem

Hoe objecten verstaanbaar,
gefocust, beheerbaar houden?



Oplossing

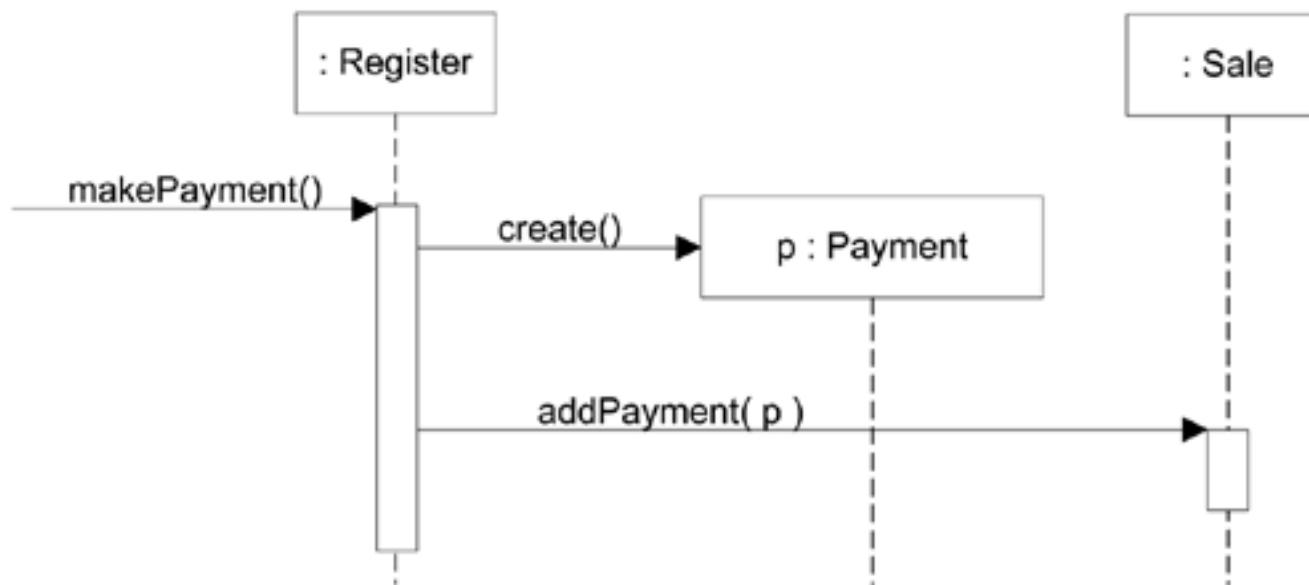
Verantwoordelijkheden uitdelen
zodat cohesie hoog blijft. Gebruik
dit principe voor het afwegen van
alternatieven

GRASP: High Cohesion

- Wat is cohesion?
 - Een maat die aangeeft hoe sterk gerelateerd en gefocust de verantwoordelijkheden van een element zijn (element = klasse, subsysteem, systeem, ...)

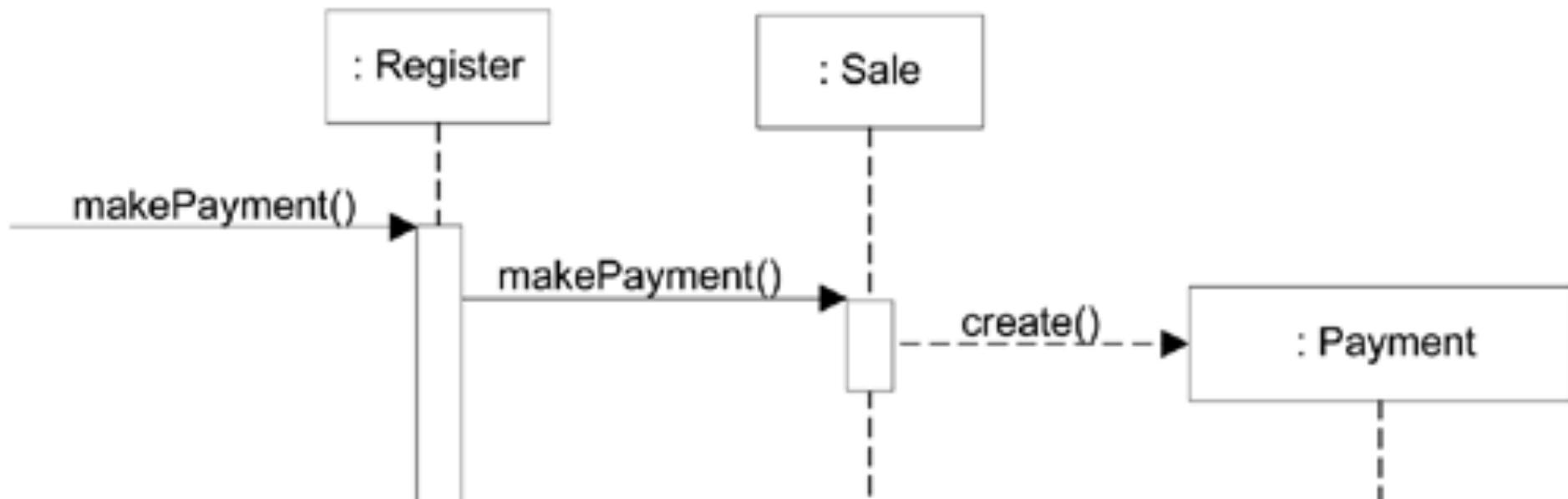
GRASP: High Cohesion

- Voorbeeld
 - Indien “Register” al het werk voor vele operaties op zich neemt, krijgen we geen cohesie meer binnen die klasse



GRASP: High Cohesion

- Voorbeeld
 - Goede oplossing is ook goede oplossing voor low coupling

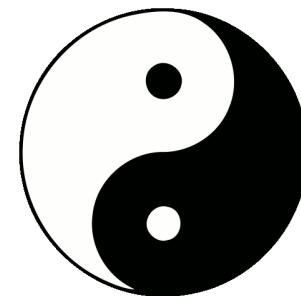


GRASP: High Cohesion

- Graden van cohesie
 - **Zeer laag:** Een klasse die verantwoordelijk is voor vele zaken in verschillende functional areas
 - **Laag:** 1 klasse heeft volledige verantwoordelijkheid voor een complexe taak
 - **Hoog:** een klasse heeft gemiddelde verantwoordelijkheden in 1 functional area en werkt samen met andere objecten

GRASP: High Cohesion

- Cohesion en coupling
= YIN en YANG



Slechte cohesion → slechte coupling

Slechte coupling → slechte cohesion

GRASP: High Cohesion

- Contra-indicaties
 - Eenvoudiger onderhoud
 - Bv. SQL statements samenzetten
 - Distributed server objects
 - Toch meer functionaliteit toelaten
 - Om te veel netwerkcommunicatie te vermijden

GRASP: High Cohesion

- Voordelen
 - Duidelijkheid/klaarheid van ontwerp
 - Onderhoud en verbeteringen
 - Low coupling ondersteunen

GRASP: Creator

Probleem

Wie maakt een nieuwe klasse-instantie aan?

Oplossing

Laat klasse B instanties maken van klasse A

- indien B de klasse A "bevat" of er is uit opgebouwd

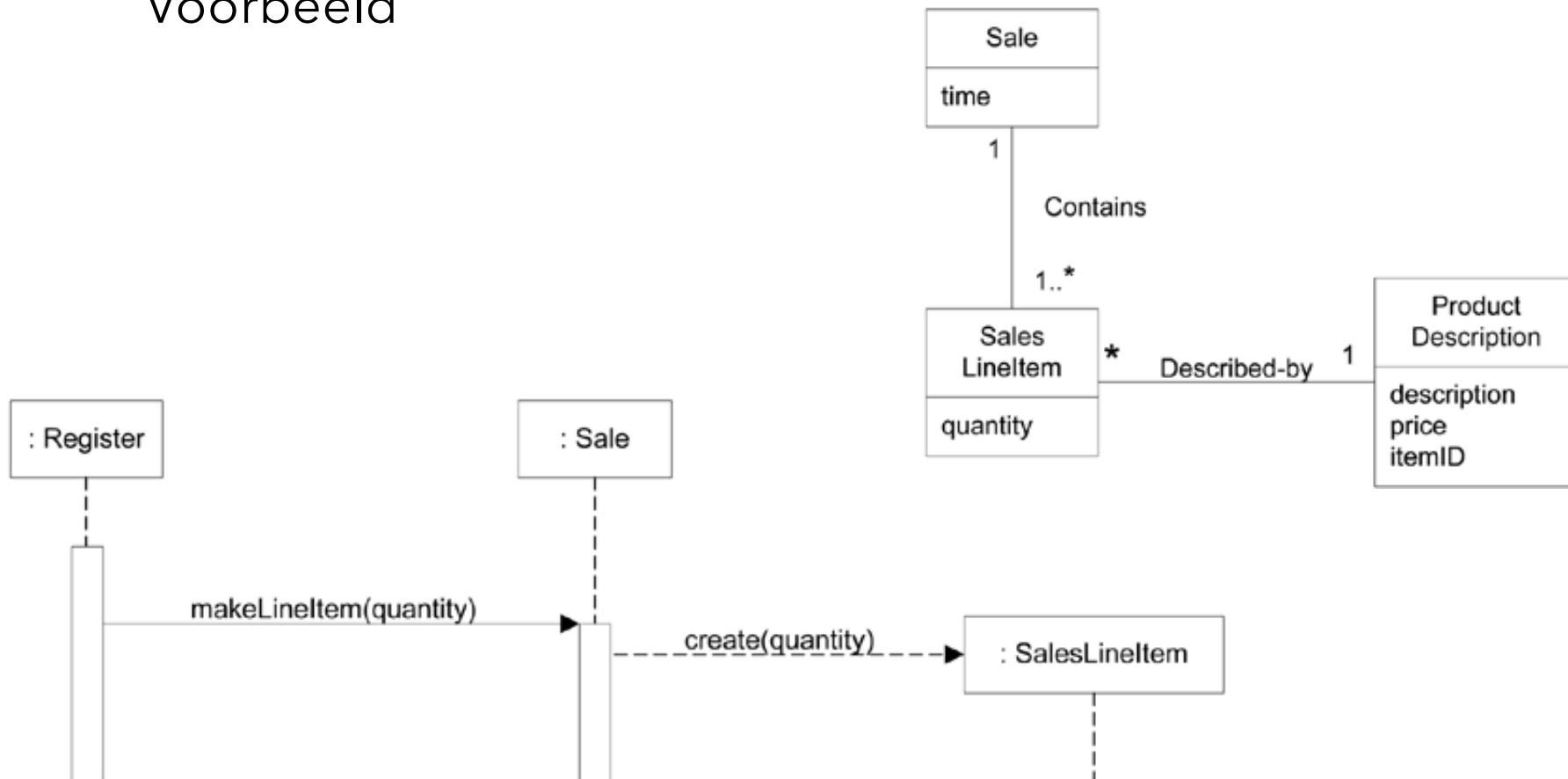
- indien B de toestand van klasse A moet bewaren

- indien B vaak klasse A gebruikt

- indien B de informatie heeft die nodig is voor creatie van A

GRASP: Creator

Voorbeeld



GRASP: Information Expert

Probleem

Hoe beslis je over toekennen van verantwoordelijkheden aan klassen?

Oplossing

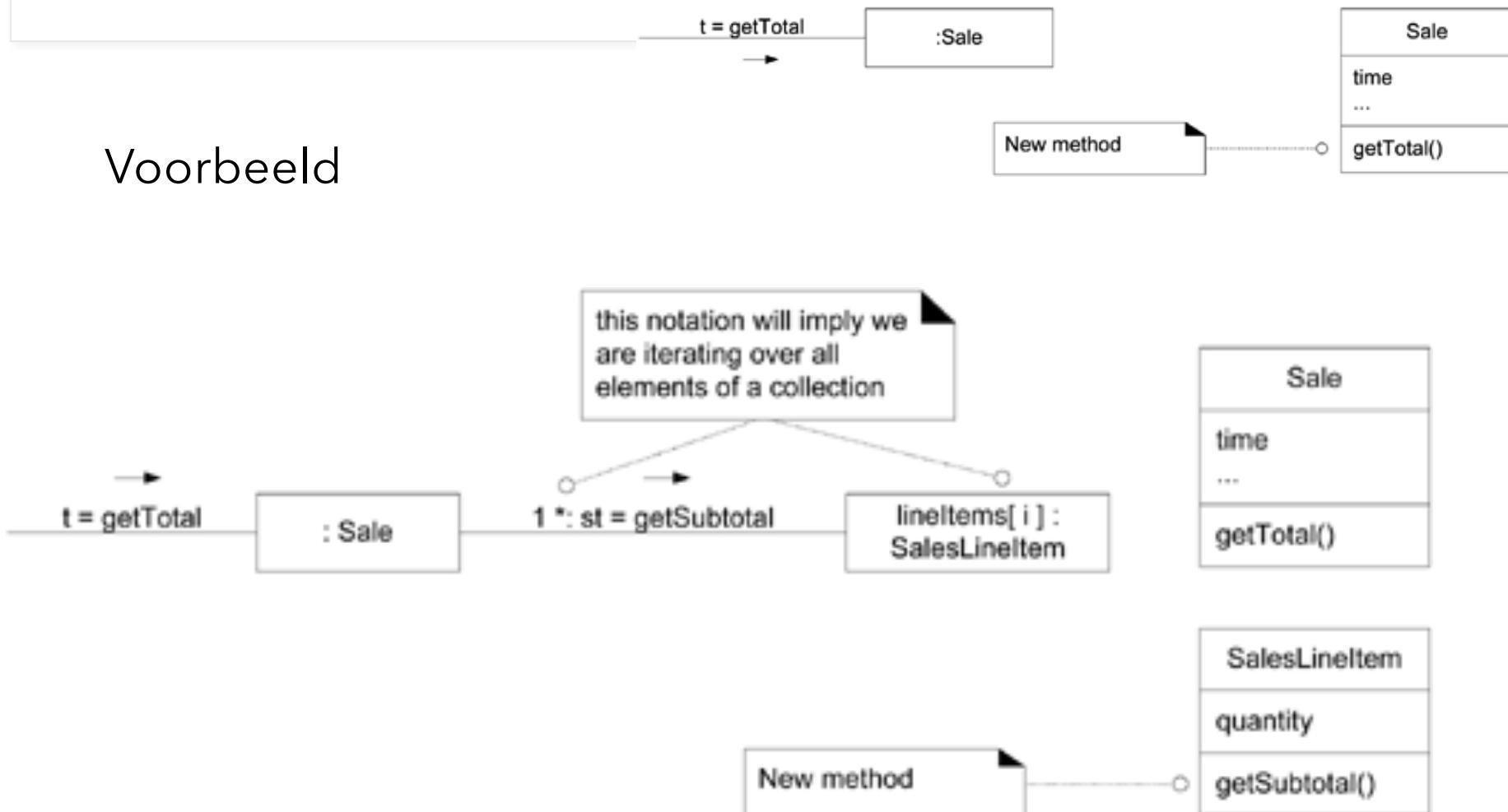
Geef verantwoordelijkheid aan de "information expert" klasse, de klasse die de **nodige informatie** bevat om de verantwoordelijkheid op te nemen



Tip: begin al met het duidelijk omschrijven van verantwoordelijkheden

GRASP: Information Expert

Voorbeeld



GRASP: Information Expert



GRASP: Information Expert

Voorbeeld: overzicht

Design Class	Responsibility
Sale	Sale total
SalesLineItem	Line item subtotal
ProductDescription	Product price

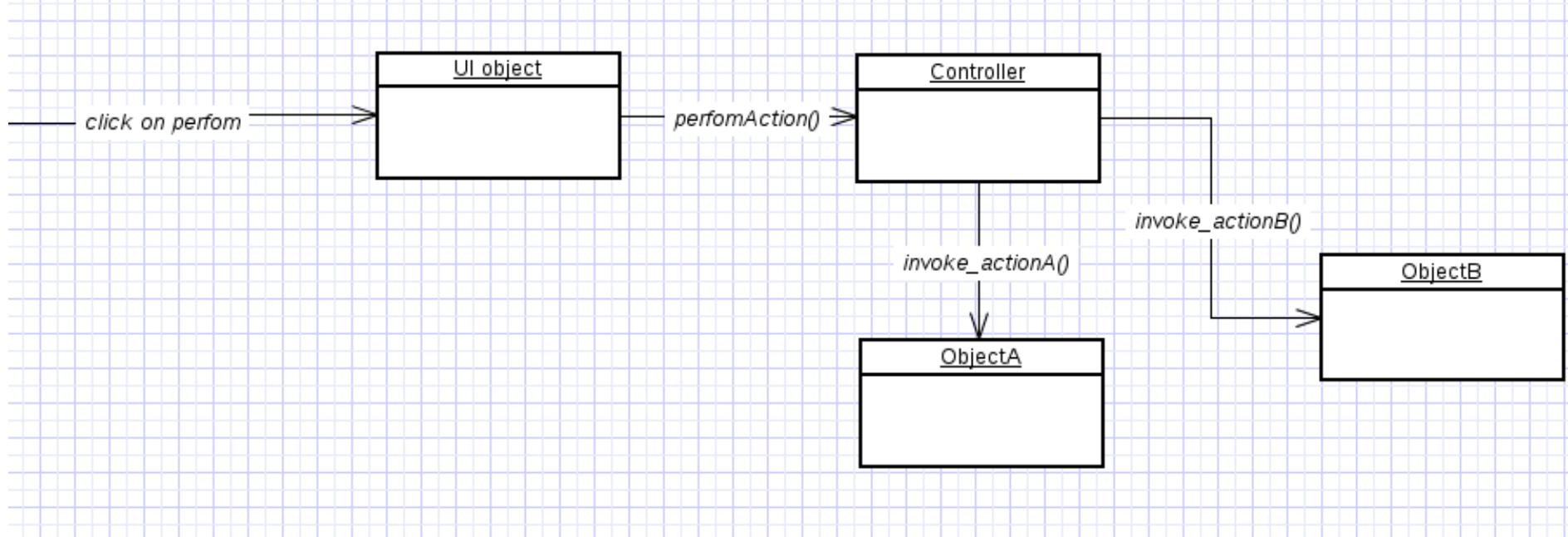
GRASP: Information Expert

- Voordelen
 - Information encapsulation
 - Object weet nodige info (encapsuleert info) om taak te volbrengen
 - Gedrag wordt verspreid over verschillende klassen
 - Voordeel: lightweight klassen

GRASP: Controller

- Het controllerpatroon wijst de verantwoordelijkheid voor het omgaan met systeemgebeurtenissen toe aan een niet-UI-klasse die het algehele systeem of een gebruiksscenario vertegenwoordigt.
- Een controller-object is een niet-gebruikersinterface-object dat verantwoordelijk is voor het ontvangen of afhandelen van een systeemgebeurtenis.

GRASP: Controller



GRASP: Controller

Probleem

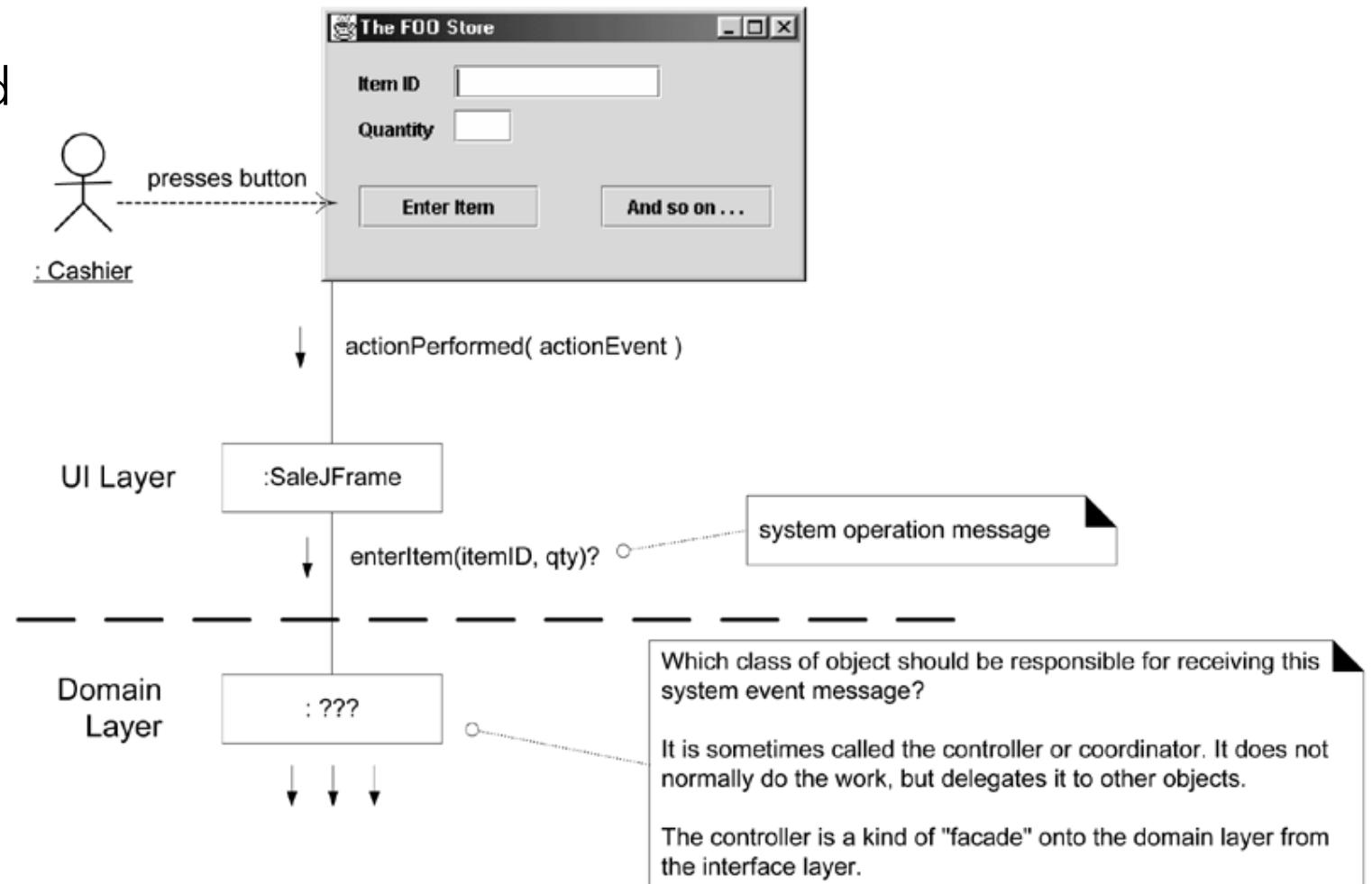
Wat is het eerste object dat onder de UI layer de system operations opvangt en coördineert?

Oplossing (2 opties)

1. Klasse die het hele onderliggende systeem representeert
(Facade Controller)
2. Klasse die use case scenario representeert
bv. <UseCaseName>Handler

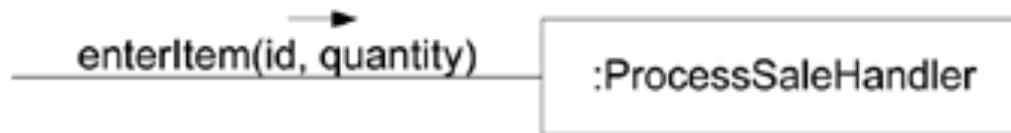
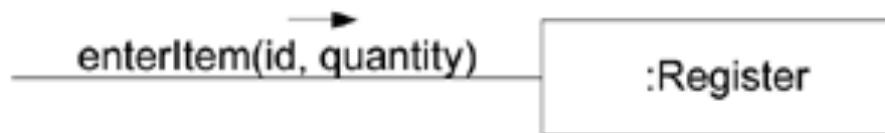
GRASP: Controller

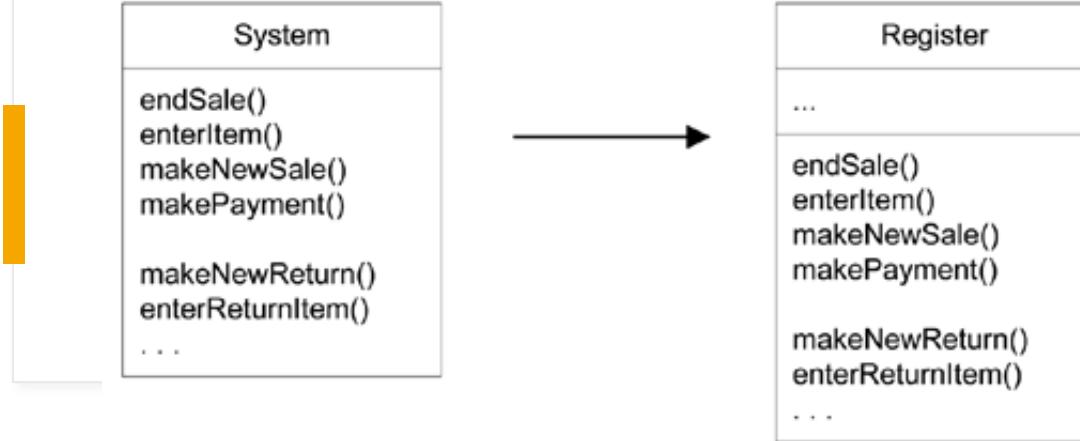
Voorbeeld



GRASP: Controller

- Voorbeeld: mogelijkheden patroon
 - Optie 1
 - Register, POSSystem klasse
 - Optie 2
 - ProcessSaleHandler, ProcessSaleSession

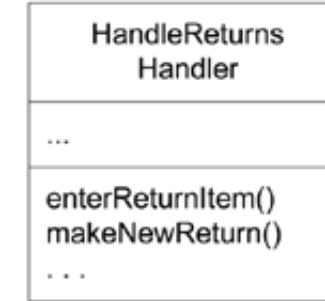
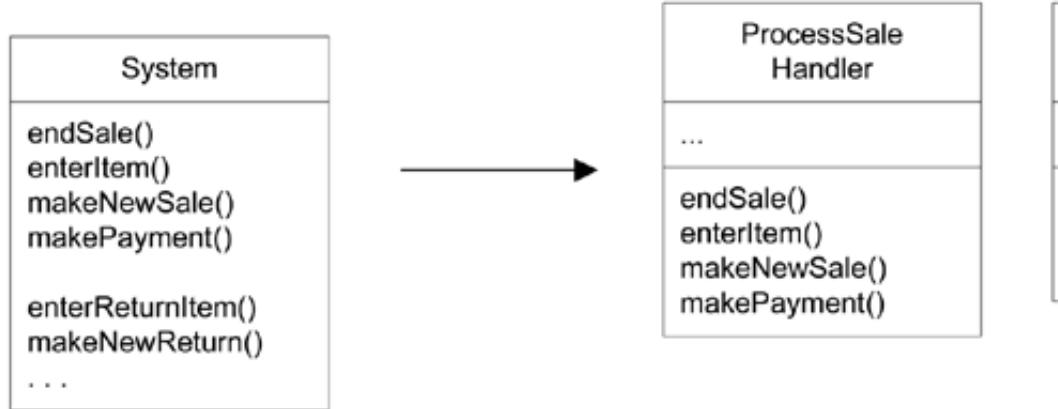




system operations
discovered during system
behavior analysis

allocation of system
operations during design,
using one facade controller

Optie 1



Optie 2

allocation of system
operations during design,
using several use case
controllers

GRASP: Controller

- Optie 2 vooral wanneer optie 1 leidt tot een veel te grote of complexe controller
- Opgelet
 - Niet te veel functionaliteit door controller laten doen
 - Tip: een controller delegert enkel werk naar andere klasse, de controller zelf voert geen of weinig werk uit

GRASP: Controller

- GRASP Controller vs. Model View Controller (MVC) controller
 - GRASP: Controller is deel van domain layer, heeft geen idee van wat soort UI gebruikt wordt (bv. web UI, Swing UI, ...)

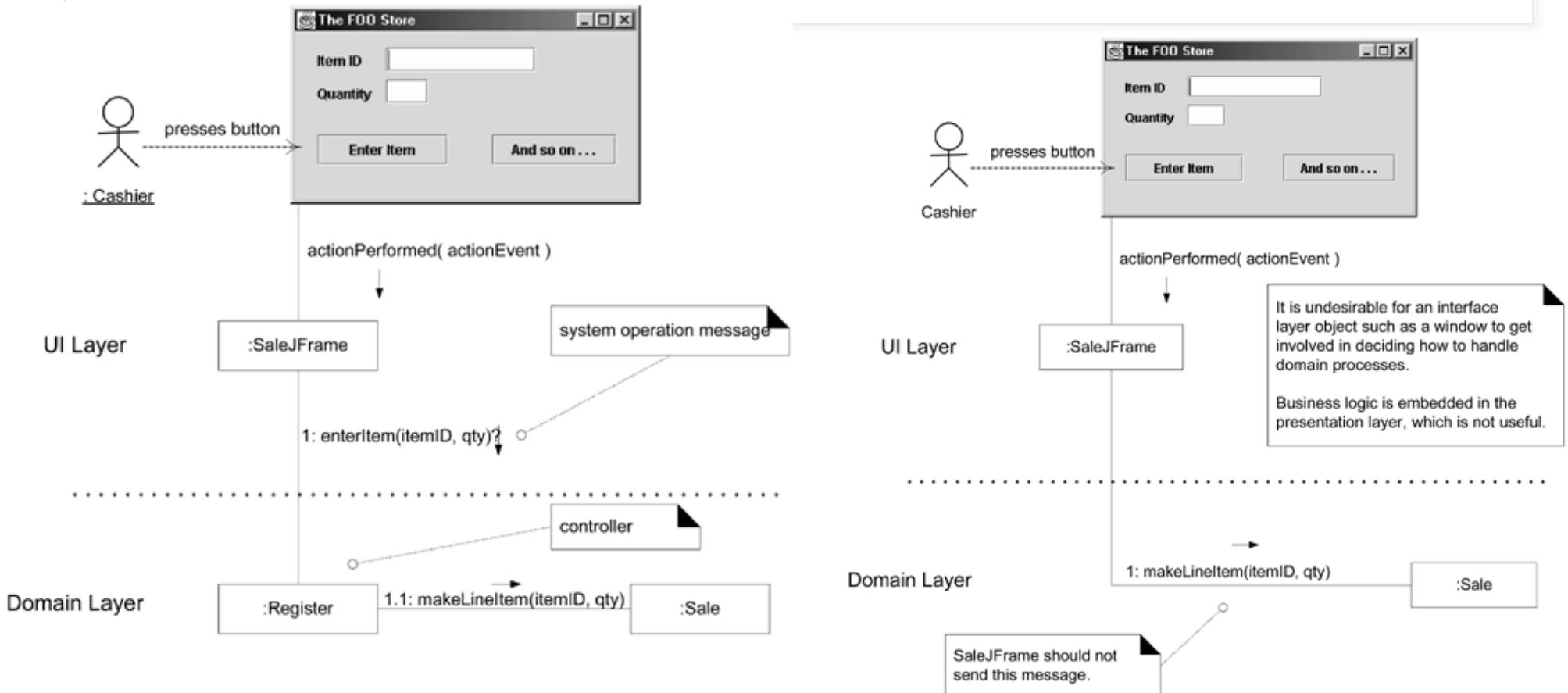
VS

- MVC
 - Controller is deel van UI en beheert UI interactie en page flow

GRASP: Controller

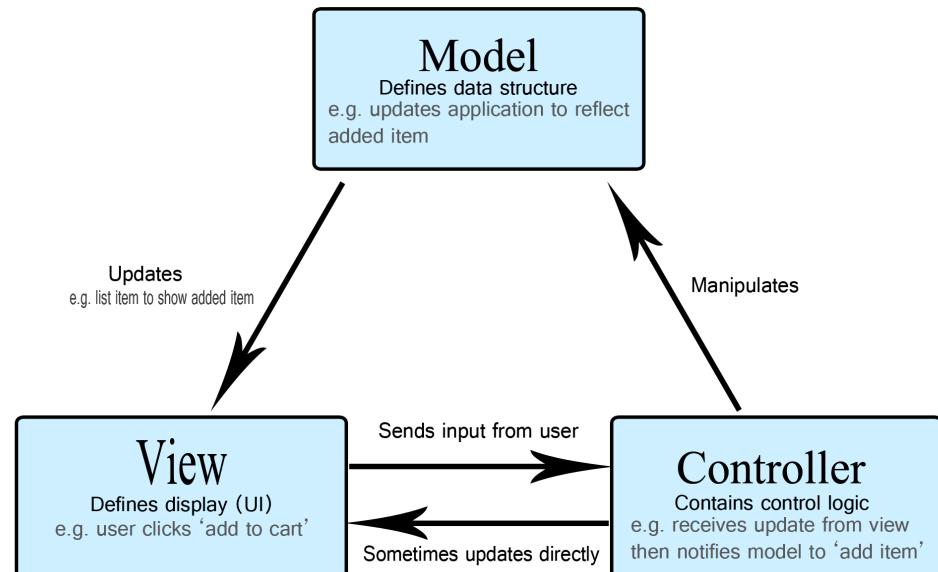
- Voordelen
 - Meer hergebruik mogelijk van klassen
 - Opnieuw nadenken over use cases
 - Bv. pas product toevoegen nadat een nieuwe sale is aangemaakt

GRASP: Controller



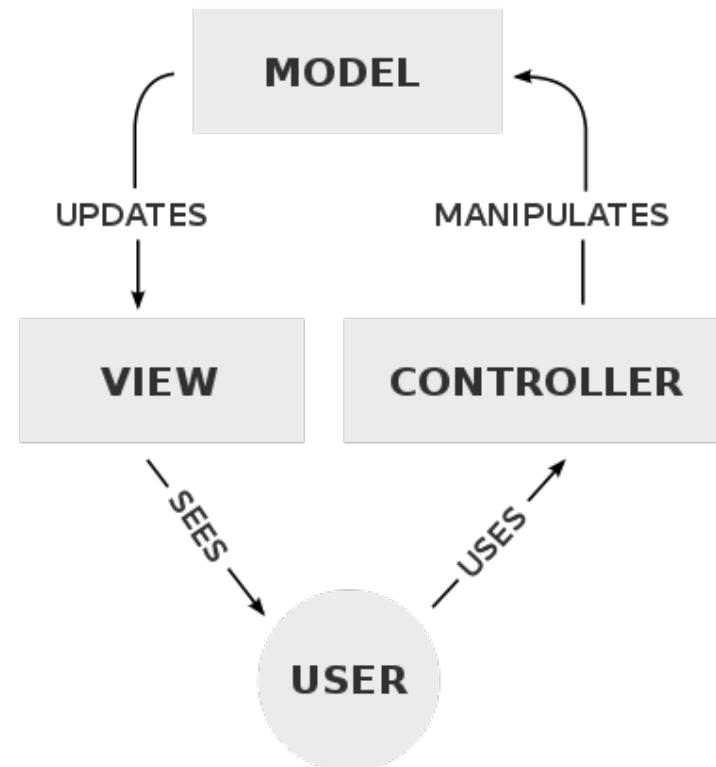
Intermezzo: MVC

- Model View Controller
 - De representatie van informatie scheiden van de gebruikersinteractie van deze informatie
 - Model = application data, business rules, logica, ...
 - View = zichtbare van gegevens (vaak GUI)
 - Controller = verwerker van input naar commando's voor de view of het model



Intermezzo: MVC

- Werking MVC-architectuur
 - **Model**: gegevensinhoud die door het programma wordt bewerkt
 - **Views**: "toonmanieren" van het model
 - **Controller(s)**: code die bepaalt hoe acties van de gebruikers invloed hebben op het model



GRASP: Indirection

- Hoe kunnen we koppeling tussen 2 of meer elementen vermijden?
- Indirection introduceert een tussenliggende element dat de communicatie voorziet tussen de andere elementen. Op deze manier zijn deze elementen niet meer rechtstreeks met elkaar verbonden.

GRASP: Indirection

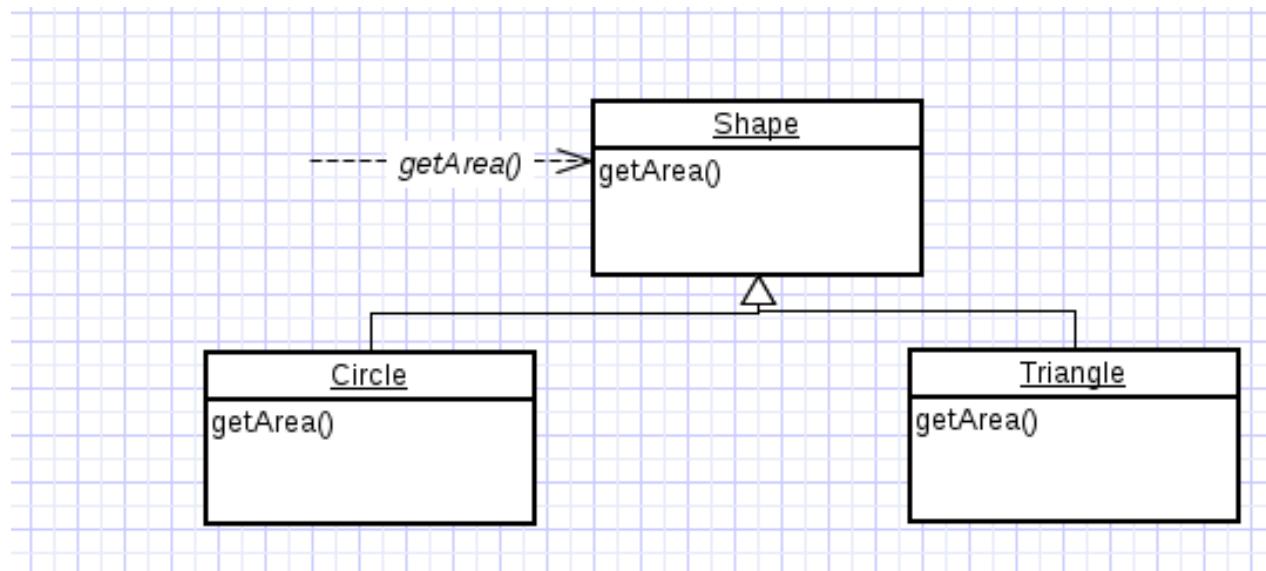
- Het Indirection pattern ondersteunt lage koppeling en hergebruikt potentieel tussen twee elementen door de verantwoordelijkheid van interactie tussen deze elementen toe te wijzen aan een tussenliggend object.
- Een voorbeeld hiervan is de introductie van een controllercomponent voor bemiddeling tussen data (model) en de representatie (view) ervan in het model-view-controller-patroon. Dit zorgt ervoor dat de onderlinge koppeling laag blijft.

GRASP: Polymorphism

- Volgens het polymorfismeprincipe wordt de verantwoordelijkheid voor het definiëren van de variatie van gedrag op basis van type toegewezen aan het type waarvoor deze variatie plaatsvindt. Dit wordt bereikt met behulp van polymorfe bewerkingen.
- De gebruiker van het type moet polymorfe bewerkingen gebruiken in plaats van gebruik te maken van verschillende types.

GRASP: Polymorphism

- Wanneer verwante alternatieven of gedragingen per type (klasse) verschillen, wijs dan de verantwoordelijkheid voor het gedrag - met behulp van polymorfe bewerkingen - toe aan de typen waarvoor het gedrag varieert.



GRASP: Protected variations

- Het principe van beschermde variaties is gerelateerd aan dat van low coupling, omdat het helpt de impact van de wijzigingen van de code van een deel A op een ander deel B te verminderen.
- De code van deel B is beschermd tegen de variaties van de code van deel A, vandaar de naam van het patroon.

GRASP: Protected variations

- Hoe bereik je zo'n bescherming?
- Door de verantwoordelijkheden te organiseren rond stabiele interfaces.
- Dit is met name relevant voor code die vaak verandert. Het introduceren van een interface tussen dit onstabiele deel van de code en de rest van de codebase helpt de gevende effecten van die frequente wijzigingen te beperken.

GRASP: Pure fabrication

- Het is normaal om in onze code objecten weer te geven die de realiteit in kaart brengen van het domein dat we proberen te modelleren.
- Maar soms heb je een verantwoordelijkheid om toe te wijzen, die niet goed lijkt te passen in een domeinklasse. Volgens het principe van high cohesion, moet je een verantwoordelijkheid niet forceren in een klas die al iets anders doet.
- Dat is wanneer het principe van Pure fabricage in het spel komt: maak een klasse die niet is toegewezen aan een domeinobject en laat deze deze nieuwe verantwoordelijkheid op een samenhangende manier bereiken.

GRASP: Pure fabrication

- Een LogInterface die verantwoordelijk is voor het loggen van informatie is een goed voorbeeld voor Pure Fabrication.



Software Design & Quality Assurance

Johan van den Broek



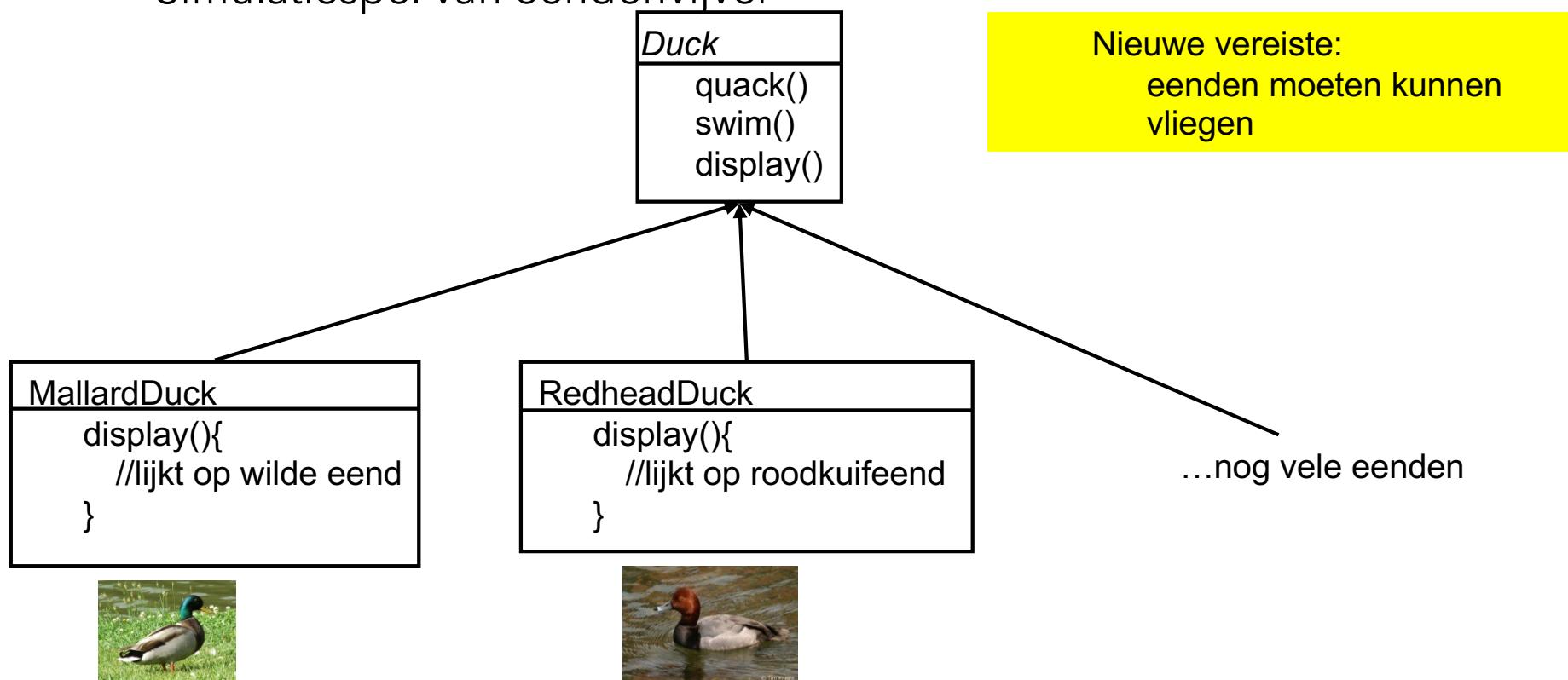
Design Patterns



Strategy

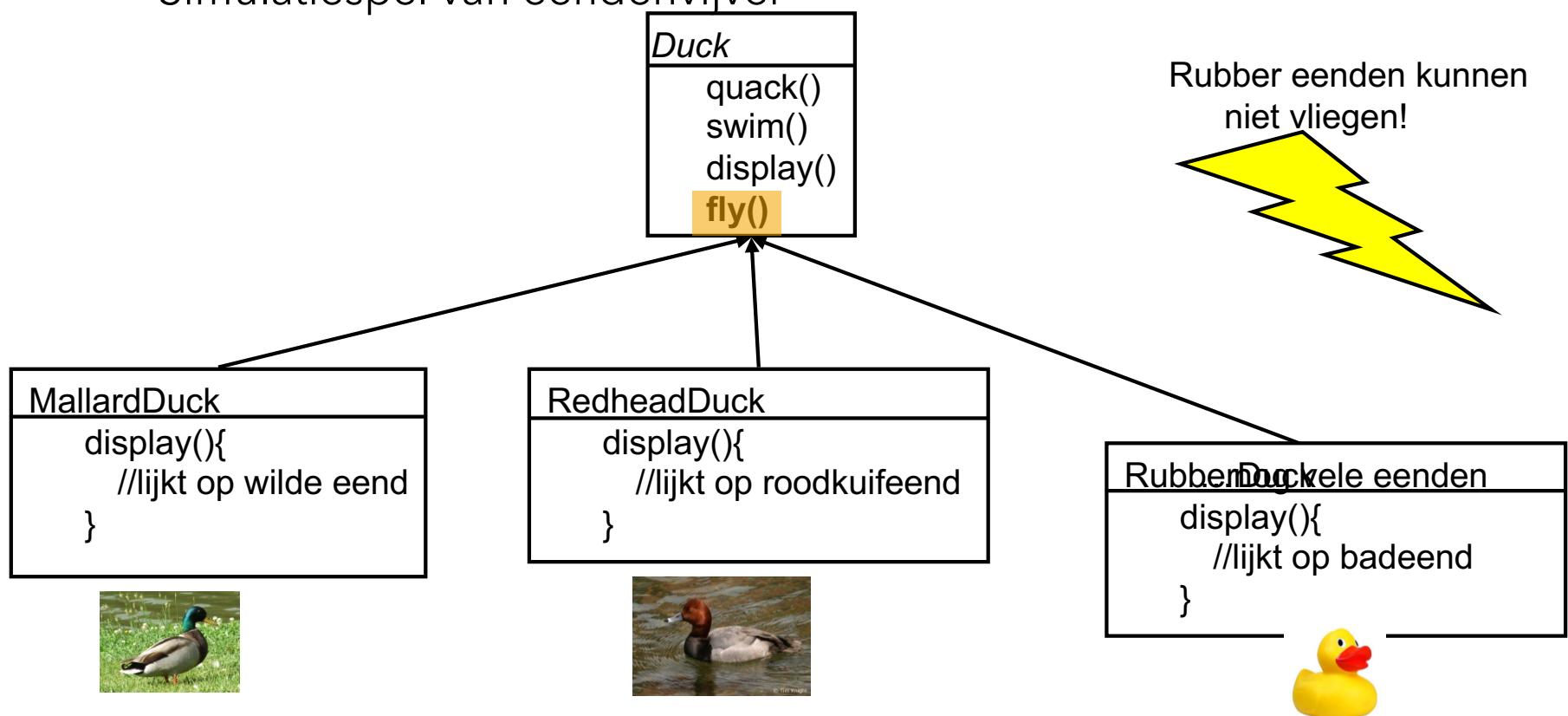
Strategy

- De SimUDuck-toepassing...
 - Simulatiespel van eendenvijver



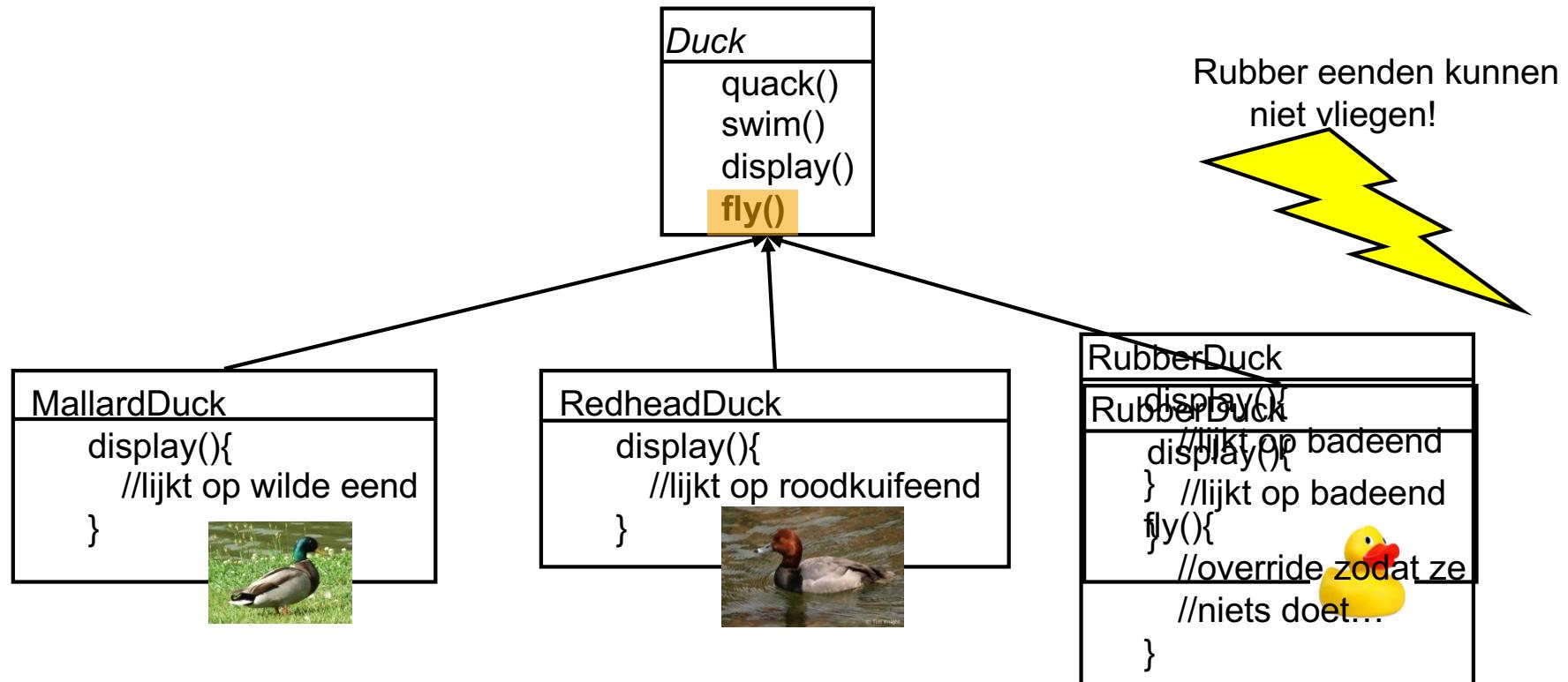
Strategy

- De SimUDuck-toepassing...
 - Simulatiespel van eendenvijver



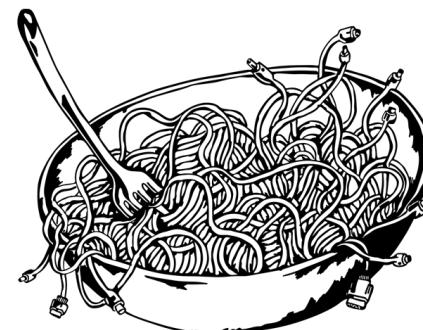
Strategy

- Idee: overerving voor **hergebruik**
- Probleem: moeilijk **hergebruik!**



Strategy

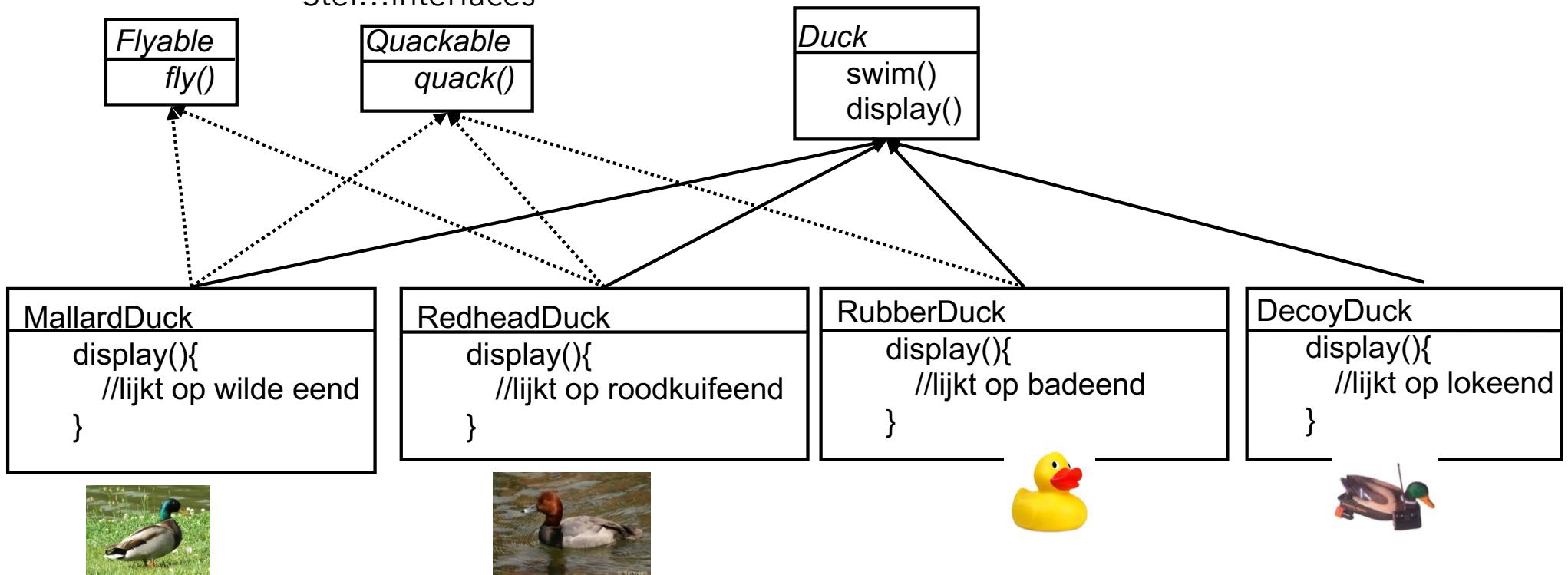
- De SimUDuck-toepassing...
 - Stel: DecoyDuck komt erbij (lokeend)
 - Geen quack()
 - Geen fly()
 - Stel...beide overriden?



SPAGHETTI
CODE

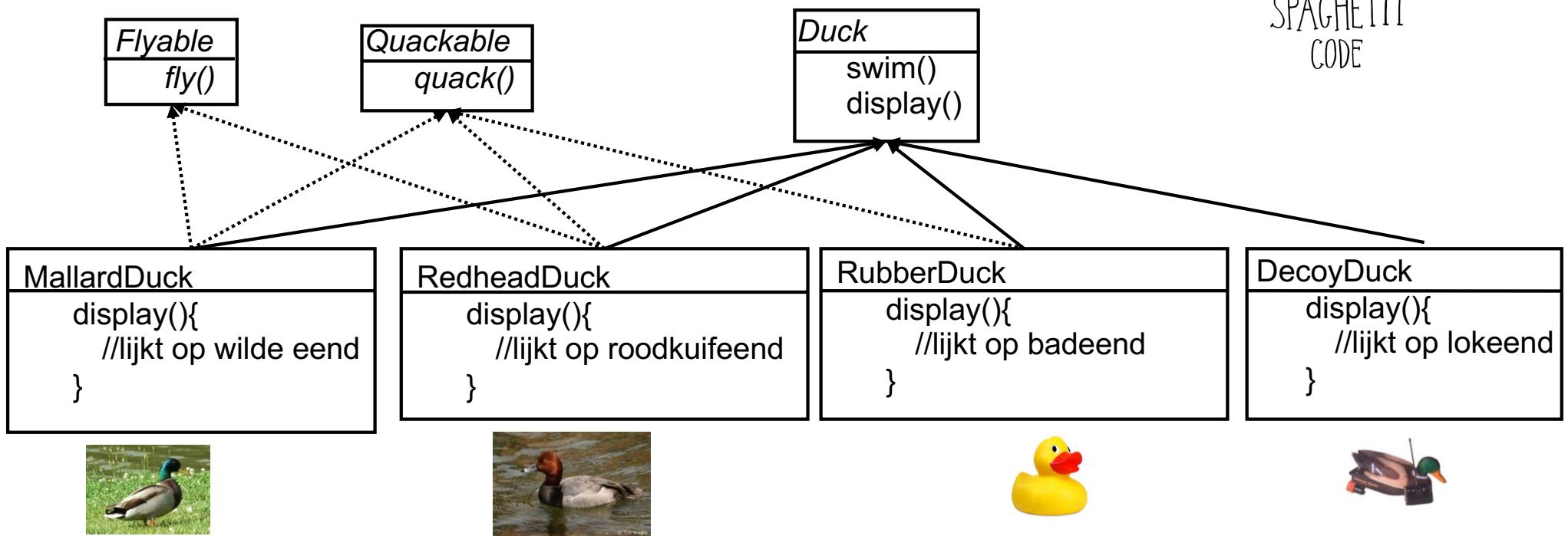
Strategy

- De SimUDuck-toepassing...
 - Stel: DecoyDuck komt erbij (lokeend)
 - Stel...interfaces



Strategy

- Stel...interfaces...nachtmerrie!
 - Duplicatie van code in concrete Duck klassen
 - Wat bij wijziging in bepaalde implementatie?



Strategy

- De SimUDuck-toepassing...
 - De enige constante bij softwareontwikkeling
 - **Verandering** van het systeem
 - Ontwerpprincipe
 - Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven

Strategy

- De SimUDuck-toepassing...
 - Delen in SimUDuck die verschillen
 - fly()
 - quack()
 - Hoe zullen we dit implementeren?
 - Flexibiliteit is belangrijk
 - We willen gedrag aan instanties van Duck toekennen
 - Misschien ook gedrag dynamisch aanpasbaar maken?
 - Bv. setQuack(...)

Strategy

- De SimUDuck-toepassing...
 - Ontwerpprincipe
 - Programmeer naar een interface, niet naar een implementatie
 - Voorbeeld programmeren naar implementatie
 - Dog h = new Dog();
h.bark();
 - Voorbeeld programmeren naar interface
 - Animal animal = new Dog();
animal.makeSound();

Strategy

- De SimUDuck-toepassing...
 - Oplossing...
 - Ontwerprincipe
 - Geef aan compositie de voorkeur boven overerving
- De SimUDuck-toepassing...
 - Uitwisselbare behaviors
 - quack()
 - fly()

Strategy

- Het **strategy** pattern definieert een familie van algoritmen, isoleert ze en maakt ze uitwisselbaar. Strategy maak het mogelijk om het algoritme los van de client die deze gebruikt, te veranderen.

Strategy: applicability

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
 - The Strategy pattern lets you indirectly alter the object's behavior at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.
- Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.
 - The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

Strategy: applicability

- Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.
 - The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.
- Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.
 - The Strategy pattern lets you do away with such a conditional by extracting all algorithms into separate classes, all of which implement the same interface. The original object delegates execution to one of these objects, instead of implementing all variants of the algorithm.

Strategy: advantages



You can swap algorithms used inside an object at runtime.



You can isolate the implementation details of an algorithm from the code that uses it.



You can replace inheritance with composition.



Open/Closed Principle. You can introduce new strategies without having to change the context.

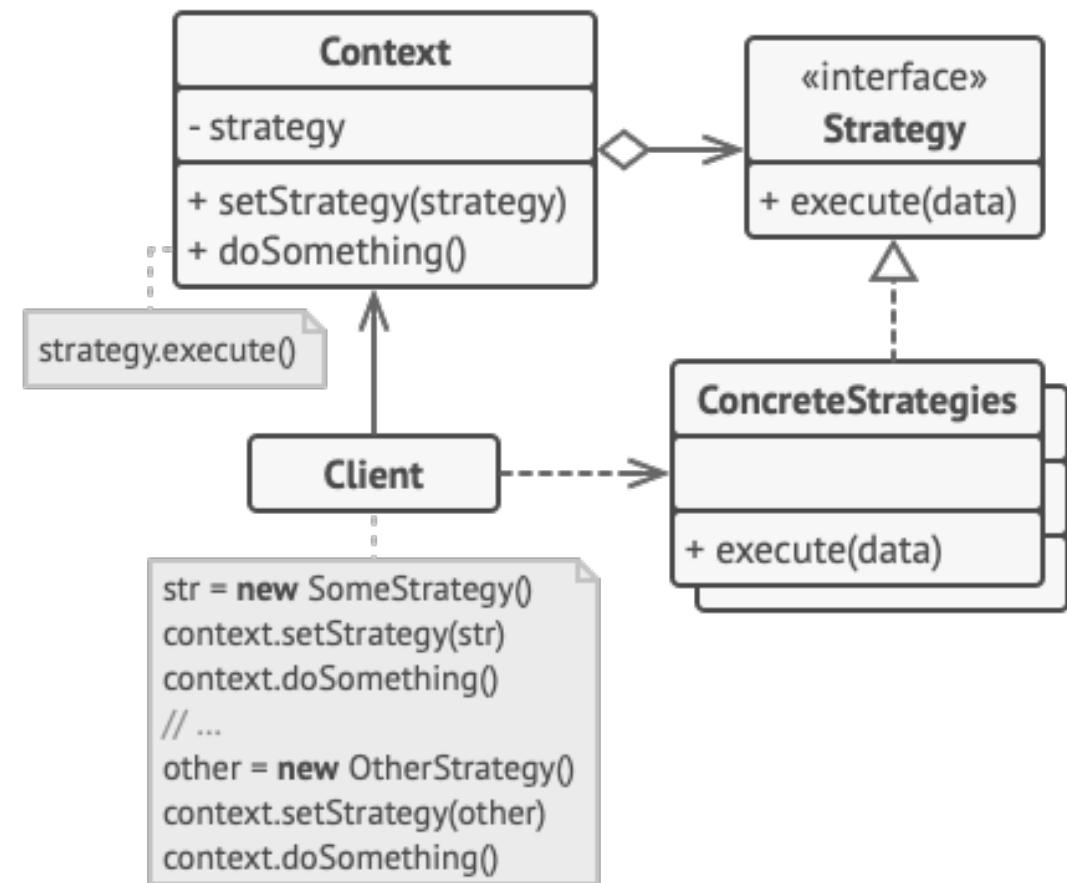
Definition of the Open/Closed Principle

- “**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.**”
- The Polymorphic Open/Closed Principle uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.

References

- <https://refactoring.guru/design-patterns/strategy>

Strategy

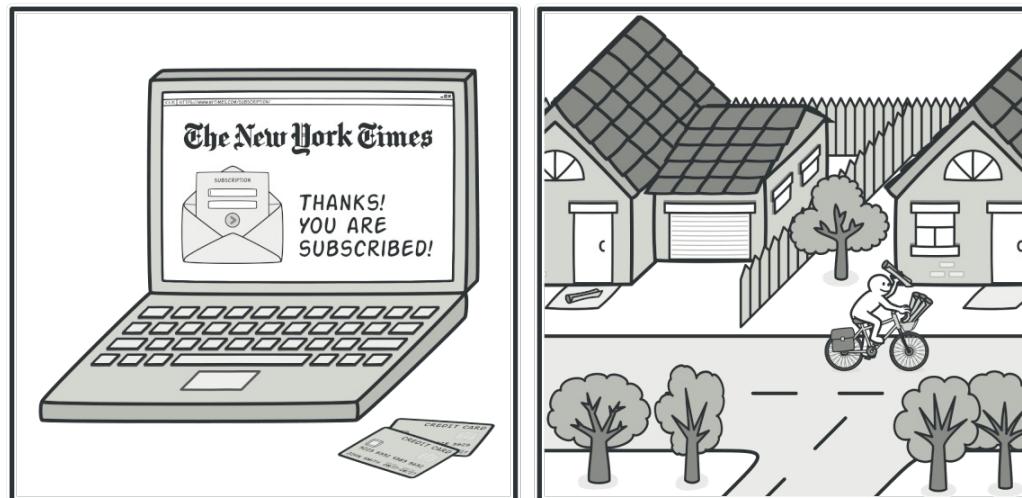




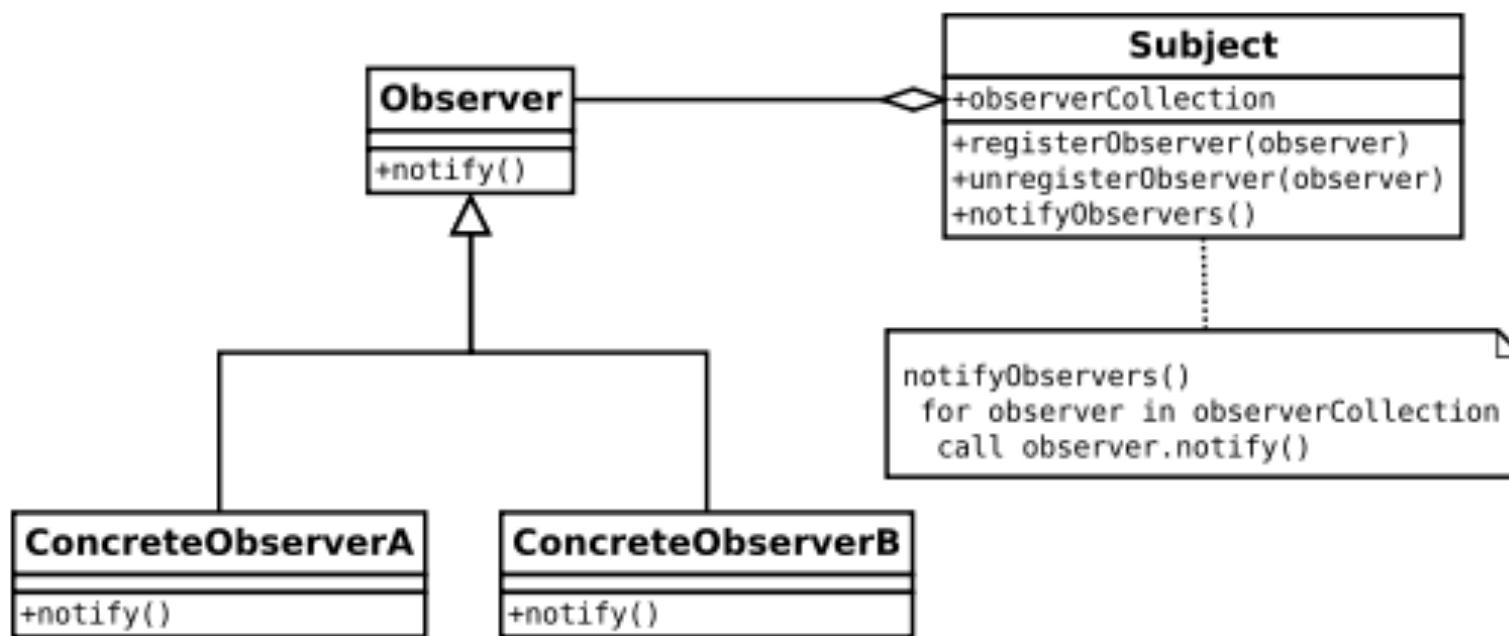
Observer

Observer

- The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.



Observer





Observer

The Observer pattern provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

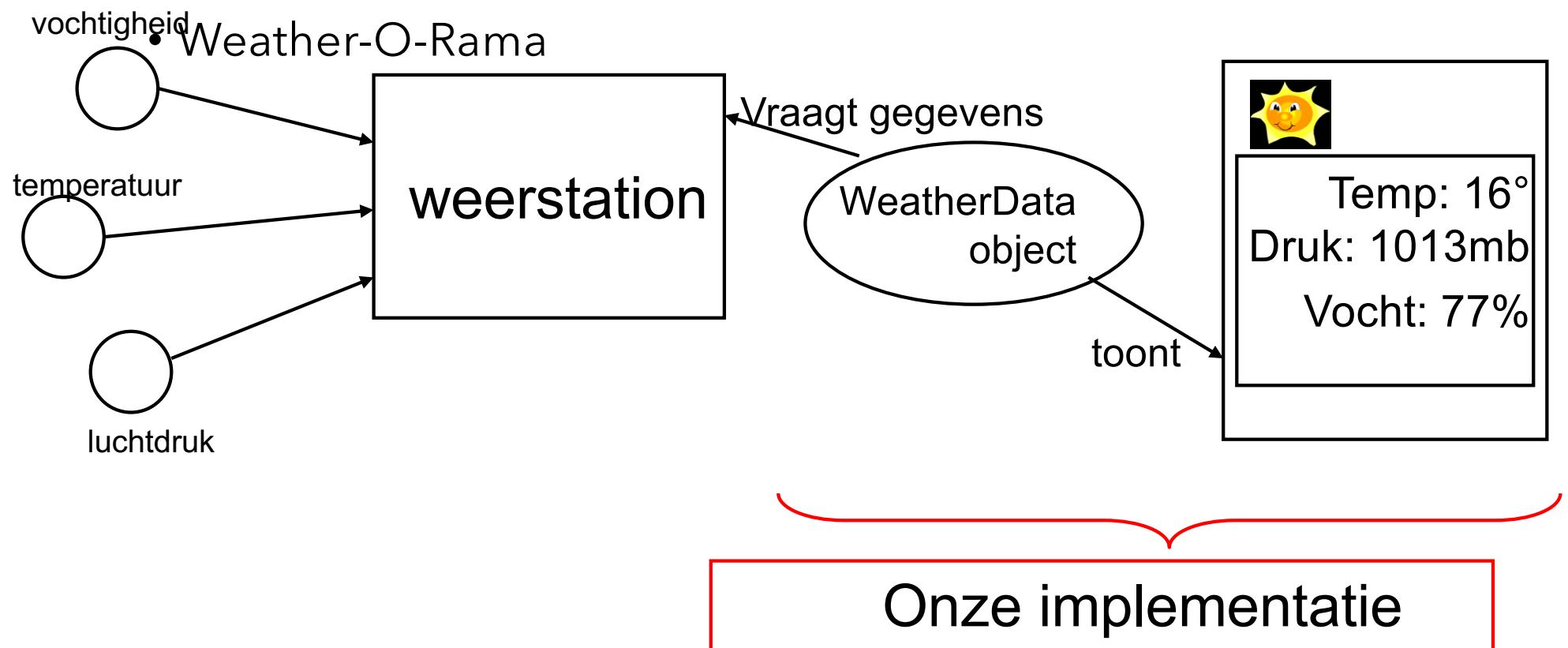
Observer

- Observer pattern provides this loose coupling as:
 - Subject only knows that observer implement Observer interface.
Nothing more.
 - There is no need to modify Subject to add or remove observers.
- We can reuse subject and observer classes independently of each other.

Observer example

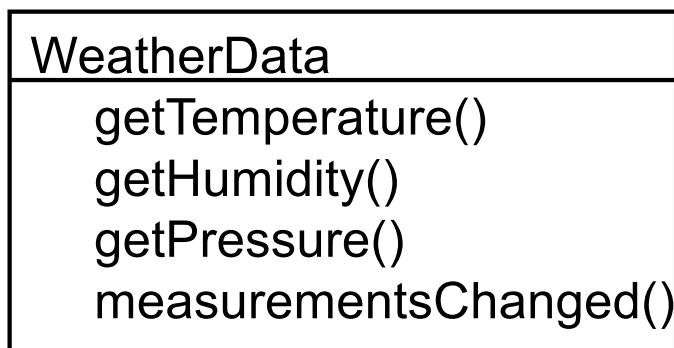
- Weather-O-Rama
 - Internet-based weerstation
 - WeatherData object geeft temperatuur, vochtigheid en luchtdruk
 - Applicatie toont deze 3 parameters
 - Weerstation moet uitgebreid kunnen worden: API zodat andere ontwikkelaars hun eigen weerschermen kunnen inpluggen
 - Voorlopig 3 schermen ontwikkelen
 - Actuele weergesteldheid
 - Statistieken scherm
 - Verwachtingen scherm

Observer



Observer

- Weather-O-Rama
 - Levering van WeatherData-object



measurementsChanged() wordt
automatisch opgeroepen
wanneer de data wijzigt. Hierin
kunnen wij onze code schrijven...

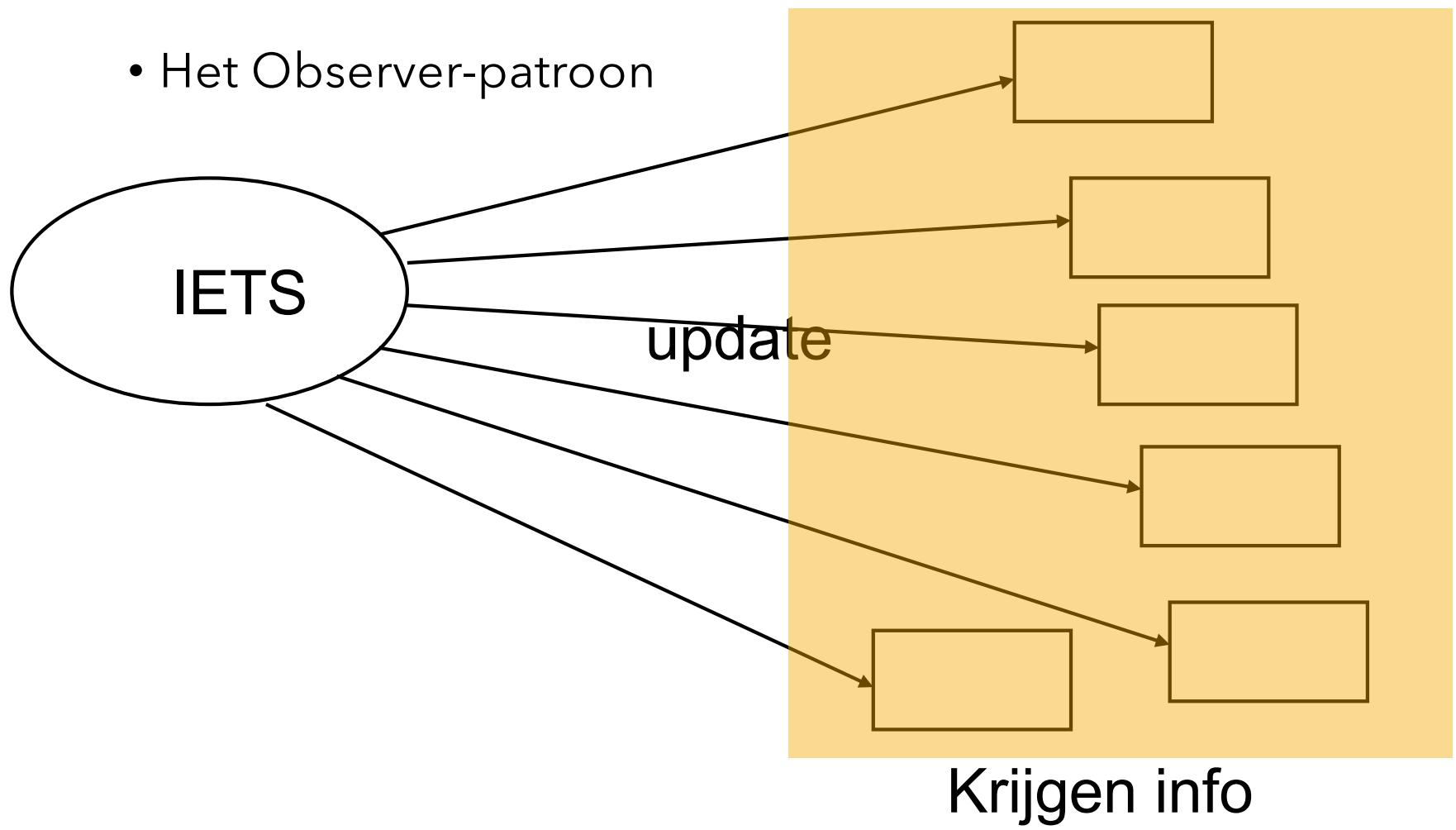
Observer - Weather-O-Rama

```
Public class WeatherData{
    //declaratie instantievariabelen
    public void measurementsChanged(){
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }
    = Naar concrete implementatie      Gezamenlijke “interface”
    coderen...}
```

Observer

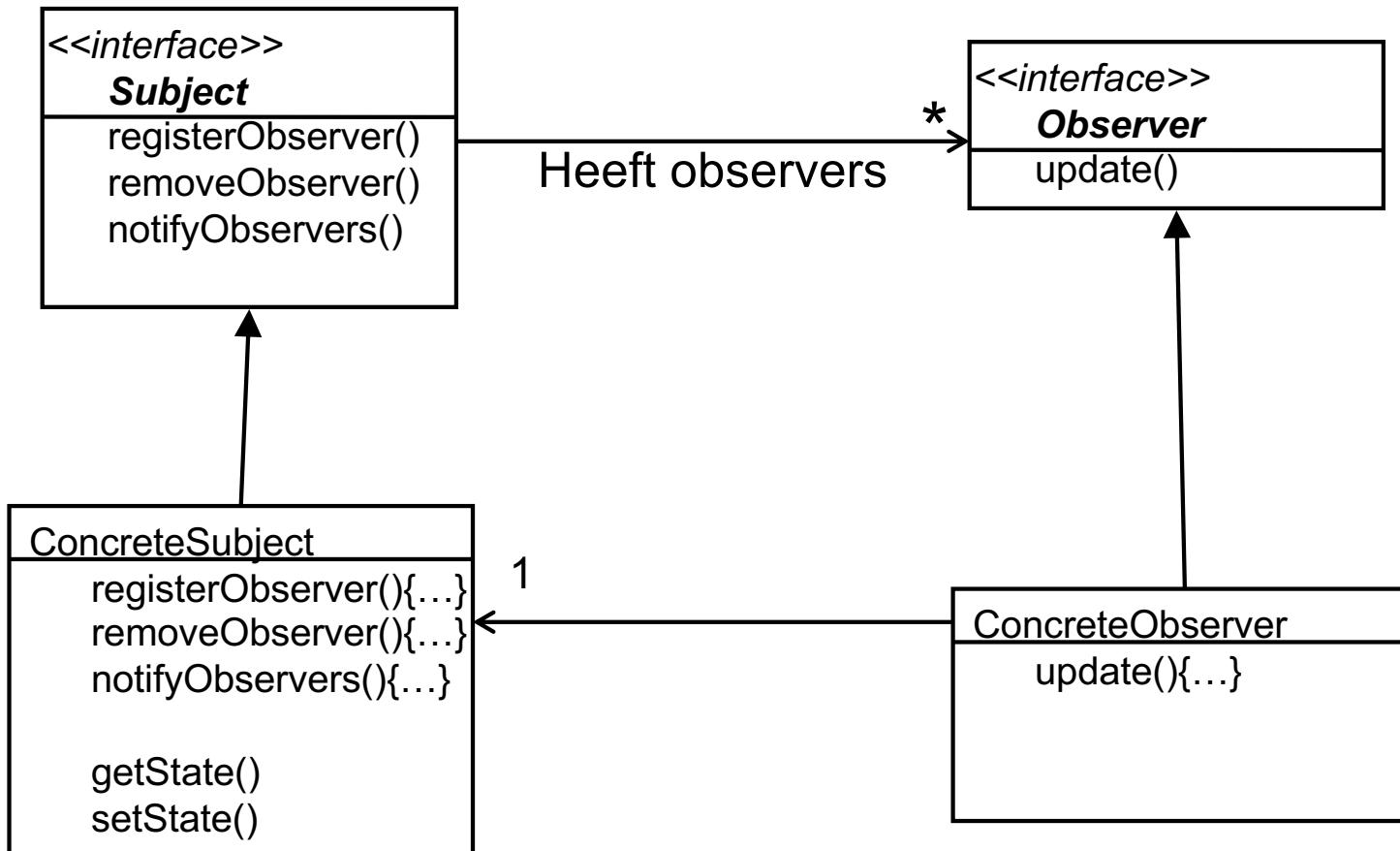
- Het Observer-patroon



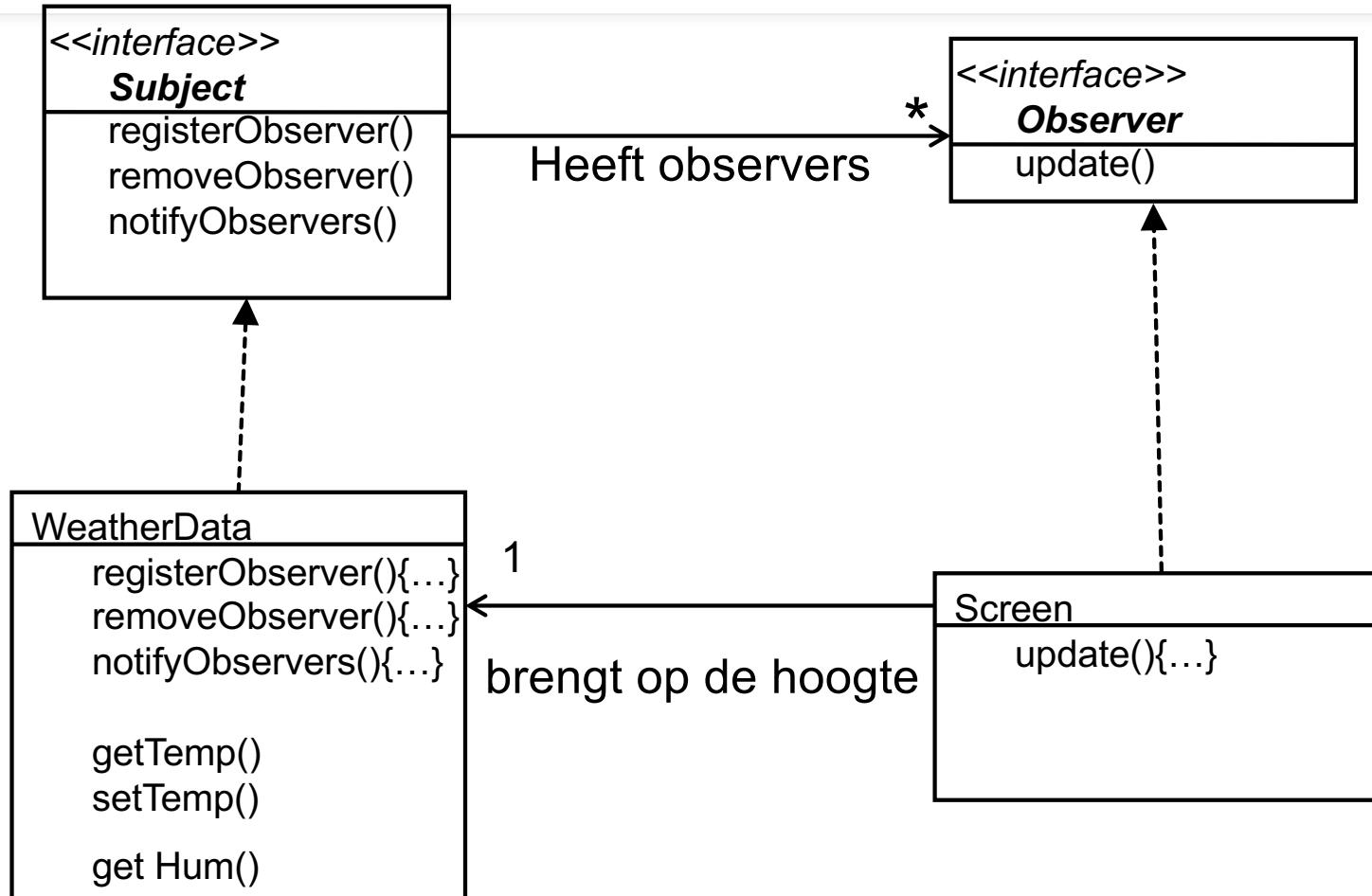
Observer

- Het Observer-patroon
 - Idee: object kan zichzelf registreren
 - ~ inschrijven op nieuwsbrief, krantenabonnement...
 - Het **Observer** Pattern definieert een een-op-veel relatie tussen objecten, zodanig dat wanneer de toestand van een object verandert, alle afhankelijke objecten worden bericht en automatisch worden geupdatet.

Observer



Observer



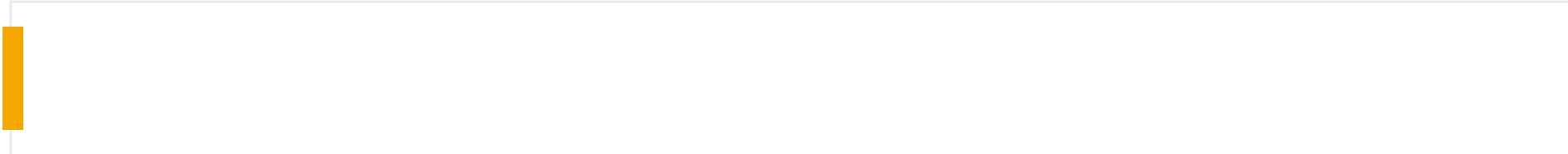
Observer

- Kracht van zwakke koppeling (low coupling)
 - Enige wat subject over de observer weet is dat deze een bepaalde interface implementeert
 - Kan deze dus waarschuwen!
 - We kunnen op ieder moment observers toevoegen
 - Verandering in het subject of observer heeft geen invloed op elkaar
- Ontwerprinciple
 - Streef naar ontwerpen met zwakke koppeling tussen de objecten die samenwerken

Observer

```
public interface Subject() {  
    public void registerObserver (Observer o);  
    public void removeObserver (Observer o);  
    public void notifyObservers ();  
}  
  
public interface Observer() {  
    public void update(float temp, float hum, float pressure);  
}
```

Singleton



Singleton

- Wanneer er maar 1 mag zijn...
 - Singleton
 - 1 object waarmee gewerkt wordt
 - Er mag maar 1 instantie zijn
 - Er is 1 toegangspunt

Singleton

```
Public class Singleton{  
    private static Singleton uniqueInstance;  
  
    //hier komen andere instantievariabelen  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if(uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
  
    //hier komen andere nuttige methoden  
}
```

**Probleem bij
meerdere threads**

Singleton

```
Public class Singleton{  
    private static Singleton uniqueInstance;  
  
    //hier komen andere instantievariabelen  
  
    private Singleton() {}  
  
    public static Synchronized Singleton getInstance() {  
        if(uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
  
    //hier komen andere nuttige methoden  
}
```

Singleton

- Synchronized 😞
 - Performantie tot 100x lager
 - Oplossingen
 - Instantie direct maken
 - ```
Private static Singleton uniqueInstance = new Singleton();
```

        - JVM garandeert dat de instantie gemaakt wordt alvorens een thread toegang krijgt tot de static variabele
        - Nadeel: geen lazy instantiation

# Singleton

- Oplossing Bill Pugh

```
public class Singleton {
 // Private constructor prevents instantiation from other classes
 private Singleton() {}

 /**
 * SingletonHolder is loaded on the first execution of Singleton.getInstance()
 * or the first access to SingletonHolder.INSTANCE, not before. */
 private static class SingletonHolder {
 private static final Singleton INSTANCE = new Singleton();
 }

 public static Singleton getInstance() {
 return SingletonHolder.INSTANCE;
 }
}
```



Thread-safe



Geen synchronize nodig

# Singleton

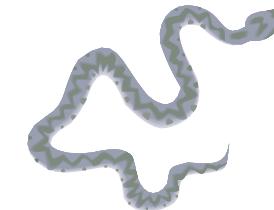
- Het Singleton patroon garandeert dat een klasse slechts één instantie heeft, en biedt een globaal toegangspunt ernaartoe.

# Singleton

- Een addertje onder het gras...
  - Wat als we de clone() methode gebruiken?

```
public static void main(String args[]){
 //Get a singleton
 SingletonObject obj = SingletonObject.getInstance();

 //Buahahaha. Let's clone the object
 SingletonObject clone = (SingletonObject) obj.clone();
}
```



# Singleton



- Een addertje onder het gras...
  - Oplossing: overriden clone() methode

```
public Object clone() throws
CloneNotSupportedException{
 throw new CloneNotSupportedException();
}
```



# Software Design & Quality Assurance

---

Johan van den Broek

# Command



# Command

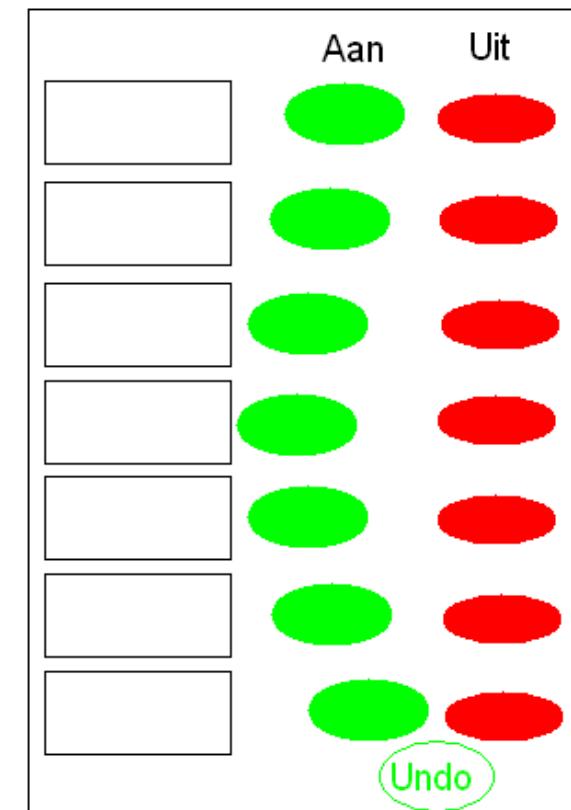
- **Command** is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



# Command

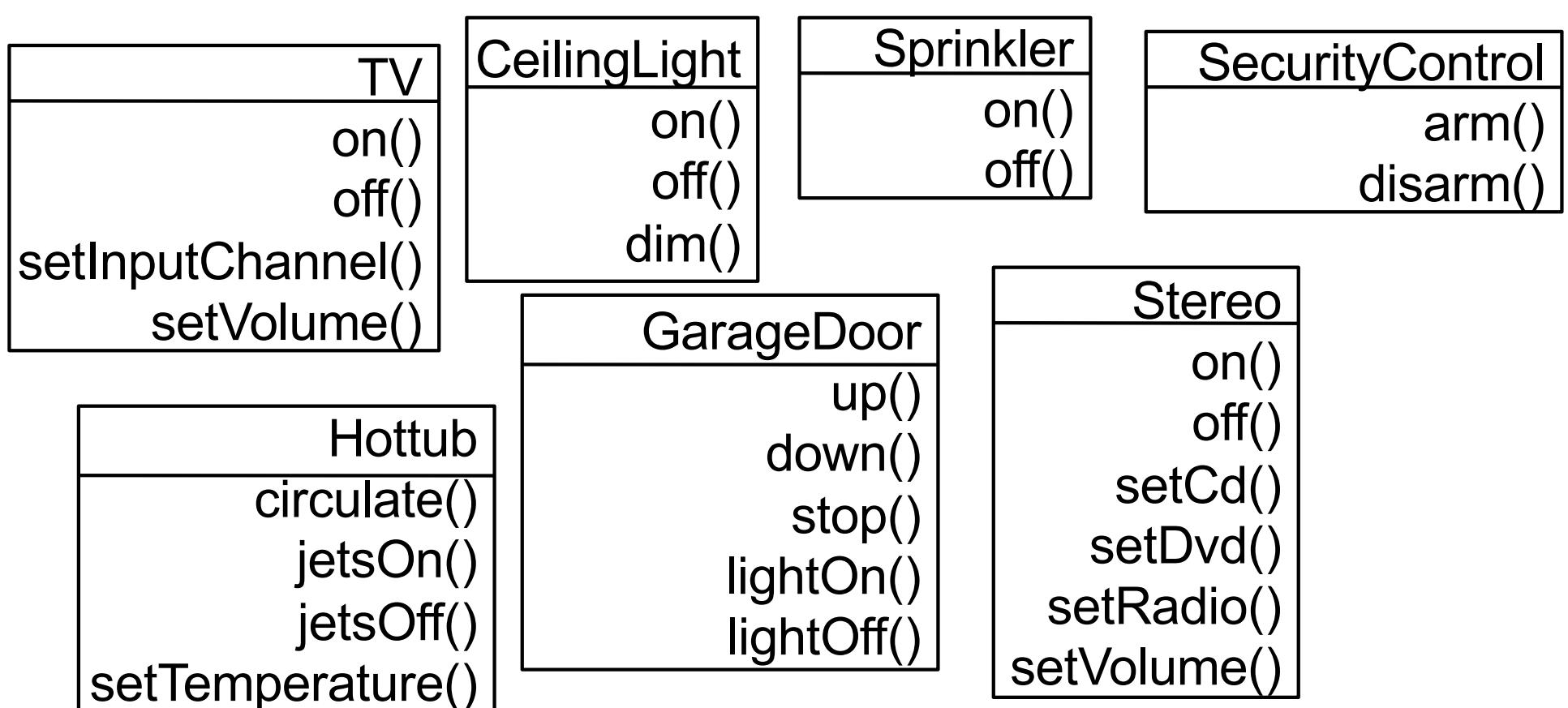
- Ontwikkelen van remote control voor aansturen

- Licht
- Tv
- Tuinverlichting
- Thermostaat
- Stereo
- ...



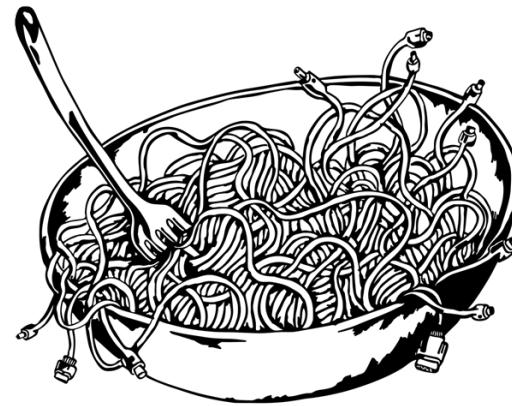
# Command

- Klassen voor aansturing van apparaten



# Command

```
if (slot1=="Light")
 light.on();
else if (slot1=="Hottub")
 hottub.jetsOn();
else if (slot1=="Stereo")
 stereo.on();
else if...
else if...
else...
```

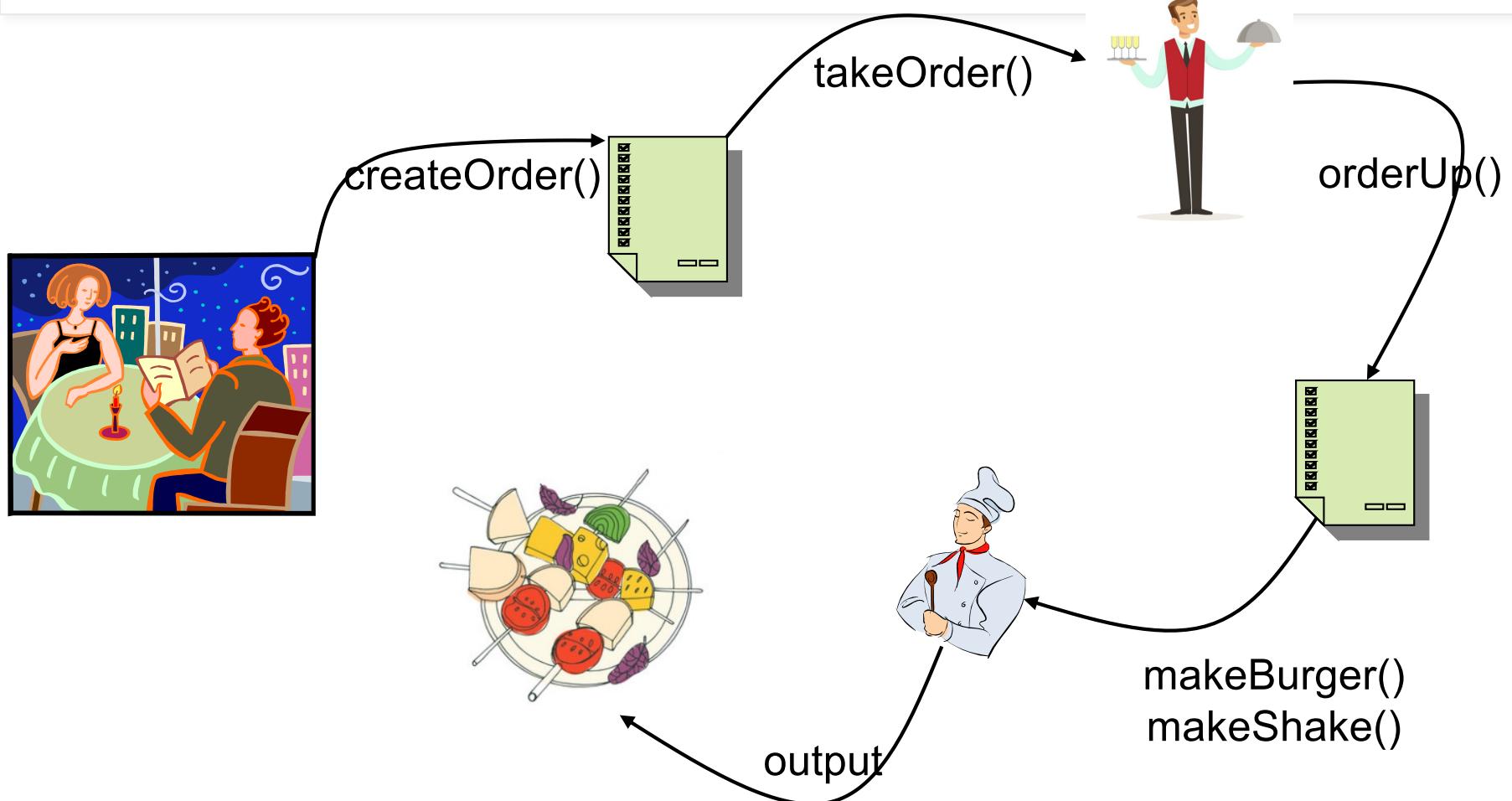


SPAGHETTI  
CODE

# Command

- Command patroon
  - Aanvrager van een actie loskoppelen van het object dat de actie daadwerkelijk uitvoert
  - Introduceren van “Command objecten”
    - Deze objecten schermen een verzoek om iets te doen af
  - Idee van ontkoppeling!

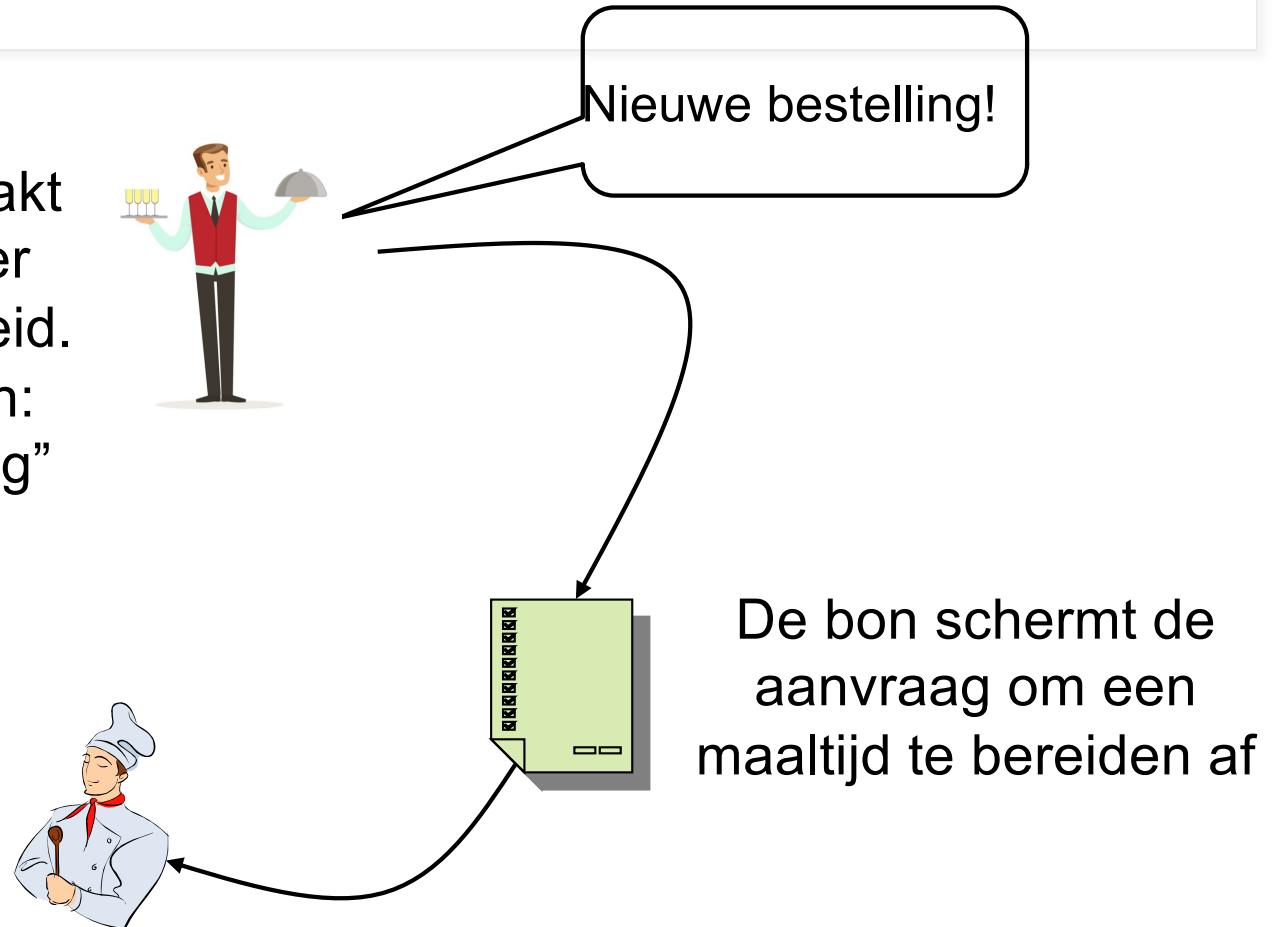
# Command - ontkoppeling



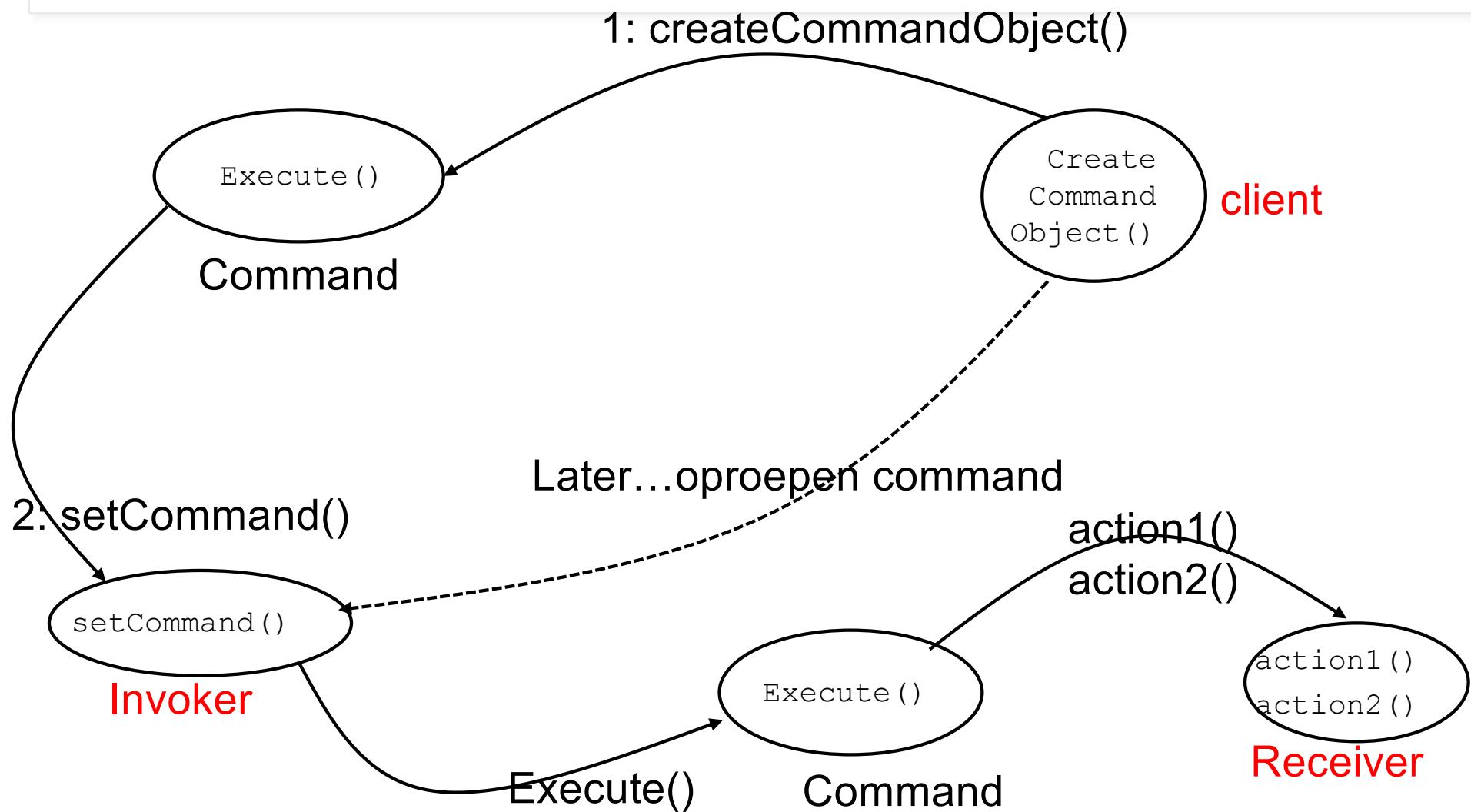
# Command

Voor de ober maakt  
het niet uit wat er  
moet worden bereid.  
Hij roept gewoon:  
“nieuwe bestelling”

De kok weet hoe de  
maaltijden bereid  
moeten worden.



# Command



# Command

- De commandinterface

```
Public interface Command{
 public void execute();
}
```

- Voorbeeld

```
Public class LightOnCommand implements Command{
 Light light;

 public LightOnCommand(Light light) {
 this.light = light;
 }

 public void execute() {
 light.on();
 }
}
```

# Command

- Het commandobject gebruiken

```
Public class SimpleRemoteControl{
 Command slot;

 public SimpleRemoteControl() {}

 public void setCommand(Command command) {
 slot = command;
 }

 public void buttonWasPressed() {
 slot.execute();
 }
}
```

# Command

- Het commandobject gebruiken

```
Public class SimpleRemoteControlTest{
 public static void main(String[] args){

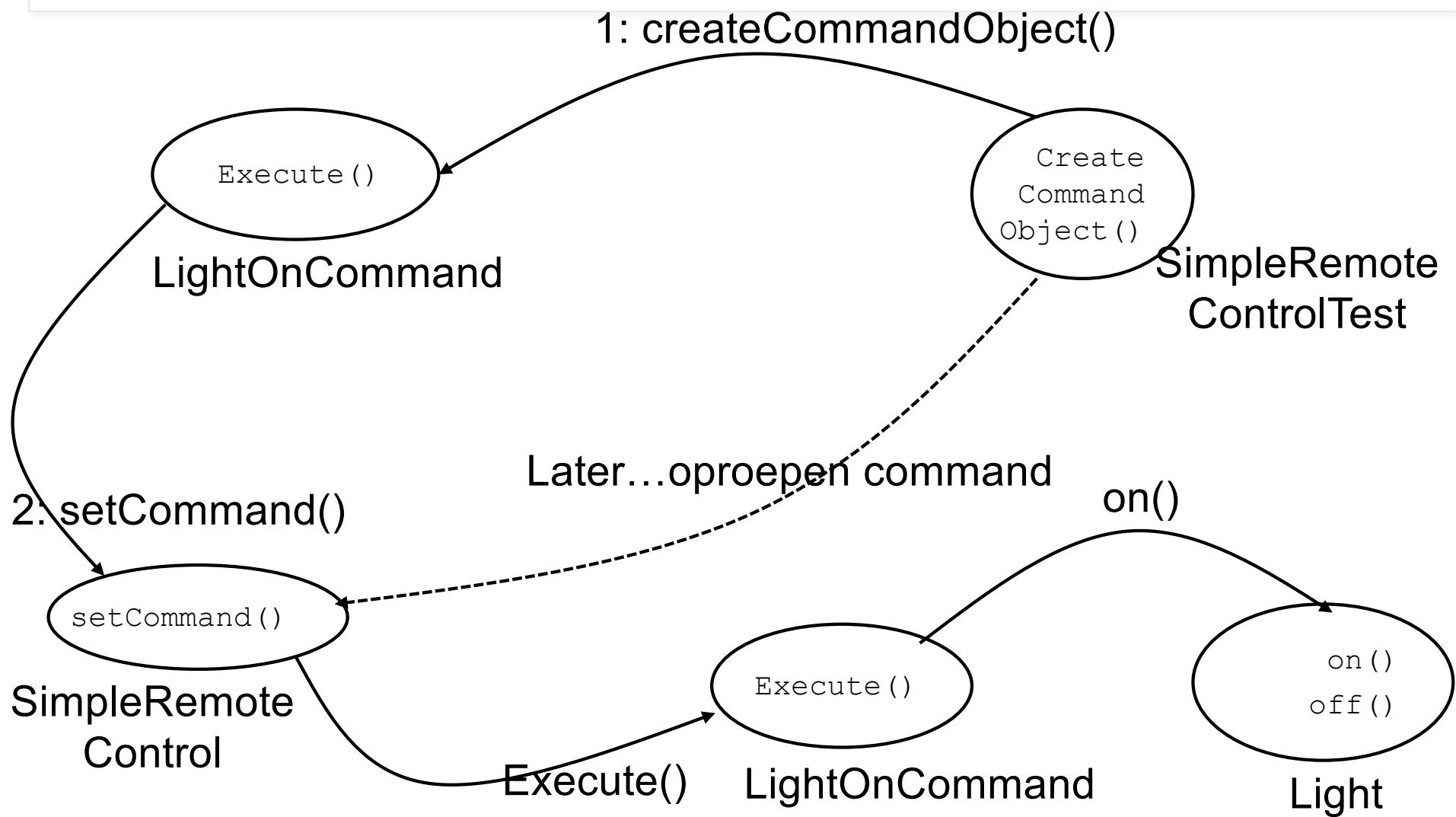
 SimpleRemoteControl remote = new SimpleRemoteControl();
 Light light = new Light();

 LightOnCommand lightOn = new LightOnCommand(light);

 remote.setCommand(lightOn);
 remote.buttonWasPressed();

 }
}
```

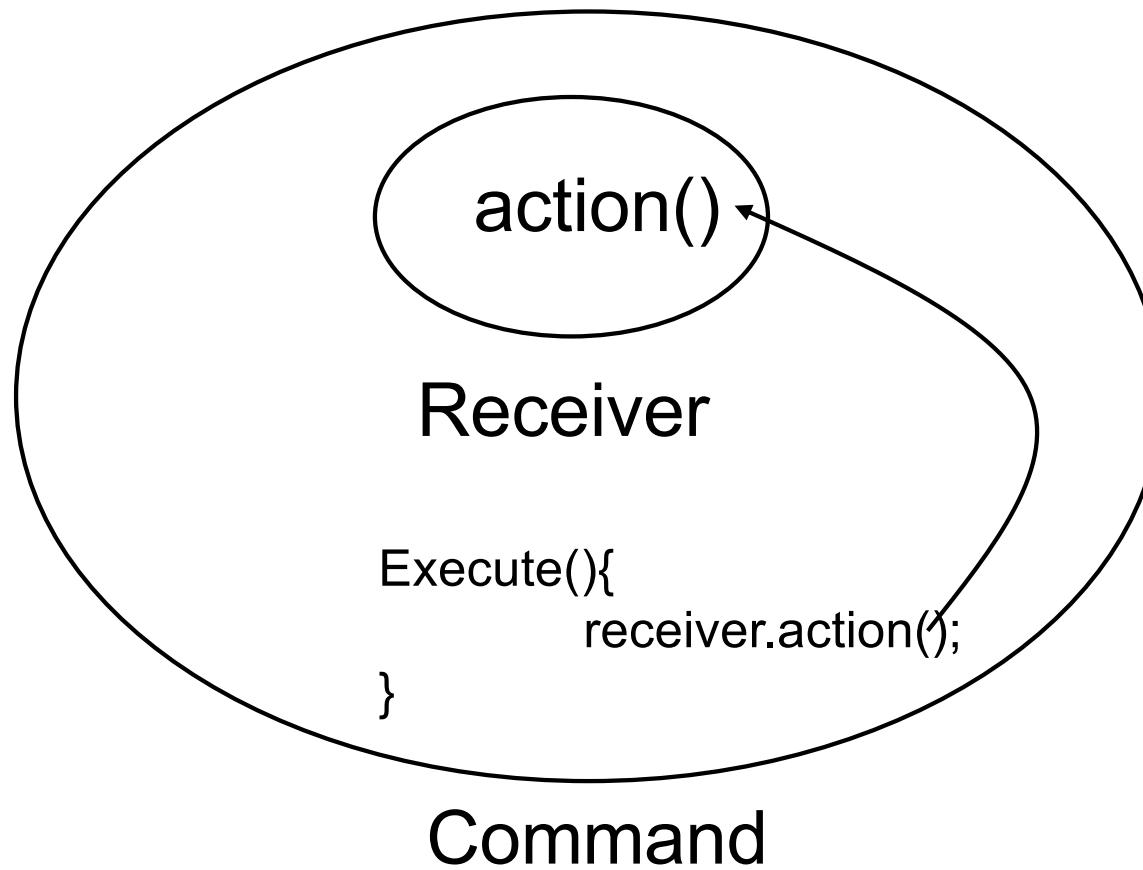
# Command



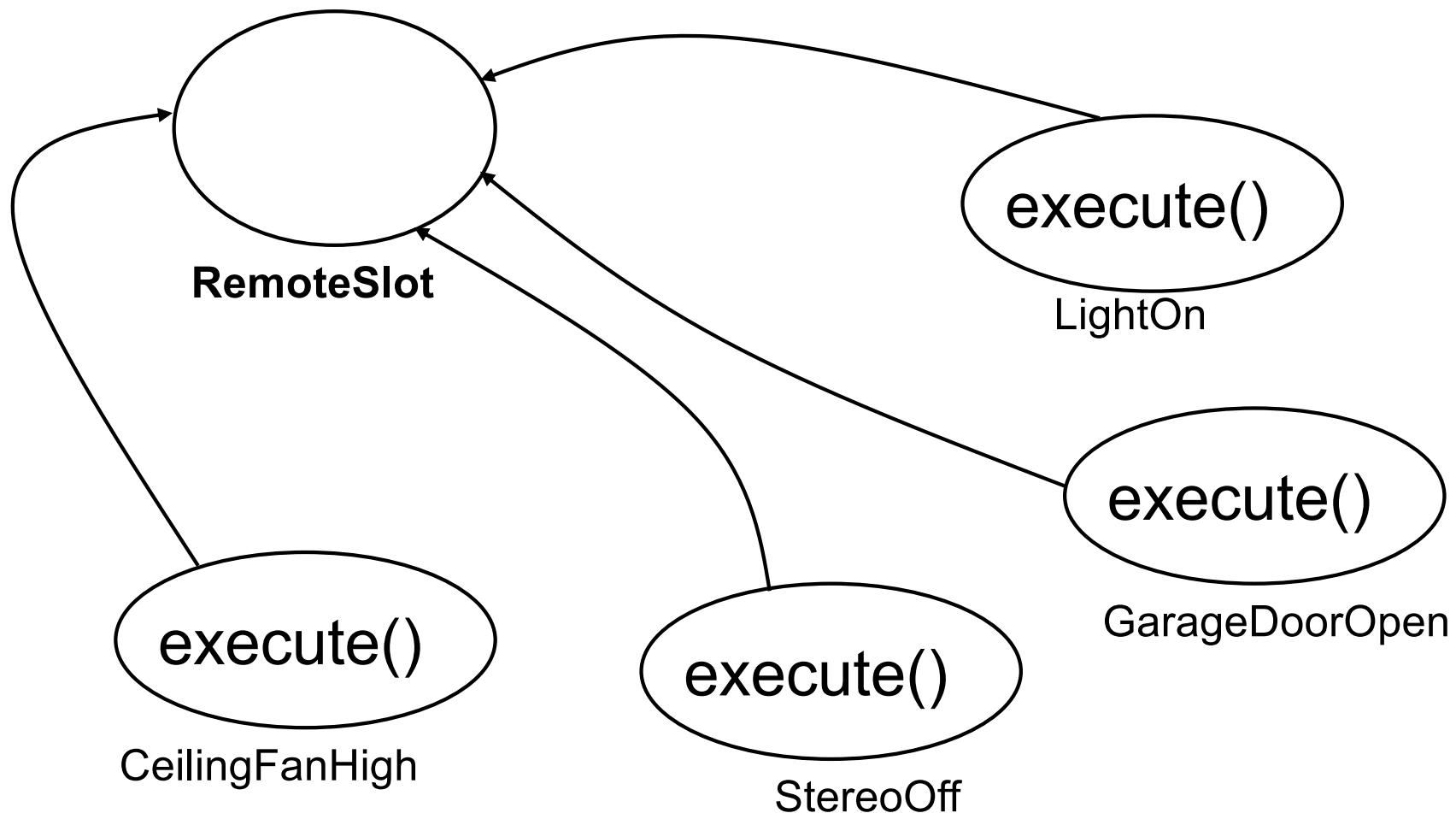
# Command

- **Het command patroon** schermt een aanroep af door middel van een object, waarbij je verschillende aanroepen in verschillende objecten kunt opbergen, in een queue kunnen zetten of op schijf kunnen bewaren; ook undo-operaties kunnen worden ondersteund.

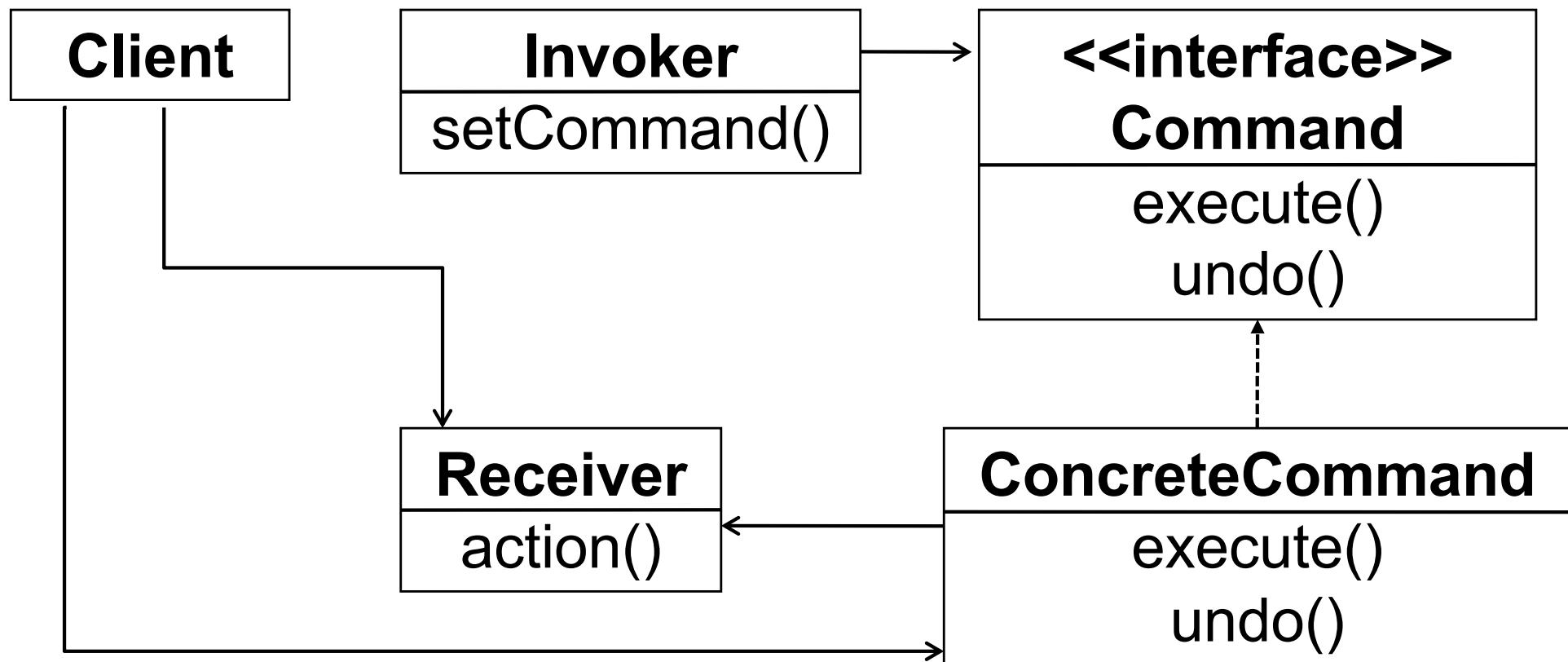
# Command



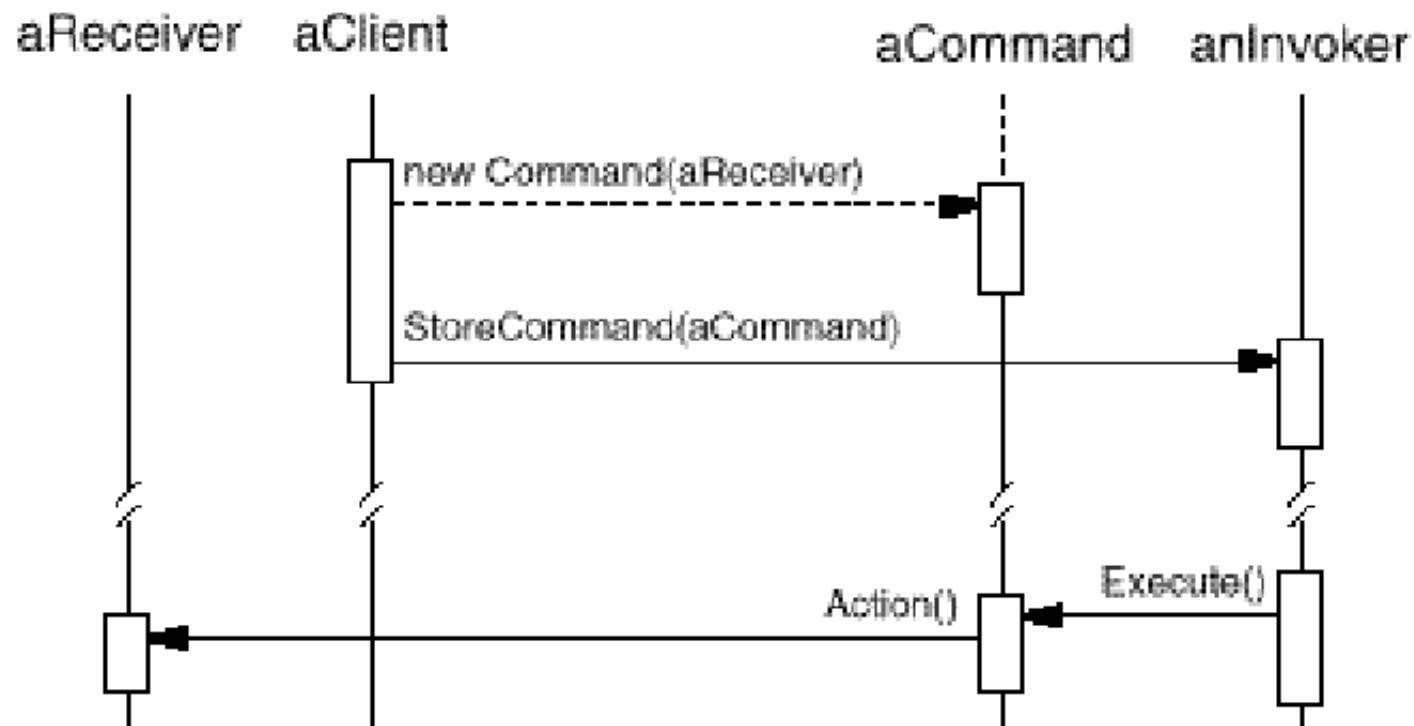
# Command



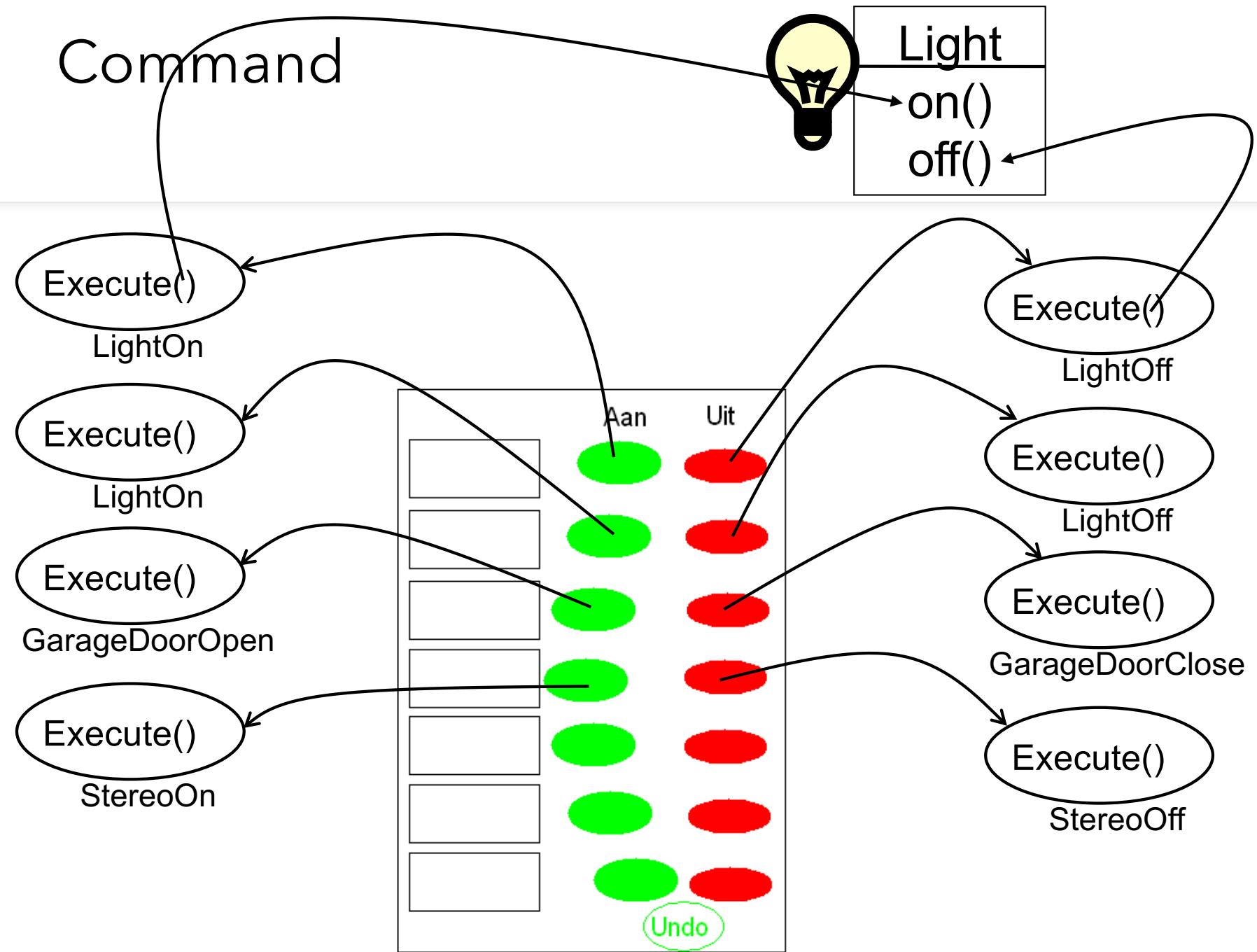
# Command



# Command



# Command



```
public class RemoteControl {
 Command[] onCommands;
 Command[] offCommands;
 public RemoteControl() {
 onCommands = new Command[7];
 offCommands = new Command[7];
 Command noCommand = new NoCommand();
 for (int i = 0; i < 7; i++) {
 onCommands[i] = noCommand;
 offCommands[i] = noCommand;
 }
 }
 public void setCommand(int slot, Command onCommand,
 Command offCommand) {
 onCommands[slot] = onCommand;
 offCommands[slot] = offCommand;
 }
 public void onButtonWasPushed(int slot) {
 onCommands[slot].execute();
 }
 public void offButtonWasPushed(int slot) {
 offCommands[slot].execute();
 }
}
```

```
public class StereoOnWithCDCommand implements Command{
 Stereo stereo;

 public StereoOnWithCDCommand(Stereo stereo) {
 this.stereo = stereo;
 }

 public void execute() {
 stereo.on();
 stereo.setCD();
 stereo.setVolume(11);
 }
}

public class LivingroomLightOffCommand implements Command{
 Light light;

 public LivingroomLightOffCommand(Light light) {
 this.light = light;
 }

 public void execute() {
 light.off();
 }
}
```

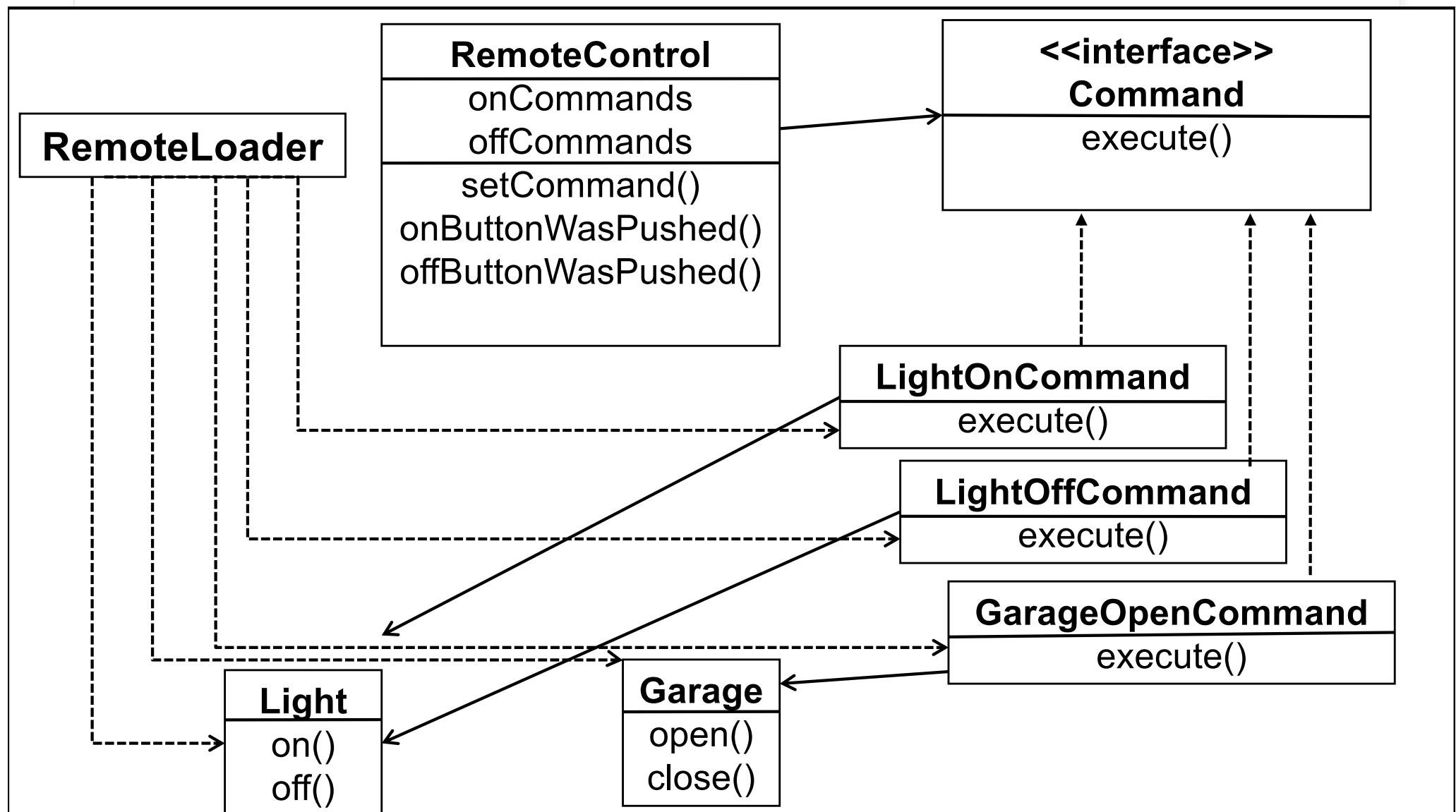
```
Public class RemoteLoader{
 public static void main(String[] args){
 RemoteControl remoteControl = new RemoteControl();
 Light livingRoomLight = new Light("Living Room");
 Light kitchenLight = new Light("Kitchen");
 CeilingFan ceilingFan= new CeilingFan("Living Room");
 GarageDoor garageDoor = new GarageDoor("");
 Stereo stereo = new Stereo("Living Room");

 //aanmaken van de commands
 LightOnCommand living = new LightOnCommand(livingroomLight);
 LightOffCommand living2 = new LightOffCommand(livingroomLight);
 . . .

 //instellen van de knoppen door gebruik te maken van commands
 remoteControl.setCommand(1,living,living2);
 . . .

 //gebruiken van knoppen
 remoteControl.onButtonWasPushed(2);
 remoteControl.offButtonWasPushed(2);
 remoteControl.onButtonWasPushed(3);
 remoteControl.offButtonWasPushed(3);
 }
}
```

# Command



# Command

- En wat met de UNDO-knop?

```
Public interface Command{
 public void execute();
 public void undo();
}
```

# Command

- Voorbeeld: LightOffCommand

```
public class LightOffCommand implements Command{
 Light light;

 public LightOffCommand(Light light) {
 this.light = light;
 }
 public void execute() {
 light.off();
 }
 public void undo() {
 light.on();
 }
}
```

# Command

- Wat met de undo-knop op de afstandsbediening?
  - Voeg een attribuut Command undoCommand toe
    - Bevat het laatste uitgevoerde command
    - Wanneer undo() van afstandsbediening wordt opgeroepen, roepen we zelf undoCommand.undo() op!
    - Bij het indrukken van een knop: instellen van undoCommand!

# Command

- Macro-commando's
  - Meerdere commands groeperen in 1 command

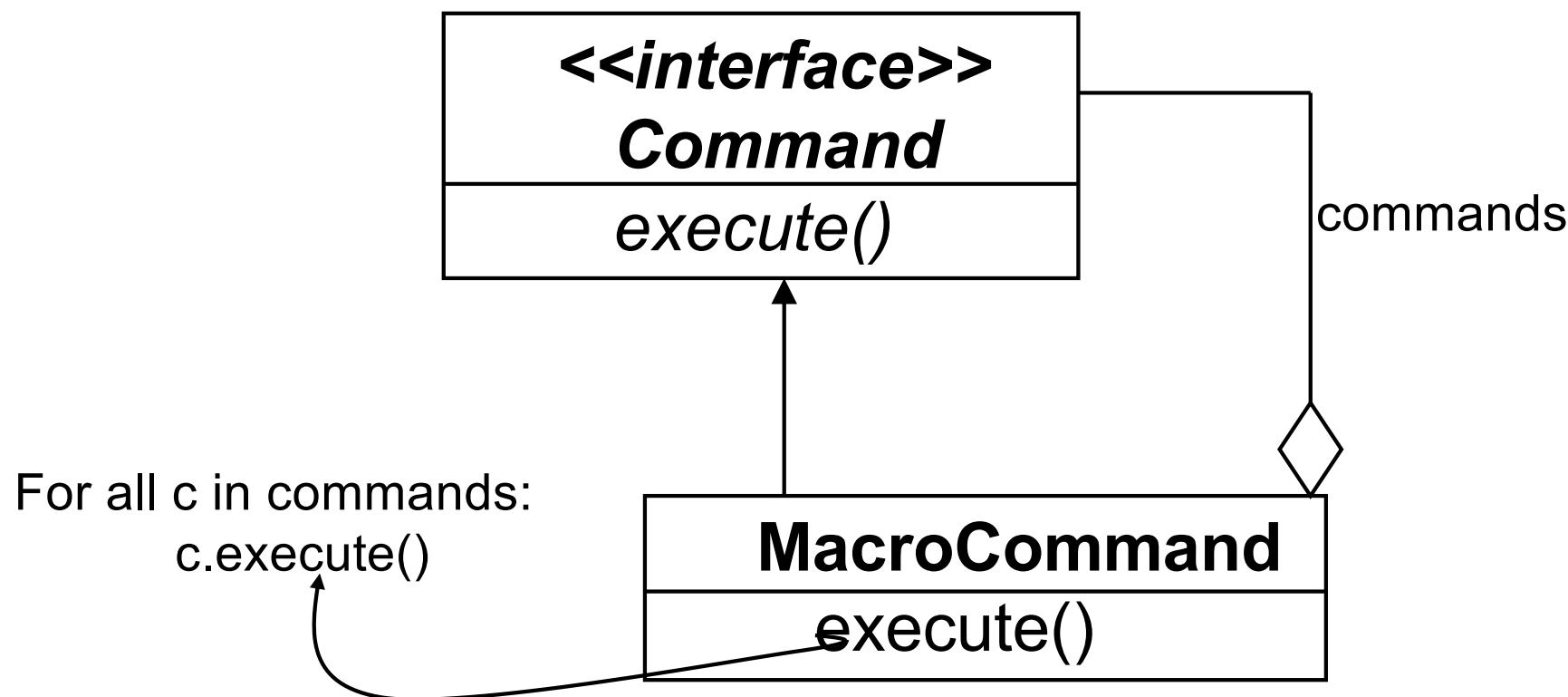
```
Public class MacroCommand implements Command{
 Command[] commands;
 public MacroCommand(Command[] commands) {
 this.commands = commands;
 }

 public void execute() {
 for(int i=0; i<commands.length; i++)
 commands[i].execute();
 }

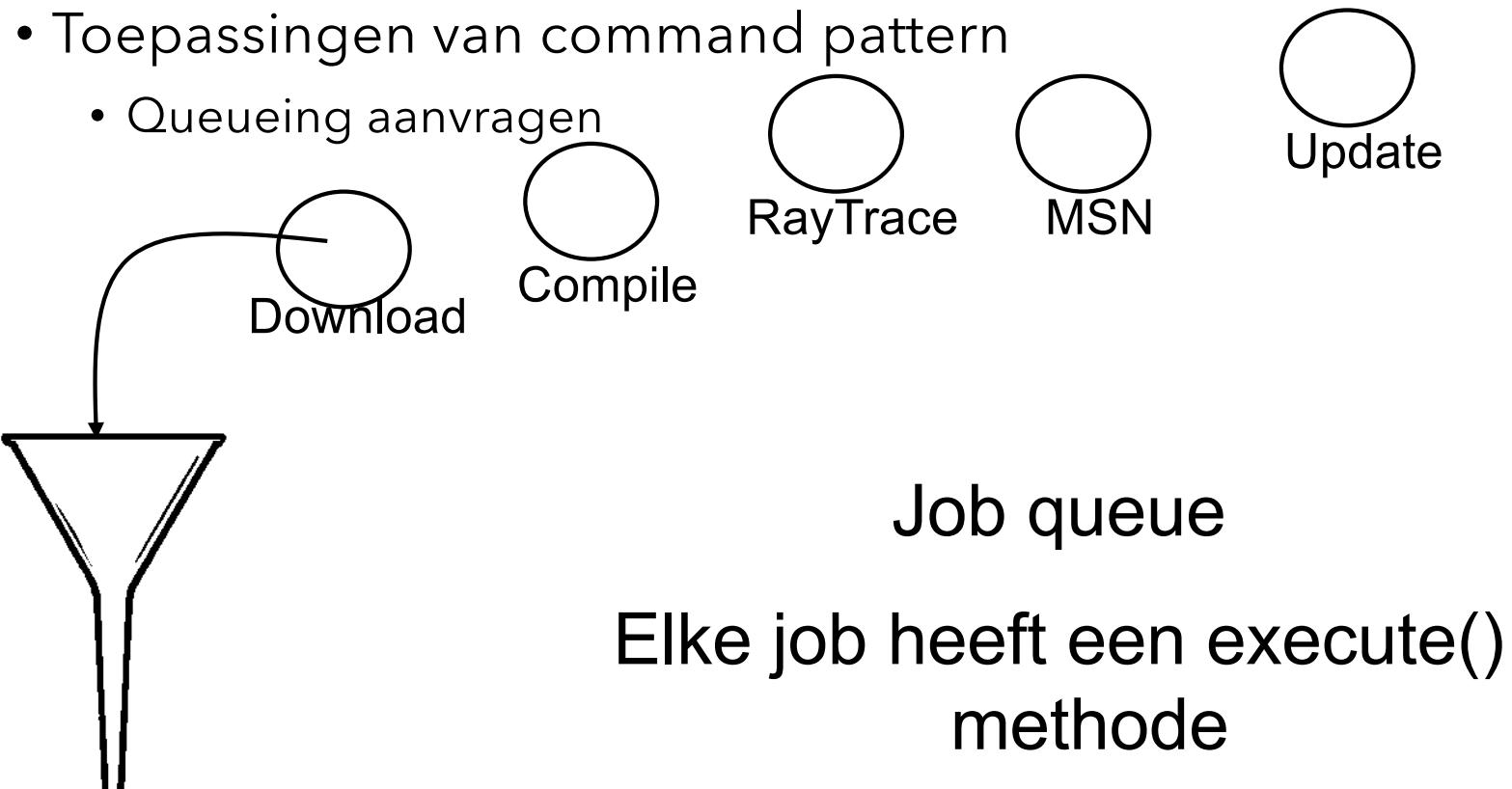
 public void undo(){
 for(int i=0; i<commands.length; i++)
 commands[i].undo();
 }
}
```

# Command

- Macro-commando's



# Command

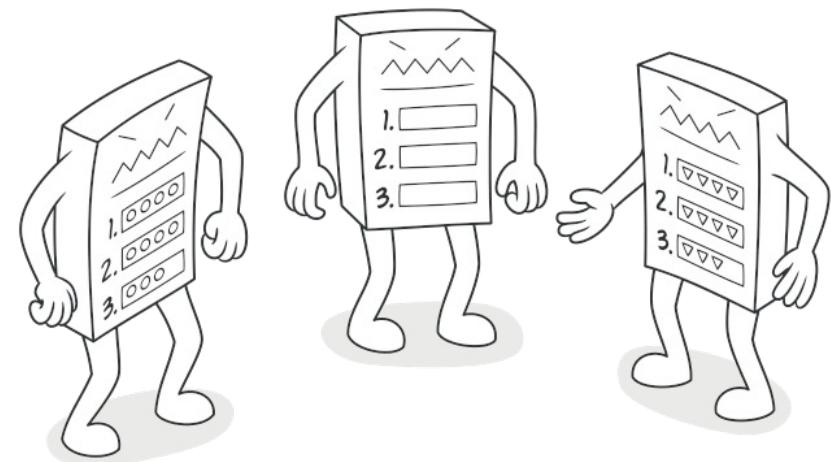


# Template method



# Template Method

- Template Method is a design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.



# Template method

- Tijd voor cafeïne...

|                                  |     |                            |
|----------------------------------|-----|----------------------------|
| • Koffie                         | vs. | Thee                       |
| 1. Water koken                   |     | Water koken                |
| 2. Heet water op koffie schenken |     | Thee laten trekken         |
| 3. Koffie inschenken             |     | Thee inschenken            |
| 4. Suiker en melk toevoegen      |     | Schijfje citroen toevoegen |

# Template method

```
public class Coffee{
 void prepareRecipe() {
 boilWater();
 brewCoffeeGrinds();
 pourInCup();
 addSugarAndMilk();
 }

 public void boilWater()
 {System.out.println("Kook water");}

 public void brewCoffeeGrinds()
 {System.out.println("Laat koffie door filter lopen");}

 public void pourInCup()
 {System.out.println("Schenk in een kopje");}

 public void addSugarAndMilk()
 {System.out.println("Voeg suiker en melk toe");}
}
```

# Template method

```
public class Tea{
 void prepareRecipe() {
 boilWater();
 steepTeaBag();
 pourInCup();
 addLemon();
 }

 public void boilWater()
 {System.out.println("Kook water");}

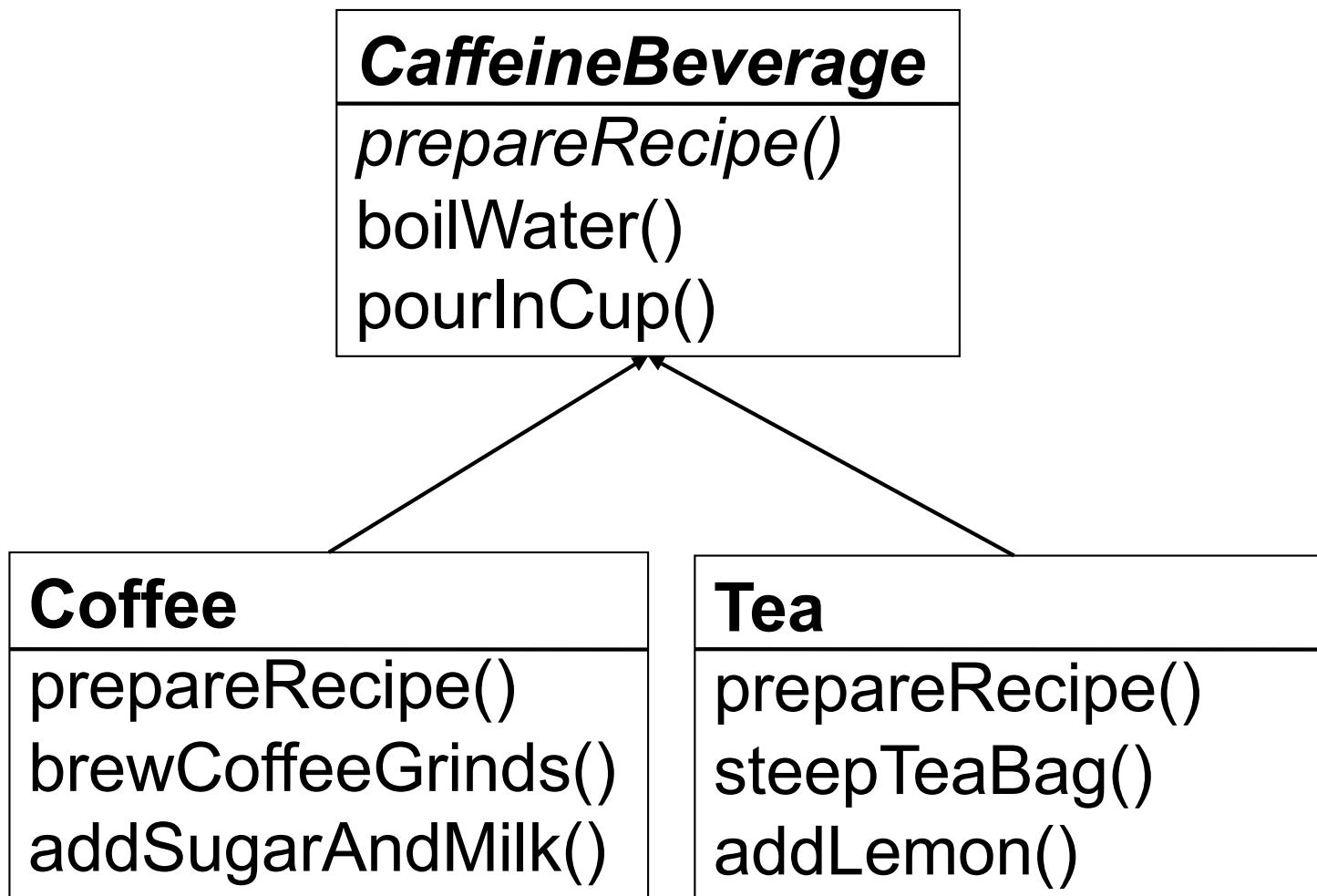
 public void steepTeaBag()
 {System.out.println("Laat de thee trekken");}

 public void pourInCup()
 {System.out.println("Schenk in een kopje");}

 public void addLemon()
 {System.out.println("Voeg citroen toe");}
}
```

Gemeenschappelijke  
code

# Template method



# Template method

- Zijn er nog gemeenschappelijke delen?
  - JA!
  - Recept zelf, volgt een stramien:
    - Water koken
    - Heet water gebruiken om koffie of thee te zetten
    - De drank in een kopje schenken
    - De juiste smaakstoffen toevoegen

# Template method

Coffee

```
void prepareRecipe() {
 boilWater();
 brewCoffeeGrinds();
 pourInCup();
 addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
 boilWater();
 ≠
 steepTeaBag();
 pourInCup();
 addLemon();
}

void prepareRecipe() {
 boilWater();
 brew();
 pourInCup();
 addCondiments();
}
```

# Template method

```
public abstract class CaffeineBeverage{
 final void prepareRecipe() {
 boilWater();
 brew();
 pourInCup();
 addCondiments();
 }

 public void boilWater(
 {System.out.println("Kook water");}

 public void pourInCup()
 {System.out.println("Schenk in een kopje");}

 abstract void brew();
 void addCondiments() { }
}
```

# Template method

```
public class Tea extends CaffeineBeverage{
 public void brew() {
 System.out.println("Laat de thee trekken");
 }
 public void addCondiments() {
 System.out.println("Voeg citroen toe");
 }
}

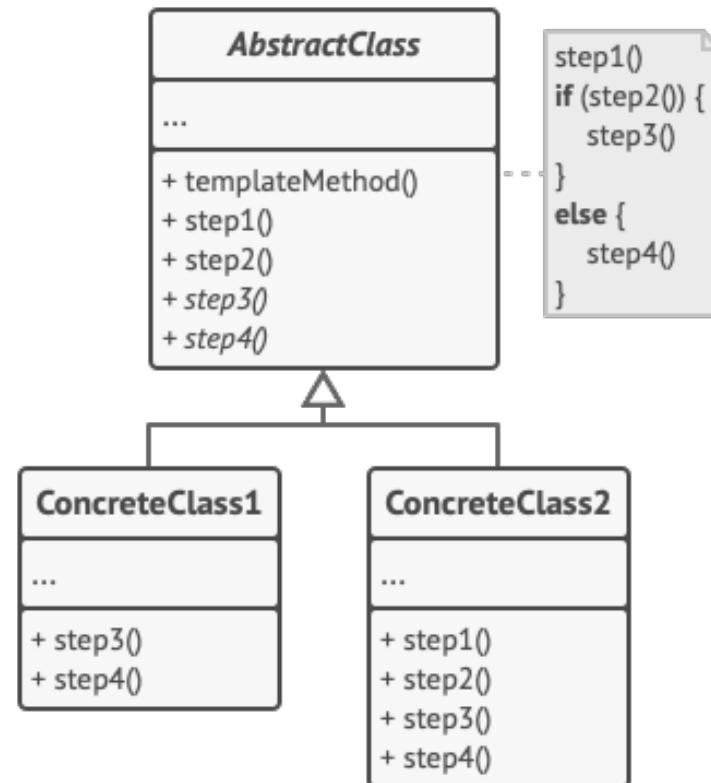
public class Coffee extends CaffeineBeverage{
 public void brew() {
 System.out.println("Laat koffie door filter lopen");
 }
 public void addCondiments() {
 System.out.println("Voeg suiker en melk toe");
 }
}
```

# Template method

- Het **template method** patroon definieert het skelet van een algoritme in een methode, waarbij sommige stappen aan subklassen worden overgelaten. De template method laat subklassen bepaalde stappen in een algoritme herdefiniëren zonder de structuur van het algoritme te veranderen.

# Template method

- The Abstract Class declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared abstract or have some default implementation.



# Template method

```
abstract class AbstractClass{
 final void templateMethod() {
 primitiveOperation1();
 primitiveOperation2();

 hook();
 concreteOperation();
 }
 abstract void primitiveOperation1();
 abstract void primitiveOperation2();
 public void hook() {

 }

 void concreteOperation() {
 //implementatie
 }
}
```

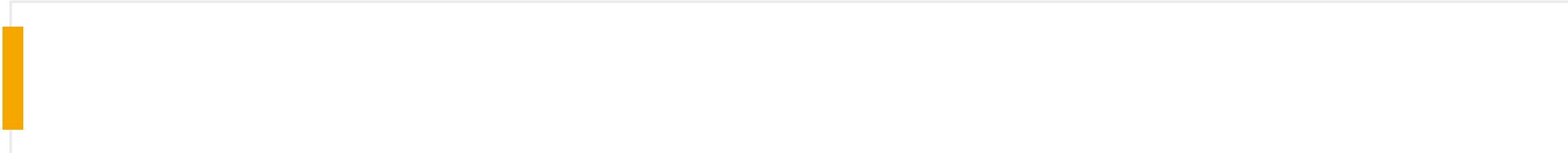
# Template method

- Uitbreidingen op template method
  - Hooks
    - Een method waarvoor een standaard-implementatie wordt voorzien in de superklasse
    - De subklassen kunnen dit (naar keuze) wel overschrijven

# Template method

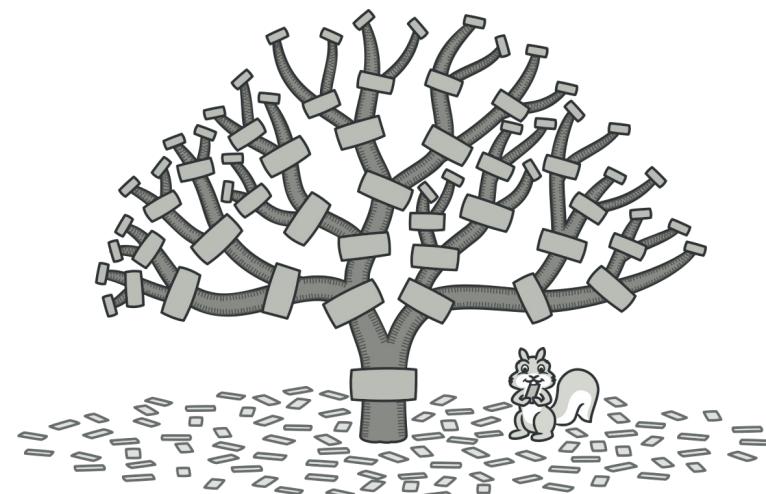
- Bedenkingen
  - Kiezen tussen abstracte methode en hook
    - Abstract indien de subklasse een implementatie MOET voorzien
    - Hook indien de subklasse een implementatie MAG voorzien
  - Wanneer hooks gebruiken?
    - Subklasse kan een optioneel deel van een algoritme implementeren **of overslaan**

# Composite



# Composite

- Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



# Composite

- Stel: menukaart met submenu's in een restaurant
  - Desserten
  - Voorgerechten
  - Hoofdgerechten
    - Vlees
    - Vis
    - Wild
  - Dranken

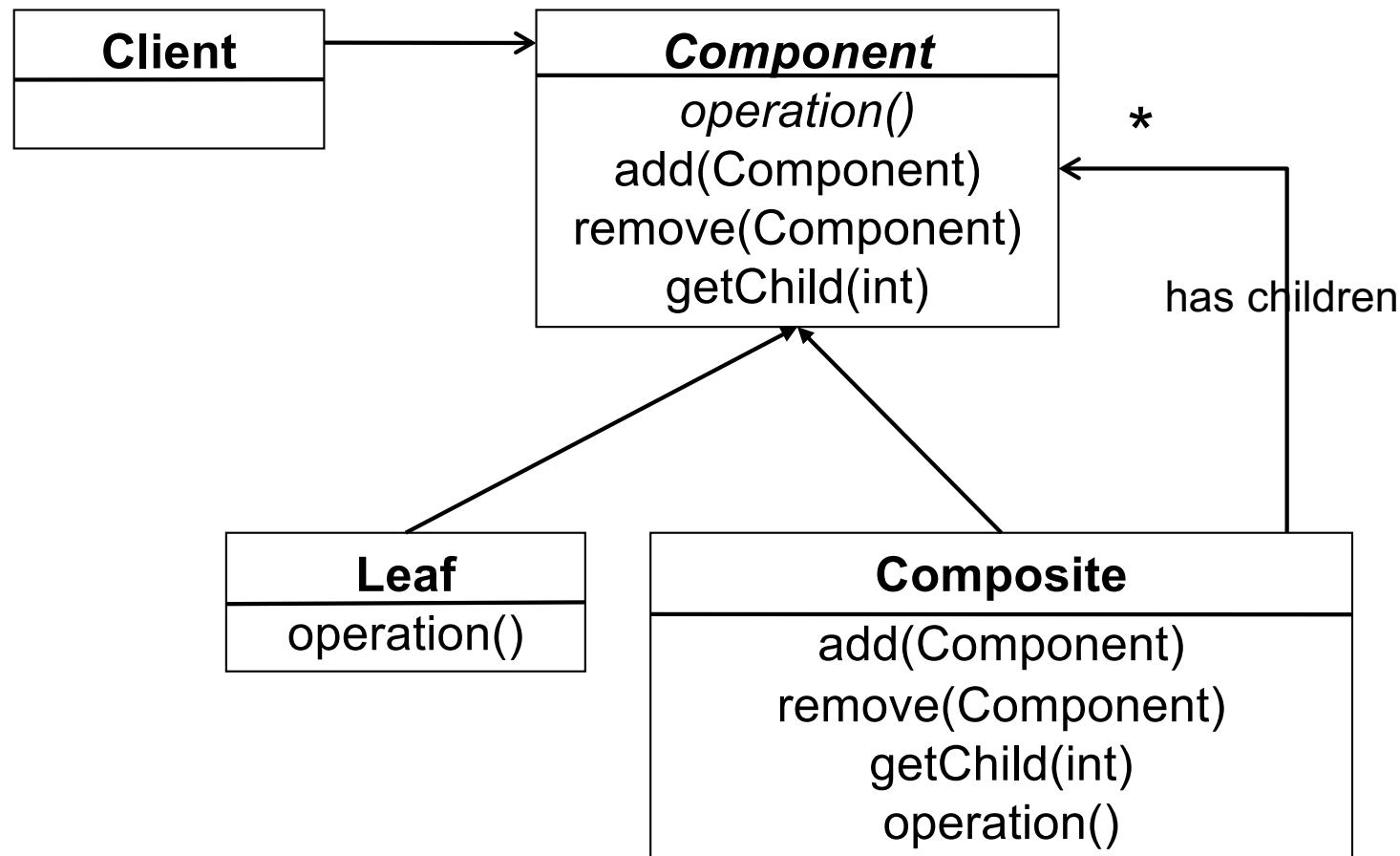
# Composite

- Nodig
  - Een soort boomstructuur voor de menu's
    - Menu bevat menu-item(s) en/of submenu(s)
  - Makkelijk itereren
  - Flexibiliteit
    - Itereren door alleen het dessertmenu
    - Itereren door het restaurantmenu inclusief dessertmenu

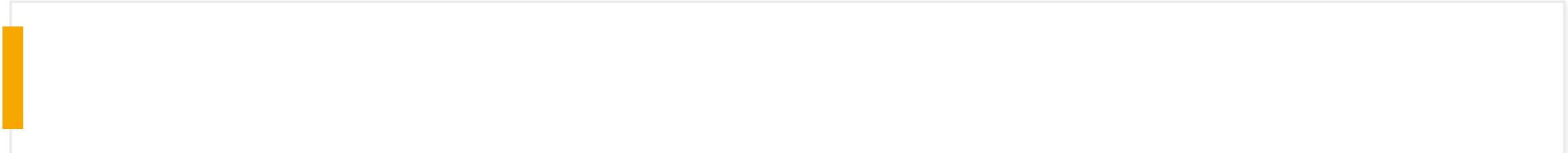
# Composite

- Het **composite patroon** stelt je in staat om objecten in boomstructuren samen te stellen om part-whole hiërarchieën weer te geven. Composite laat clients de afzonderlijke objecten of samengestelde objecten op uniforme wijze te behandelen.

# Composite



# 4 elements of composite pattern



## Component

Declares abstract class for objects in composition.

Implements default behavior for the interface common to all classes as appropriate.

Declares an interface for accessing and managing its child components.

## Leaf

Represents leaf objects in composition. A leaf has no children.

Defines behavior for primitive objects in the composition.

## Composite

Defines behaviour for components having children.

Stores child component.

Implements child related operations in the component interface.

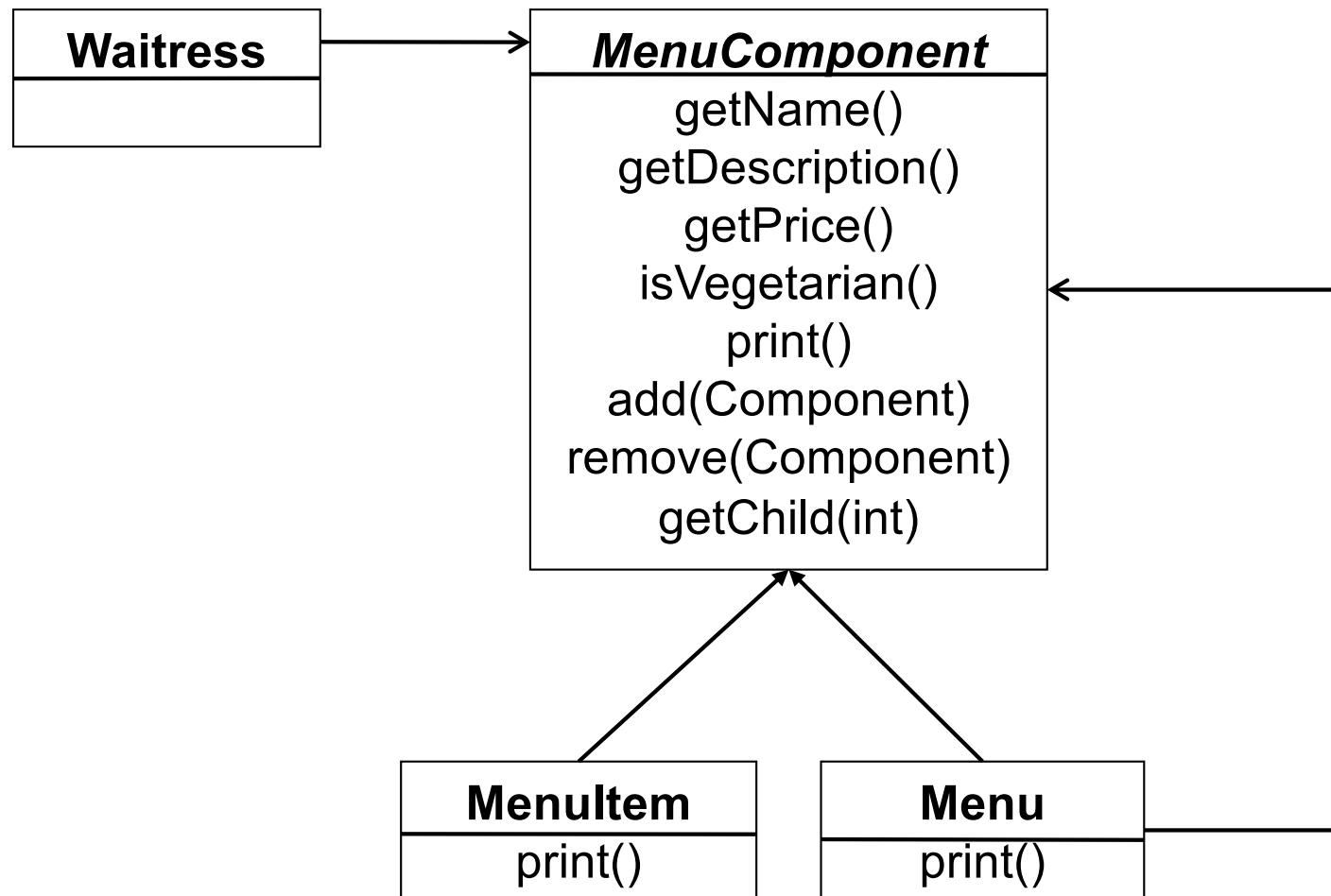
## Client

- Manipulates objects in the composition through the component interface.

# Example



# Composite



```
public abstract class MenuComponent {

 public void add(MenuComponent menuComponent) {
 throw new UnsupportedOperationException();
 }
 public void remove(MenuComponent menuComponent) {
 throw new UnsupportedOperationException();
 }
 public MenuComponent getChild(int i) {
 throw new UnsupportedOperationException();
 }

 public String getName() {
 throw new UnsupportedOperationException();
 }
 public String getDescription() {
 throw new UnsupportedOperationException();
 }
 public double getPrice() {
 throw new UnsupportedOperationException();
 }
 public boolean isVegetarian() {
 throw new UnsupportedOperationException();
 }

 public void print() {
 throw new UnsupportedOperationException();
 }
}
```

```
public class MenuItem extends MenuComponent {
 String name;
 String description;
 boolean vegetarian;
 double price;

 public MenuItem(String name,
 String description,
 boolean vegetarian,
 double price)
 {
 this.name = name;
 this.description = description;
 this.vegetarian = vegetarian;
 this.price = price;
 }

 public String getName() {
 return name;
 }

 public String getDescription() {
 return description;
 }

 public double getPrice() {
 return price;
 }

 public boolean isVegetarian() {
 return vegetarian;
 }

 public void print() {
 System.out.print(" " + getName());
 if (isVegetarian()) {
 System.out.print("(v)");
 }
 System.out.println(", " + getPrice());
 System.out.println(" -- " + getDescription());
 }
}
```

```

public class Menu extends MenuComponent {
 ArrayList menuComponents = new ArrayList();
 String name;
 String description;

 public Menu(String name, String description) {
 this.name = name;
 this.description = description;
 }

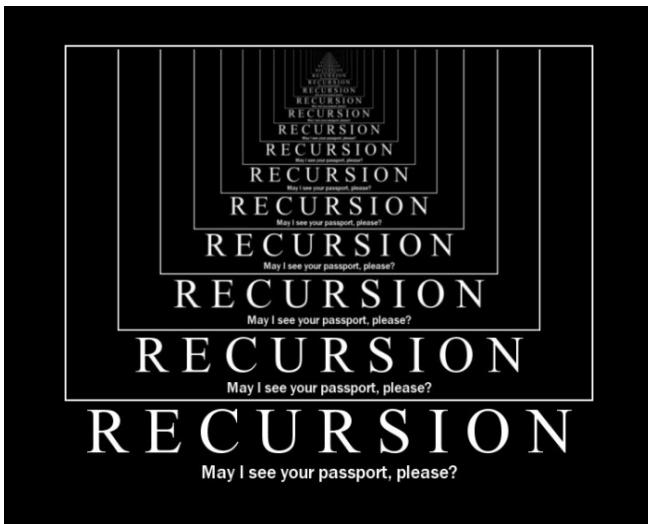
 public void add(MenuComponent menuComponent) {
 menuComponents.add(menuComponent);
 }

 public void remove(MenuComponent menuComponent) {
 menuComponents.remove(menuComponent);
 }

 public MenuComponent getChild(int i) {
 return (MenuComponent)menuComponents.get(i);
 }

 public String getName() {
 return name;
 }
}

```



Recurse!!

```

public void add(MenuComponent menuComponent) {
 menuComponents.add(menuComponent);
}

public void remove(MenuComponent menuComponent) {
 menuComponents.remove(menuComponent);
}

public MenuComponent getChild(int i) {
 return (MenuComponent)menuComponents.get(i);
}

public String getName() {
 return name;
}

public String getDescription() {
 return description;
}

public void print() {
 System.out.print("\n" + getName());
 System.out.println(", " + getDescription());
 System.out.println("-----");
}

Iterator iterator = menuComponents.iterator();
while (iterator.hasNext()) {
 MenuComponent menuComponent =
 (MenuComponent) iterator.next();
 menuComponent.print();
}
}

```

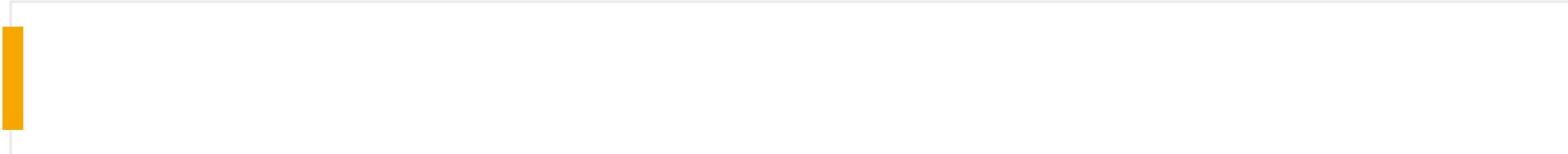
# Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

# Alternative implementation

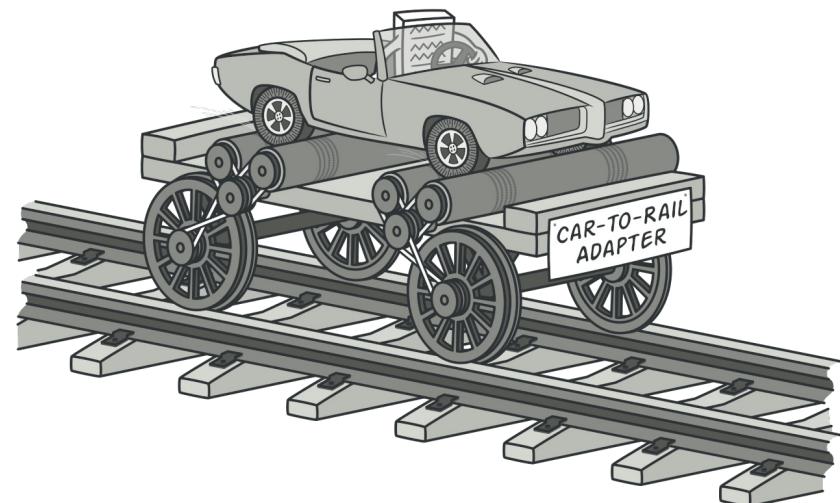
- Composite pattern has one problem - how should leaf classes that doesn't have internal collection handle add(), remove()? Should they ignore these calls or should they throw exception?
- IsComposite property helps us to solve the problem. Leaf classes return false and composite classes return true.

# Adapter



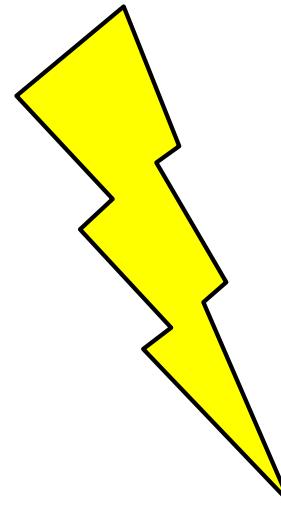
# Adapter

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.



# Adapter

- Probleem
  - Verschillende soorten interfaces



# Adapter

- Oplossing



# Adapter

- In de software-wereld...



Bestaande systeem



Leveranciersklassen

# Adapter

- In de software-wereld...



Bestaande systeem



adapter



Leveranciersklassen

Enige nieuwe code

# Adapter

- Stel: vereenvoudigde Duck-interface

```
Public interface Duck{
 public void quack();
 public void fly();
}
```

- Bv.: de MallardDuck

```
Public class MallardDuck implements Duck{
 public void quack() {System.out.println("Quack");}
 public void fly() {System.out.println("Fly...");}
}
```

# Adapter

- Stel: ook kalkoenen worden mogelijk

```
Public interface Turkey{
 public void gobble();
 public void fly();
}
```

- Bv.: de WildTurkey

```
Public class WildTurkey implements Turkey{
 public void gobble() {System.out.println("Gobble");}
 public void fly() {System.out.println("Short fly...");}
}
```

# Adapter

- Invloegen van een TurkeyAdapter

```
Public class TurkeyAdapter implements Duck{
 Turkey turkey;
 public TurkeyAdapter(Turkey turkey) {
 this.turkey = turkey;
 }
 public void quack() {
 turkey.gobble();
 }
 public void fly() {
 for(int i=0;i<5;i++)
 turkey.fly();
 }
}
```

# Adapter

- Invloegen van een TurkeyAdapter...praktijk

```
Public class DuckTester{
 public static void main(String[] args) {
 MallardDuck duck = new MallardDuck();

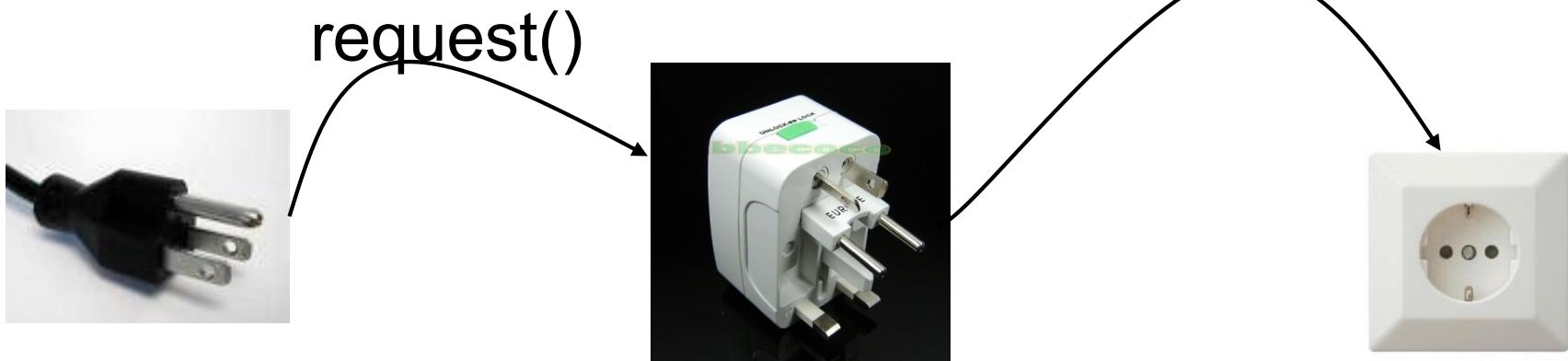
 WildTurkey turkey = new WildTurkey();
 Duck turkeyAdapter = new TurkeyAdapter(turkey);

 testDuck(duck);
 testDuck(turkeyAdapter);
 }

 public static testDuck(Duck duck) {
 duck.quack();
 duck.fly();
 }
}
```

# Adapter

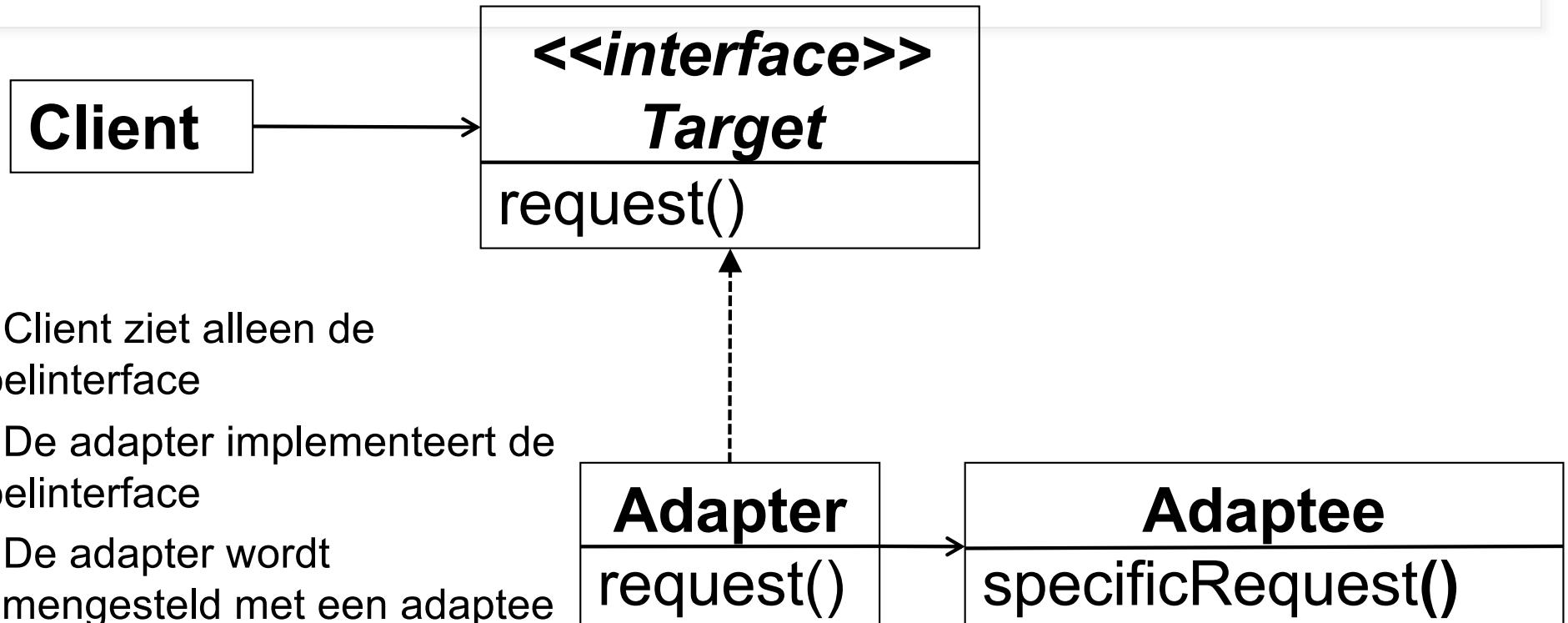
- Het principe...



# Adapter

- **Het Adapter patroon** converteert de interface van een klasse naar een andere interface die de client verwacht. Adapters zorgen ervoor dat klassen samenwerken. Zonder de adapters lukt dit niet vanwege incompatibele interfaces.

# Adapter (**object** adapter)



- Client ziet alleen de doelinterface
- De adapter implementeert de doelinterface
- De adapter wordt samengesteld met een adaptee
- Alle aanvragen worden gedelegeerd aan de adaptee

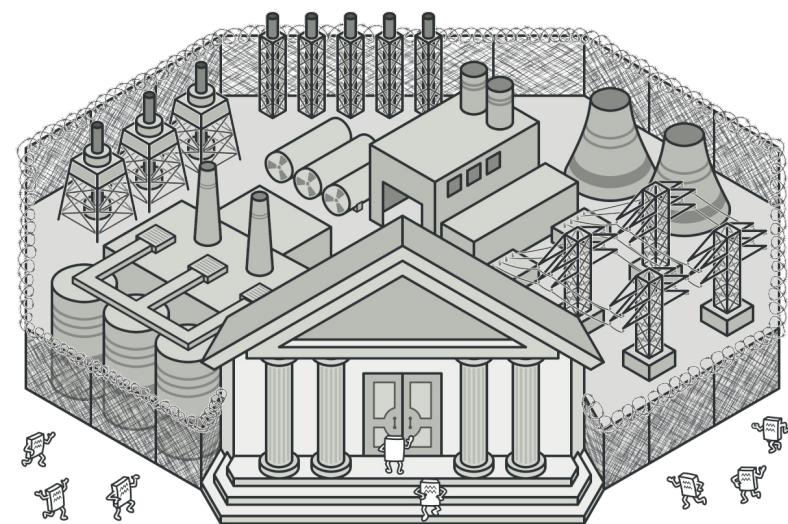


# Facade

---

# Facade

- **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



# Facade

- Probleem
  - Interface kan soms heel ingewikkeld zijn
  - Facade kan vóór deze interface gaan staan en een eenvoudigere interface aanbieden
- Voorbeeld
  - Thuistheater: klassen Amplifier, Tuner, DvdPlayer, Screen, CdPlayer, Projector, DvdPlayer, PopcornPopper, TheaterLights, ...

# Facade

- Voorbeeld: thuistheater
  - Een film bekijken...op de moeilijke manier
    - Popcornmachine aanzetten
    - Popcornmachine opstarten
    - De verlichting dimmen
    - Het scherm neerlaten
    - De beamer aanzetten
    - De beamer op dvd instellen
    - De beamer op breedbeeld instellen
    - De versterker aanzetten
    - De versterker op dvd input instellen
    - De versterker op surround sound instellen
    - Het volume van de versterker op gemiddeld zetten
    - De dvd-speler aanzetten
    - De dvd-speler starten

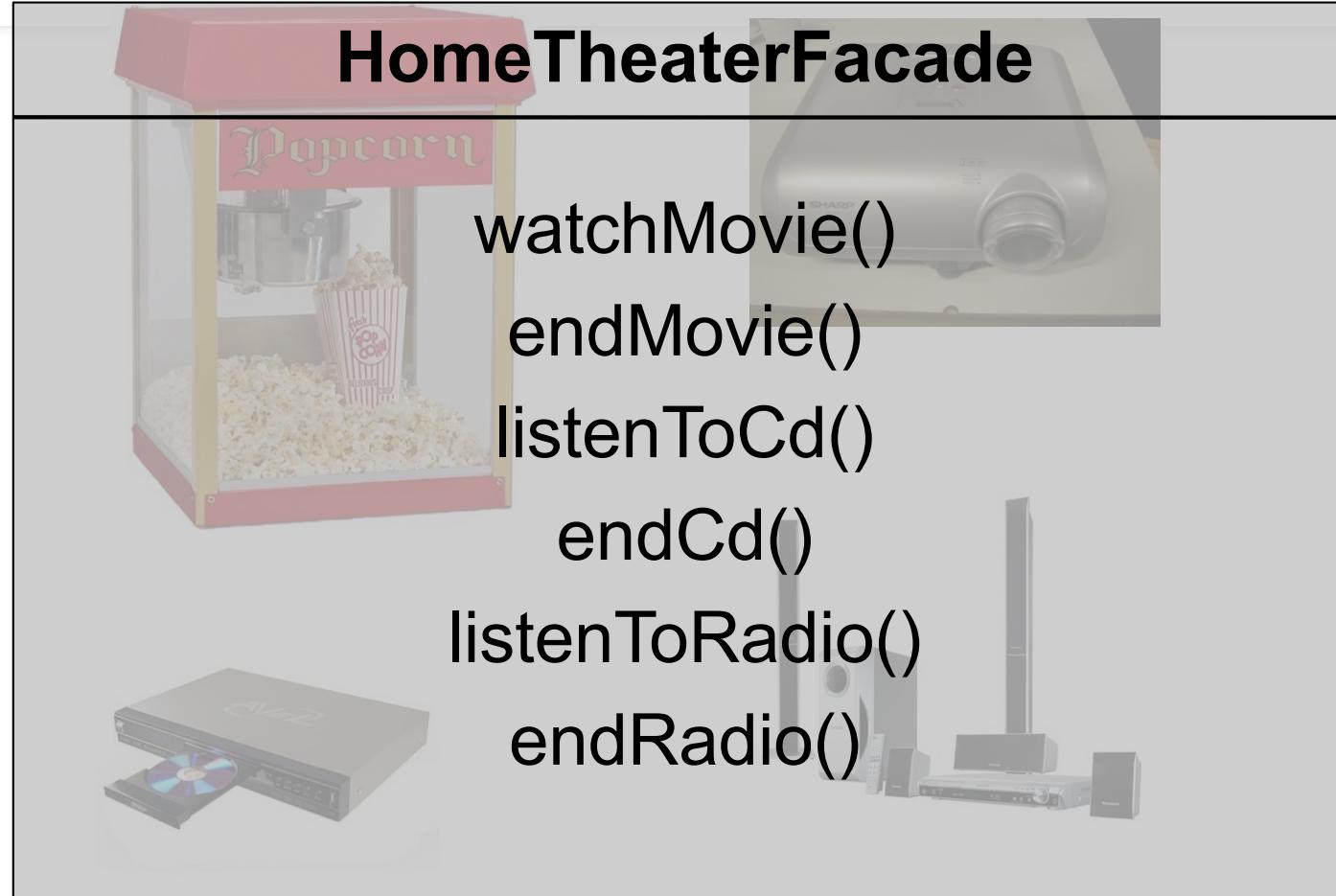


# Facade

- Voorbeeld: thuistheater

```
popper.on();
popper.pop();
lights.dim(10);
screen.down();
projector.on();
projector.setIntput(dvd);
projector.setScreenMode(wide);
amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);
dvd.on();
dvd.play(movie);
```

# Facade



```
public class HomeTheaterFacade {
 Amplifier amp;
 Tuner tuner;
 DvdPlayer dvd;
 CdPlayer cd;
 Projector projector;
 TheaterLights lights;
 Screen screen;
 PopcornPopper popper;

 public HomeTheaterFacade(Amplifier amp,
 Tuner tuner,
 DvdPlayer dvd,
 CdPlayer cd,
 Projector projector,
 Screen screen,
 TheaterLights lights,
 PopcornPopper popper) {

 this.amp = amp;
 this.tuner = tuner;
 this.dvd = dvd;
 this.cd = cd;
 this.projector = projector;
 this.screen = screen;
 this.lights = lights;
 this.popper = popper;
 }

 public void watchMovie(String movie) {
 System.out.println("Get ready to watch a movie...");
 popper.on();
 popper.pop();
 lights.dim(10);
 screen.down();
 projector.on();
 projector.wideScreenMode();
 amp.on();
 amp.setDvd(dvd);
 amp.setSurroundSound();
 amp.setVolume(5);
 dvd.on();
 dvd.play(movie);
 }

 public void endMovie() {
 System.out.println("Shutting movie theater down...");
 popper.off();
 lights.on();
 screen.up();
 projector.off();
 amp.off();
 dvd.stop();
 dvd.eject();
 dvd.off();
 }

 public void listenToCd(String cdTitle) {
 System.out.println("Get ready for an audiophile experience...");
 lights.on();
 amp.on();
 amp.setVolume(5);
 amp.setCd(cd);
 amp.setStereoSound();
 cd.on();
 cd.play(cdTitle);
 }

 public void endCd() {
 System.out.println("Shutting down CD...");
 amp.off();
 amp.setCd(cd);
 cd.eject();
 cd.off();
 }
}
```

# Facade

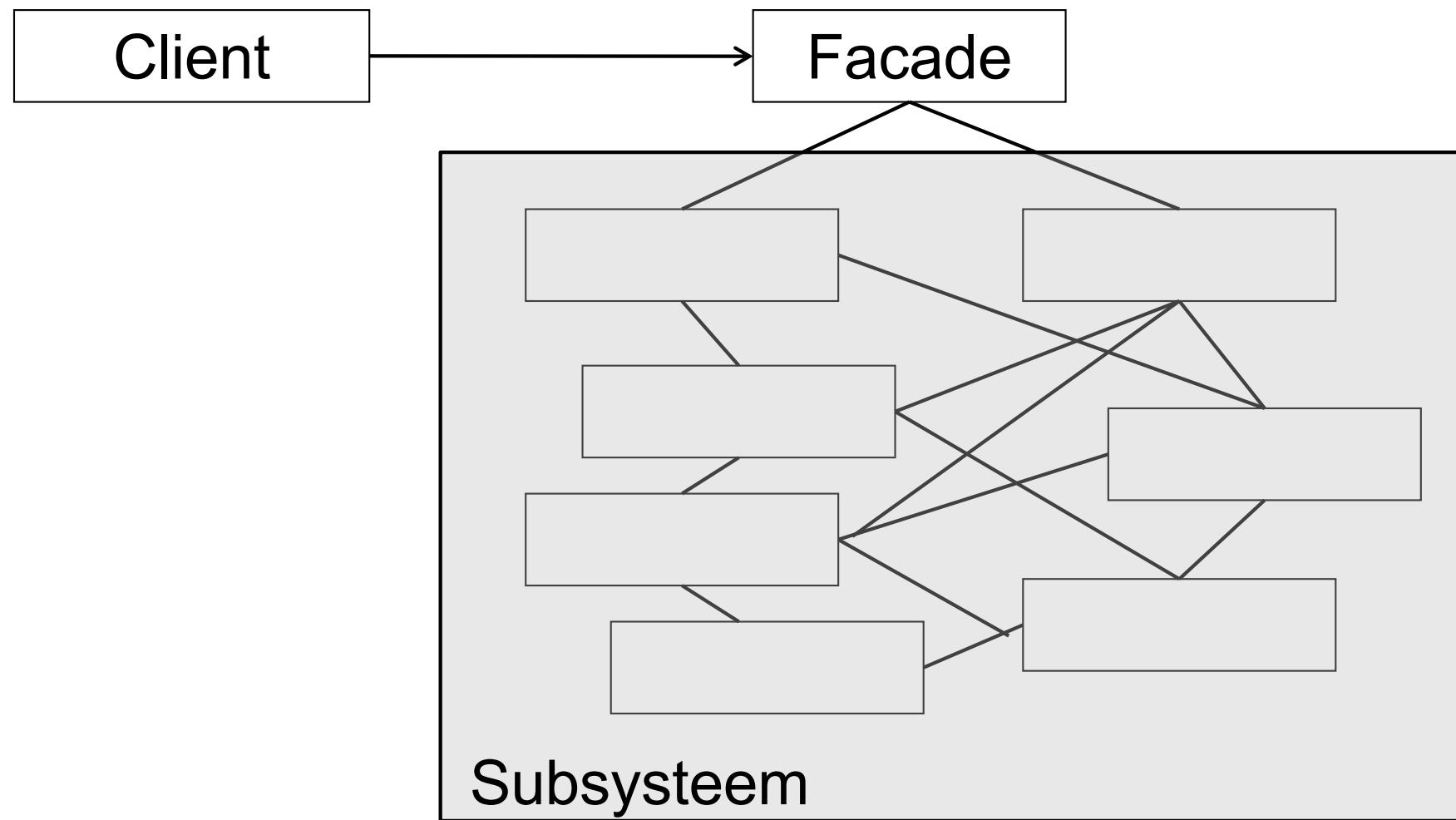
- **Het Facade patroon** zorgt voor een vereenvoudigde interface naar een verzameling interfaces in een subsysteem. De facade definieert een interface op een hoger niveau zodat het gebruik van het systeem vereenvoudigt.

# Facade

- Ontwerpprincipe
  - Principe van de **kennisabstractie**: hoe minder je weet hoe beter. Daardoor zullen klassen niet te veel gekoppeld worden.

**Low coupling, high cohesion**

# Facade



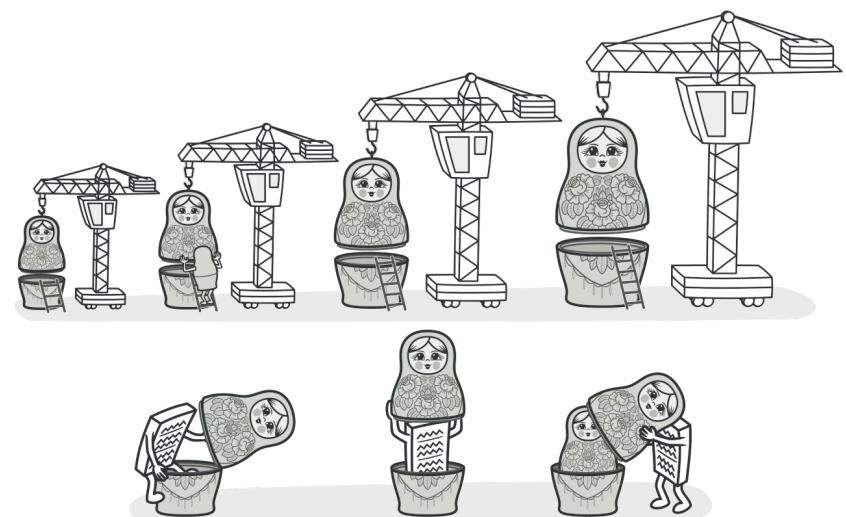


# Decorator

---

# Decorator

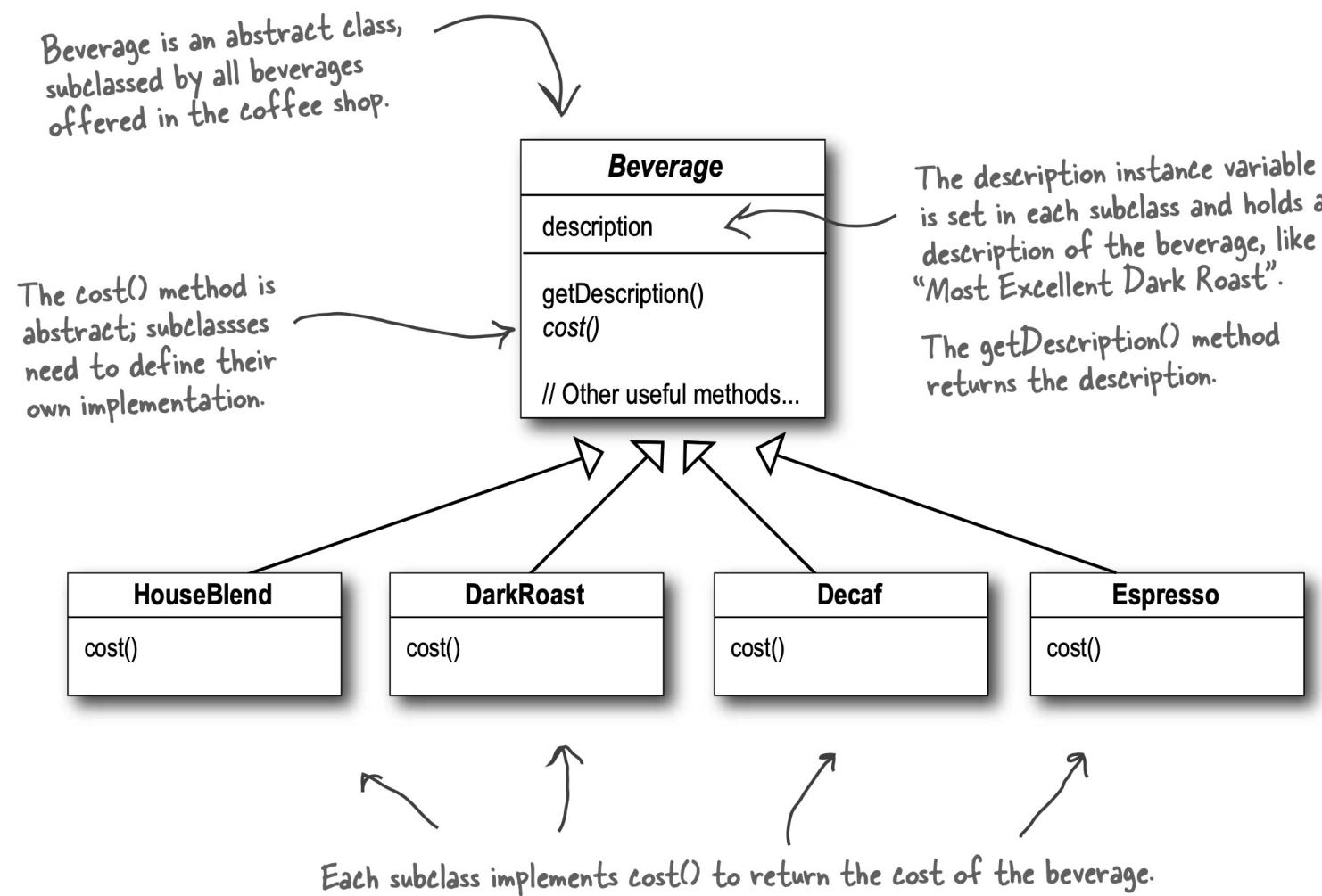
- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



# Starbuzz Coffee

- Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around.
- Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.
- When they first went into business they designed their classes like this...

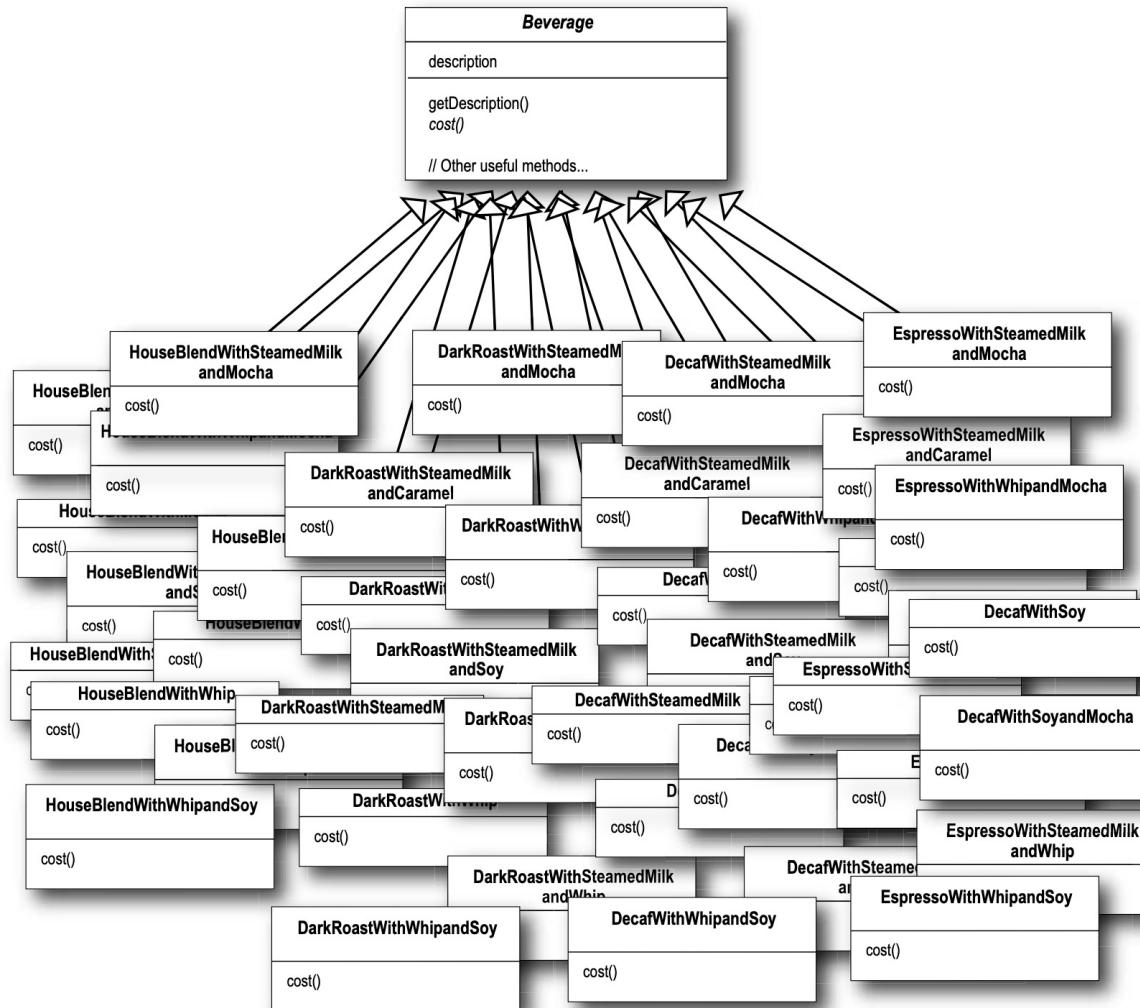
# Starbuzz Coffee



# Starbuzz Coffee

- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.
- Here's their first attempt...

# Starbuzz Coffee





# Meet the Decorator Pattern

- Take a DarkRoast object
- Decorate it with a Mocha object
- Decorate it with a Whip object
- Call the cost() method and rely on delegation to add on the condiment costs

# Constructing a drink order with Decorators

- Beverage abstract class

```
public abstract class Beverage {
 String description = "Unknown Beverage";

 public String getDescription() {
 return description;
 }

 public abstract double cost();
}
```

# Constructing a drink order with Decorators

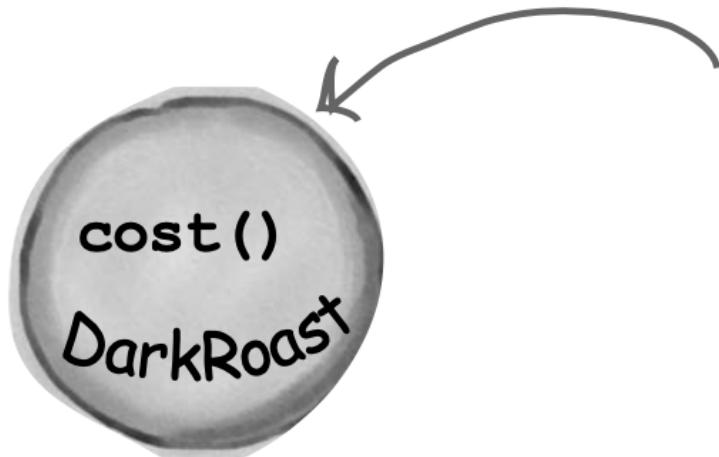
- DarkRoast class

```
public class DarkRoast extends Beverage {
 public DarkRoast() {
 description = "Dark Roast Coffee";
 }

 public double cost() {
 return .99;
 }
}
```

# Constructing a drink order with Decorators

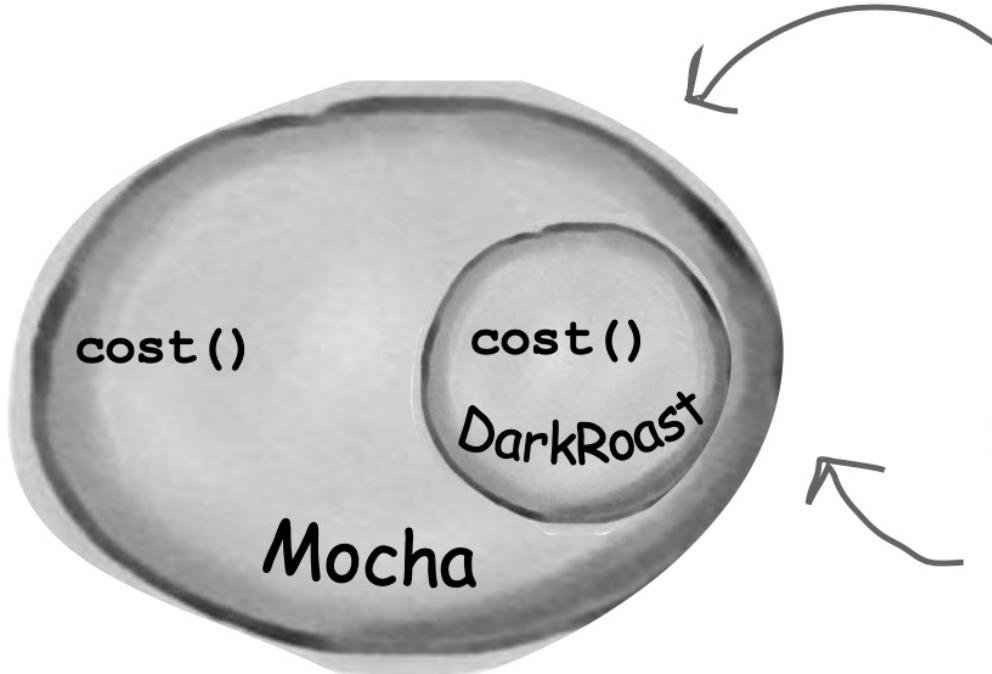
- We start with our DarkRoast object.



Remember that DarkRoast  
inherits from Beverage and has  
a cost() method that computes  
the cost of the drink.

# Constructing a drink order with Decorators

- The customer wants Mocha, so we create a Mocha object

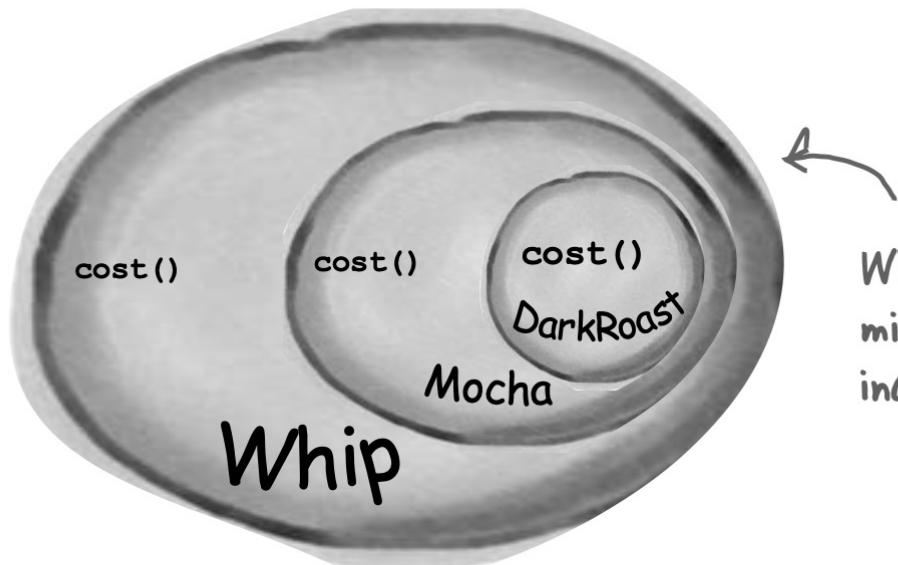


The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a `cost()` method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

# Constructing a drink order with Decorators

- The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



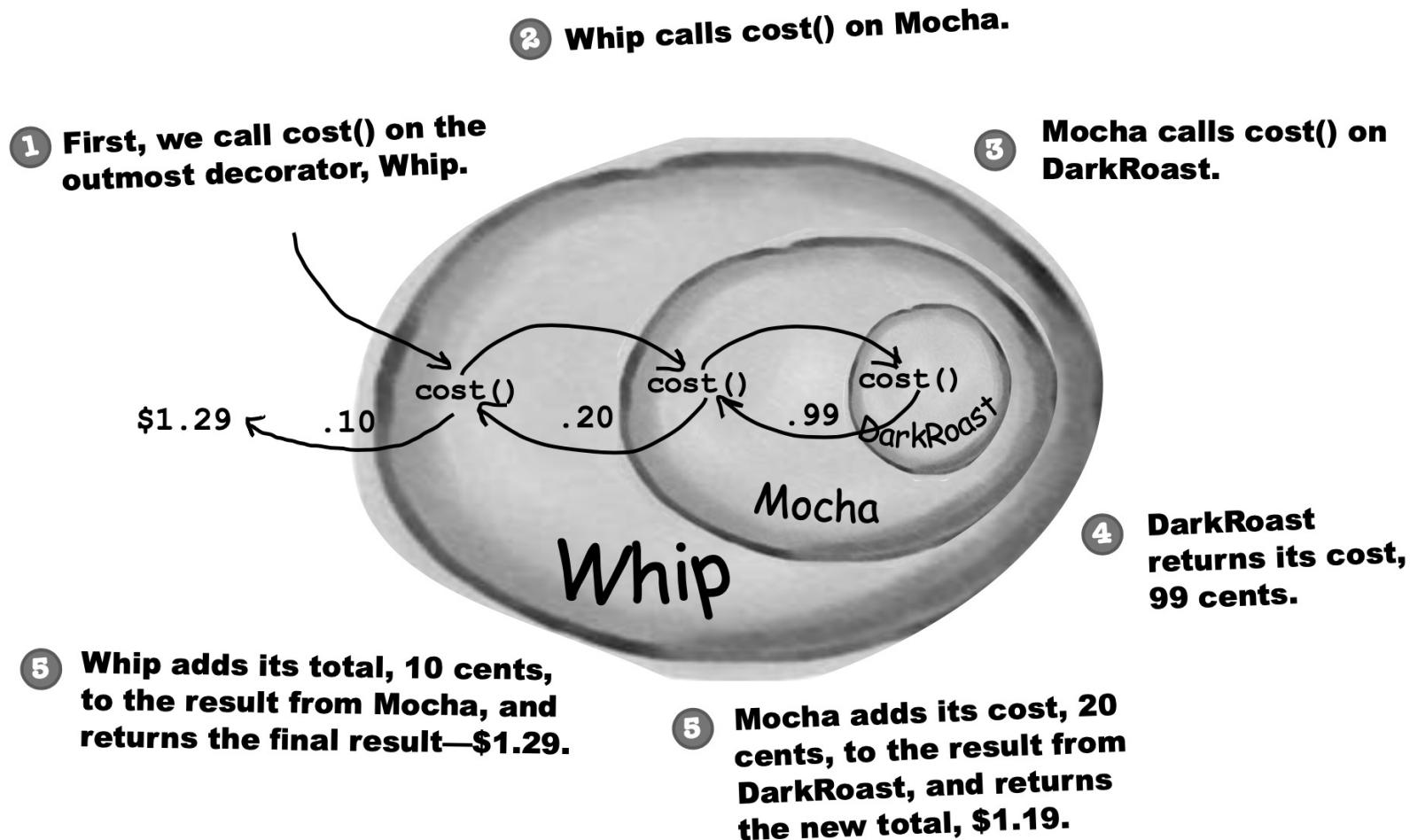
Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

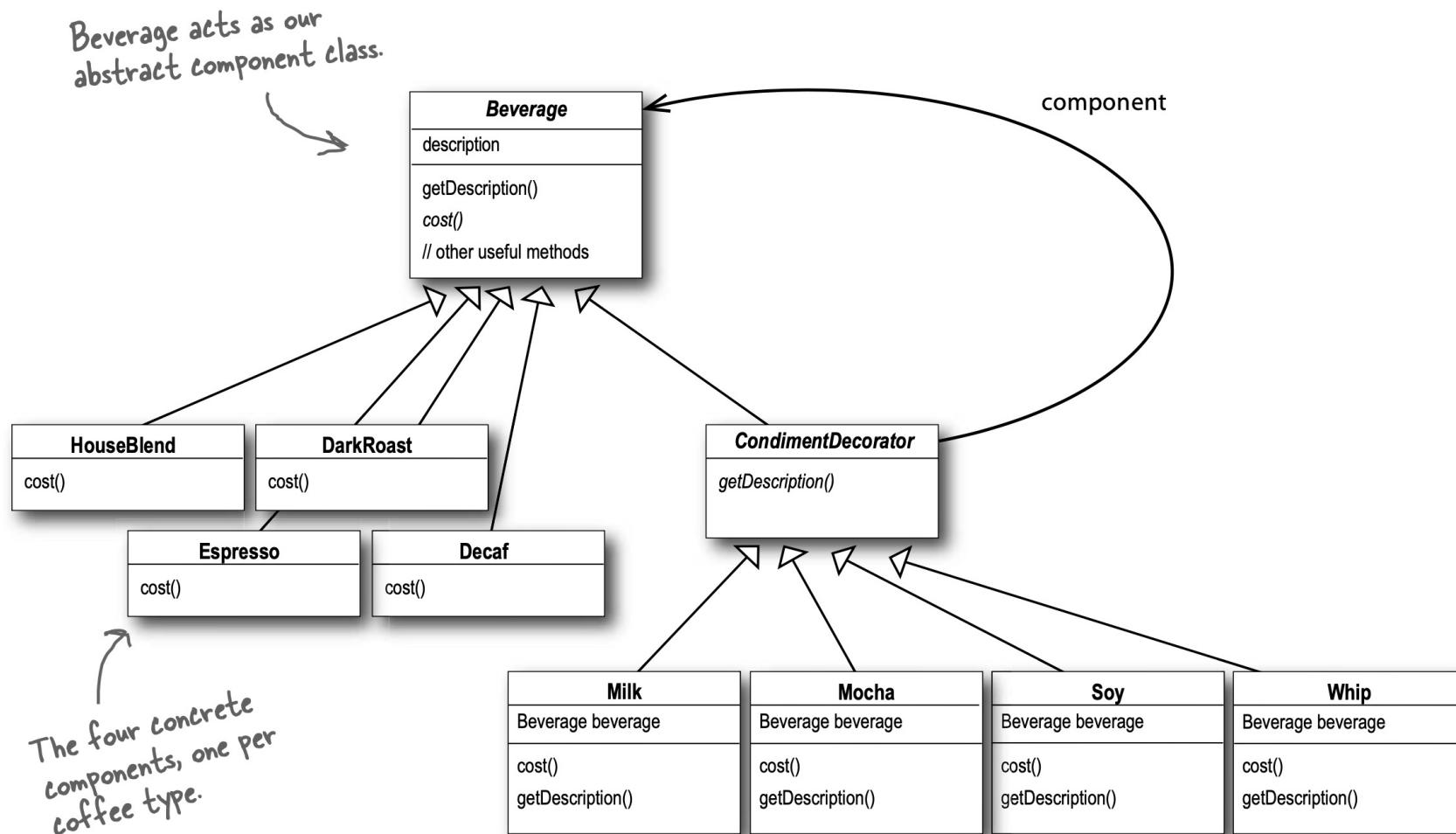
# Constructing a drink order with Decorators

- Now it's time to compute the cost for the customer. We do this by calling `cost()` on the outermost decorator, `Whip`, and `Whip` is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the `Whip`.

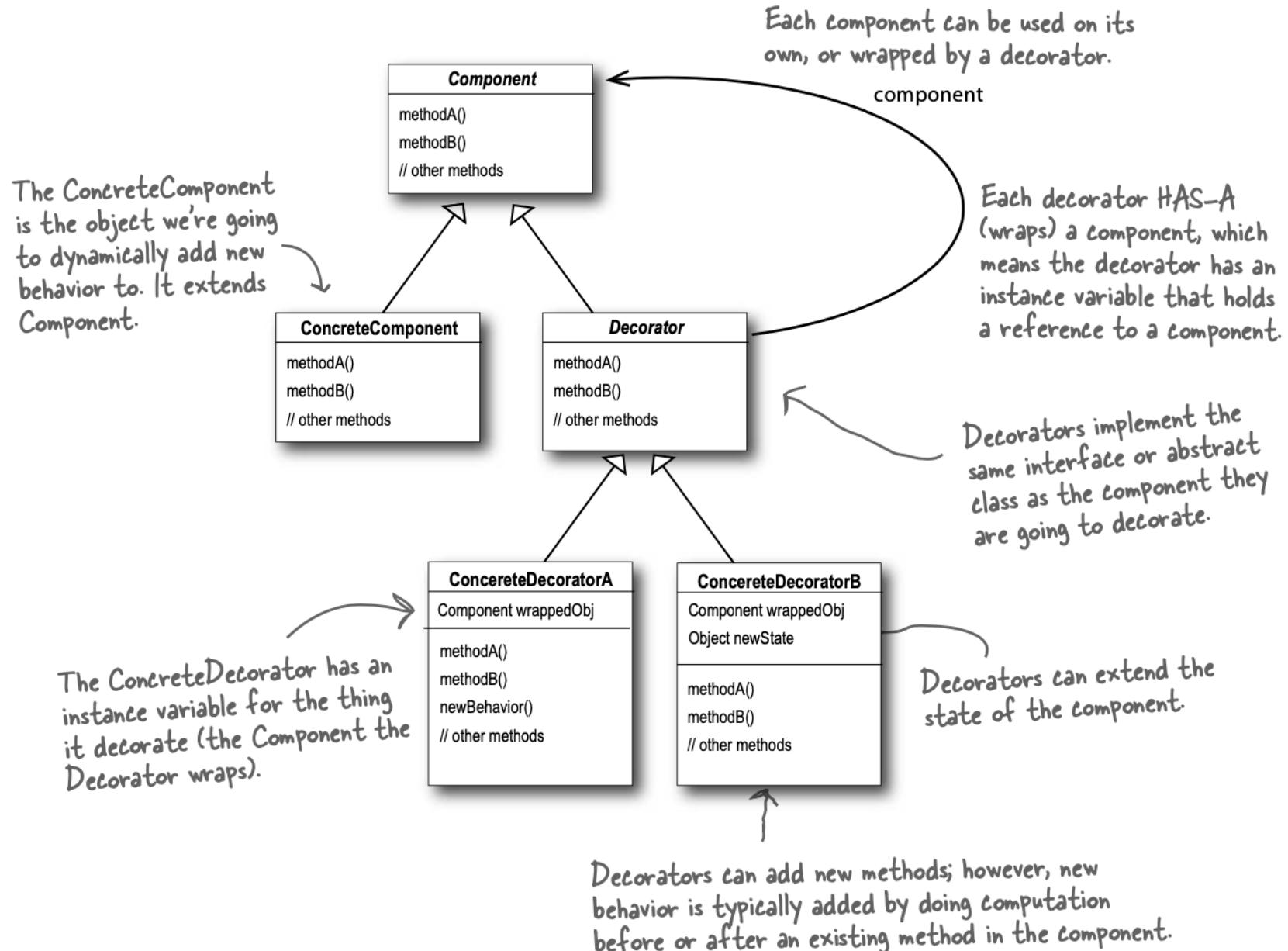
# Constructing a drink order with Decorators



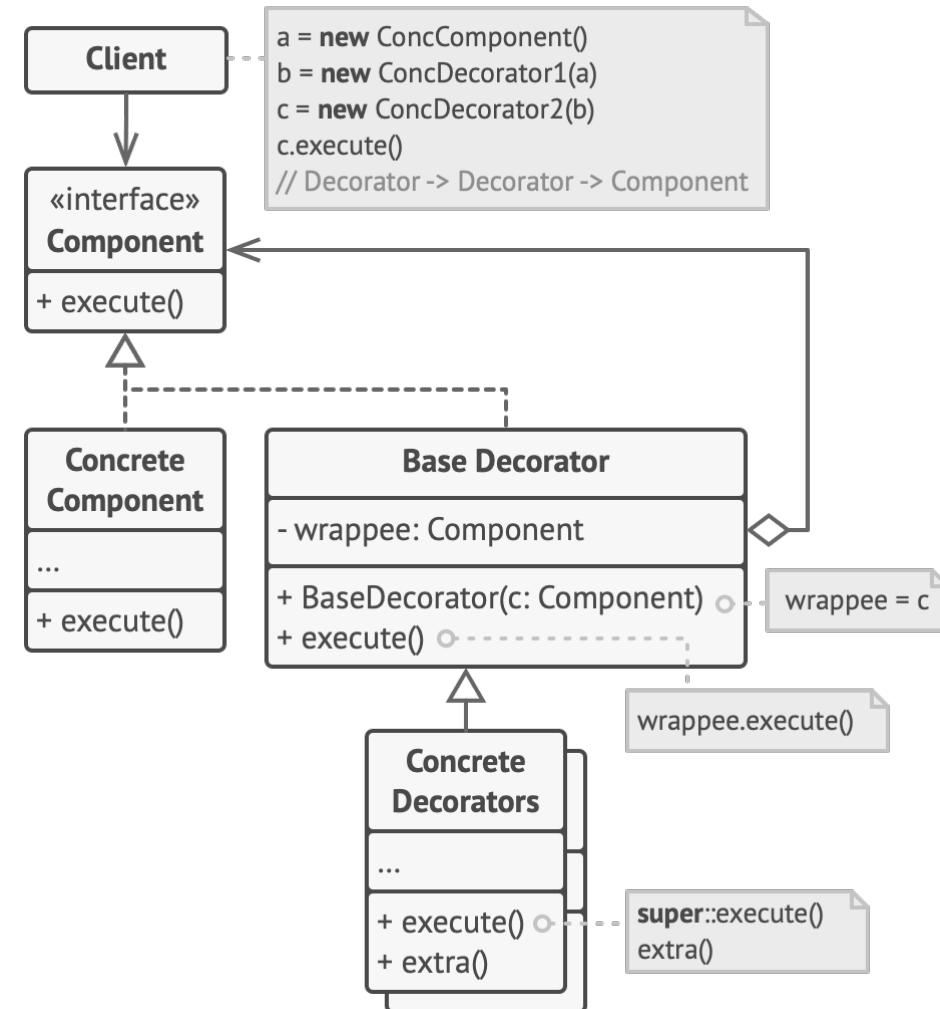
# our Starbuzz beverages

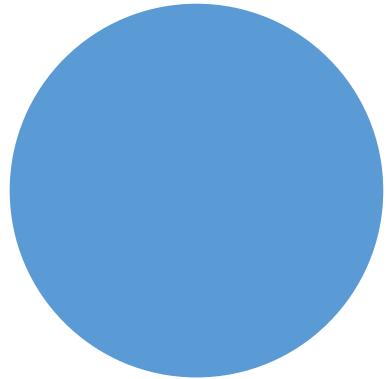


# our Starbuzz beverages



# Decorator





Software design &  
Quality Assurance

Testing

# Inleiding

---

# Inleiding testen



- Oefening
  - Wat is software?
  - Waarom testen?
  - Problemen bij testen
  - Hoe testen?
  - Fasen van het ontwikkelingsproces
  - Oorsprong van Fouten

Testen?

---

## Oefening

- We zouden graag een klein programmaatje testen dat drie integer waarden inleest. Deze waarden stellen de lengte van de zijden van een driehoek voor. Het programma zal ons vervolgens vertellen of deze driehoek ongelijkzijdig, gelijkbenig of gelijkzijdig is.

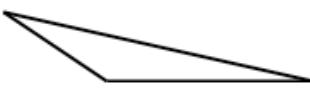
# Oefening



Gelijkzijdige  
driehoek



Gelijkbenige  
driehoek



ongelijkzijdige  
driehoek

- Schrijf de test cases op die u zou voorstellen om dit programma te testen.

## Oplossing

Heb je een test case die een geldige ongelijkzijdige driehoek voorstelt  
(merk op zijden 1, 2, 3 en 2, 5, 10 stellen geen driehoek voor)

Heb je een geldige gelijkbenige driehoek?

Heb je een geldige gelijkzijdige driehoek? (test cases a la 2, 2, 4 zijn geen geldige driehoeken!)

## Oplossing

Heb je minstens 3 test cases die geldige gelijkbenige driehoeken voorstellen waarbij je alle mogelijke combinaties vd permutaties vd zijden hebt uitgeprobeerd? Vb. 3,3,4; 3,4,3; en 4,3,3

Heb je een test case waarbij 1 zijde een 0 waarde heeft?

Heb je een test case waarbij 1 zijde een negatieve waarde heeft?

# Oplossing

Heb je een test case met 3 integers waarbij de som van twee van deze integers gelijk is aan de derde integer? Bv. 1, 2, 3

Heb je van voorgaand geval 3 test cases zodat je ook hier alle permutaties hebt uitgeprobeerd?

Heb je een test case met 3 integers waarbij de som van 2 vd integers kleiner is dan de derde integer?

Heb je van vorige geval ook alle 3 de permutaties opgegeven?

## Oplossing

Heb je een test case waarbij alle drie de zijden 0 zijn? Vb. 0, 0, 0

Heb je minstens 1 test case met niet-integer waardes?

Heb je minstens 1 test case die het verkeerde aantal integers bevat?

Heb je voor elke test case de verwachte uitvoer ook genoteerd?

# Wat gaan we testen en waarom?

---

## Wat is software?

### **Computer programma's, procedures, data en daarbij horende documentatie**

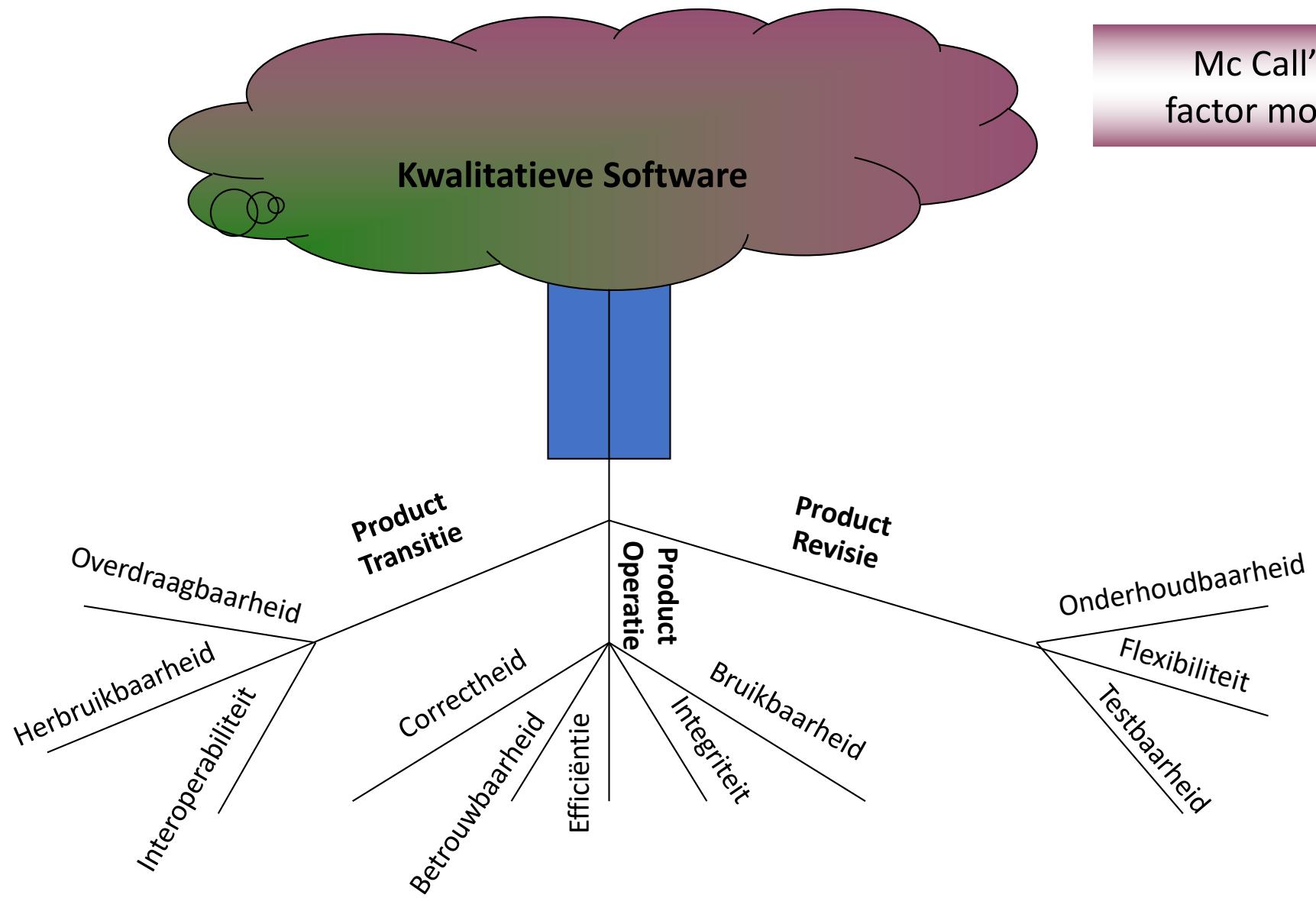
- Computer programma's
  - de code
- Procedures
  - geven volgorde van uitvoering aan
- Data
  - Parameters
  - DB
- Documentatie
  - voor gebruikers
  - voor andere programmeurs
  - ontwikkelingsdocumentatie

# Wat is kwalitatieve software?

**De kwaliteit van software is**

- **de mate waarin een systeem, component of proces bepaalde vereisten tegemoet komt**
- **de mate waarin een systeem, component of proces de verwachtingen van de klant of gebruiker tegemoet komt**

Mc Call's  
factor model



Product  
operatie

Correctheid

Betrouwbaarheid

Efficiëntie

Integriteit

Bruikbaarheid

# Product revisie

Onderhoudbaarheid

Flexibiliteit

Testbaarheid

Product  
transitie

Overdraagbaarheid

Herbruikbaarheid

Interoperabiliteit

# Waarom testen?

---

## Waarom testen?

- Foutloos systeem = mythe
- Vertaling vereisten – objectieven – externe specificaties – systeemontwerp – programmastructuur – specificatie modules – code = 6-tal vertaalstappen
- Programmeurs
  - 15 – 150 fouten / 100 lijnen
  - 80% – 99.3%

## Waarom testen

- Software die niet correct werkt kan leiden tot
  - Verlies van tijd, geld, reputatie
  - zelfs letsels of de dood

## Waarom testen?

- Contractuele of wettelijke verplichtingen
- Industrie standaarden naleven

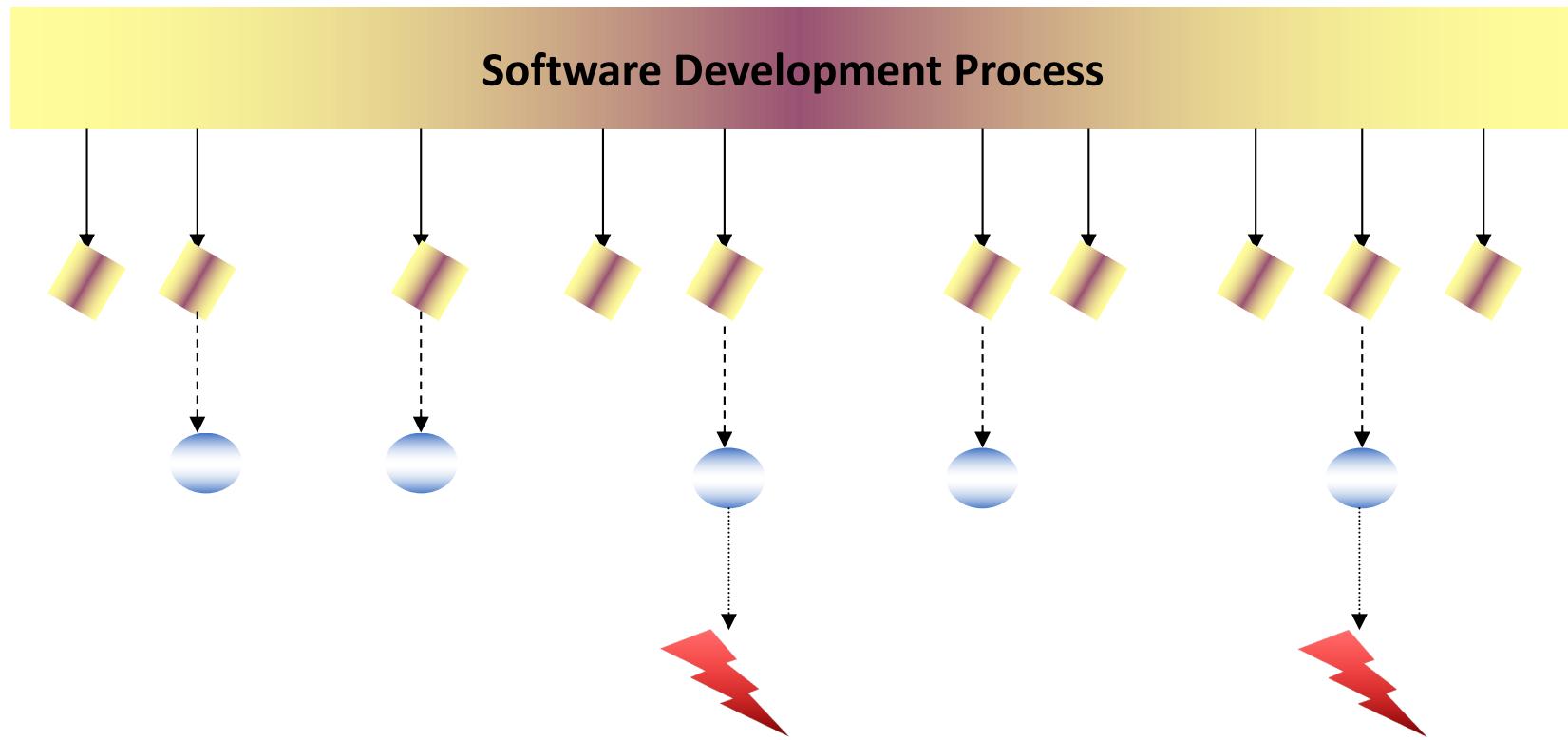
## Waarom testen?

- Wij gebruiken het Simplex Dev systeem reeds drie jaar en hebben nooit problemen ondervonden.
- 2 maanden geleden hebben we Simplex Dev geïnstalleerd en niets dan foutmeldingen en crashes gekend. We denken eraan het systeem door een andere applicatie te vervangen.
- We gebruiken hetzelfde softwarepakket nu bijna 4 jaar en zijn er enorm tevreden over. De laatste maanden doen er zich echter gereeld zware bugs voor. Het support team van Simplex Dev beweert echter nooit dit type fout te zijn tegen gekomen, hoewel ze 700 klanten bedienen.

## Waarom testen?

- Is het mogelijk dat deze commentaar driemaal over hetzelfde product wordt gegeven?
- software errors/mistake
- software faults/defect/bug
- software failures

# Waarom testen?



-- Software Error



-- Software Fault



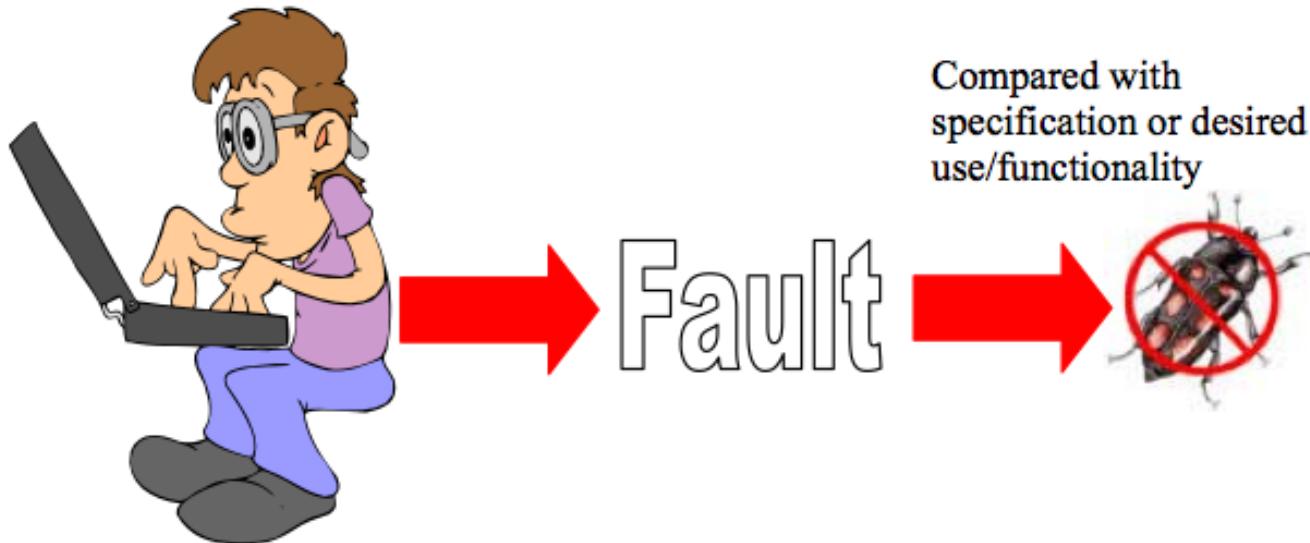
-- Software Failure

# Software error vs fault

- Een software error zal niet altijd een software failure tot gevolg hebben.
- bv.:
  - int a;
  - int b;
  - //waarden inlezen voor a en b
  - int c = a/b;

# Mistake, fault, failure

---



A programmer makes a **mistake**.

The mistake manifests itself as a **fault<sup>1</sup>** [or defect] in the program.

A **failure** is observed if the fault [or defect] is made visible. Other faults remain **latent** in the code until they are observed (if ever).

Wat zijn de objectieven van software tests?

Fouten vinden

De kwaliteit van het product inschatten

Informatie verschaffen om productbeslissingen te nemen

Voorkomen van fouten

# Waarom testen

- Bij voorbeeld
  - Managers helpen bij het aankondigen van release
  - Voortijdige release kunnen uitstellen
  - Kosten support helpen inschatten
  - Interoperabiliteit met andere producten nagaan
  - Veilige gebruiksscenario's achterhalen

# Problemen bij testen

---

# Problemen bij testen

```
int een_functie (int j)
{
 j = j - 1; // moet zijn j = j + 1
 j = j / 30000;
 return j;
}
```

# Problemen bij testen

```
int een_functie (int j)
{
 j = j - 1; // moet zijn j = j + 1
 j = j / 30000;
 return j;
}
```

| Input (j) | Verwachte resultaat | Eigenlijke resultaat |
|-----------|---------------------|----------------------|
| 1         | 0                   | 0                    |
| 42        | 0                   | 0                    |
| 40000     | 1                   | 1                    |
| -32000    | -1                  | -1                   |

# Problemen bij testen

```
int een_functie (int j)
{
 j = j - 1; // moet zijn j = j + 1
 j = j / 30000;
 return j;
}
```

| Input (j) | Verwachte resultaat | Eigenlijke resultaat |
|-----------|---------------------|----------------------|
| 29999     | 1                   | 0                    |
| 30000     | 1                   | 0                    |
| -30000    | 0                   | 1                    |
| -29999    | 0                   | 1                    |

# Problemen bij testen

Onvoldoende middelen (geld, mensen, tijd, ...)

Te groot aantal combinaties

Output achterhalen

Wijzigende/onbestaande vereisten

Onvoldoende opleiding

Geen (geschikte) kennis van tools

Alles testen? → onmogelijk

Management

# Problemen bij testen

- Meesten denken aan tevredenheid van de klant
  - Met welke features is een klant tevreden?
  - Wat verwacht hij van dit product?
    - ➔ de juiste features
    - ➔ goede instructies

# Problemen bij testen

- Software testers denken aan ontevredenheid van de klant
  - Wat zorgt ervoor dat de klant ontevreden is met het product?
    - ➔ onbetrouwbaarheid van het product
    - ➔ moeilijk in gebruik
    - ➔ te traag
    - ➔ incompatibel met hardware van de klant

# Hoe testen

---

# Hoe testen?

## Definities van softwaretesten:

- ‘aantonen dat er geen fouten meer aanwezig zijn in het programma’
- ‘aantonen dat het programma zijn functies correct uitvoert’
- ‘het proces waarbij u uzelf verzekert van het feit dat het programma doet wat het moet doen’

## Constructief vs Destructief

# Constructief versus Destructief

- ‘aantonen dat er geen fouten meer aanwezig zijn in het programma’
- ‘aantonen dat het programma zijn functies correct uitvoert’
- ‘het proces waarbij u uzelf verzekert van het feit dat het programma doet wat het moet doen’

## Versus

- ‘het uitvoeren van het programma met de bedoeling er fouten in te vinden’

## Hoe testen?

- Afhankelijk van de te testen software
  - Bv. Programma testen dat berekeningen uitvoert
    - Als onderdeel van computer game voor een vliegsimulator
    - Als onderdeel van de vluchtsoftware van een Boeing 747
  
- Hoe gedreven ga je bugs opsporen
- Welke bugs zijn minder belangrijk
- Hoe intensief zal je je werk documenteren

# Oorsprong van fouten

# Oorsprong van Fouten

Vereisten

Communicatie klant – ontwikkelaar

Opzettelijke afwijkingen van de vereisten

Logische ontwerpfouten

Codeerfouten

Codeerinstructies

Testfouten

# Vereisten

Verkeerde definitie van  
de vereisten

Afwezigheid van vitale  
vereisten

Onvolledige definitie van  
de vereisten

Toevoegen van niet-  
gevraagde vereisten

Communicatie  
klant –  
ontwikkelaar

Verkeerdelijk verstaan  
van de vereisten zoals  
opgesteld door de  
klant

Verkeerdelijk verstaan  
van het antwoord van  
de klant op  
ontwerpvragen

## Opzettelijke afwijkingen van de vereisten

Hergebruik van bestaande modules zonder na te gaan of deze aanpassingen vereisen

Het weglaten van bepaalde functionaliteiten omwille van tijdsdruk

Het toevoegen van verbeteringen zonder goedkeuring van de klant

## Logische ontwerpfouten

Niet juist functionerende  
algoritmen

Definities van processen die fouten  
bevatten qua volgtijdelijkheid

Verkeerde definitie van  
grenswaarden

Het weglaten van reacties op  
illegale operaties/data

Codeerfouten

Verkeerdelijk  
verstaan van de  
ontwerpdocumenten

Taalkundige fouten  
tegen de  
programmeertaal

## Codeerinstructies

Niet overeenstemmen met de huisregels qua codeerstijl en naamgeving, maakt het lastig voor:

nieuwe programmeurs die bestaande code moeten verstaan

bestaande programmeurs die je code moeten uitbreiden

bestaande programmeurs die je code moeten nakijken

software testers die aan de hand van je code test cases moeten opstellen of er fouten moeten uithalen

## Testfouten

Onvolledige  
testplannen, onvolledig  
uitvoeren van delen  
van het testplan

Incorrect rapporteren

# 7 Principles of Software Testing

---

Tests tonen  
de  
aanwezigheid  
van fouten

Testing bewijst de aanwezigheid  
van fouten

Voldoende testen vermindert de  
kans dat er nog niet ontdekte  
fouten aanwezig zijn

Testen bewijst niet dat er geen  
fouten meer aanwezig zijn, ook al  
worden ze niet gevonden

Tests bewijzen nooit dat een  
systeem vrij is van fouten

Exhaustief  
testen is niet  
mogelijkheid

Een exhaustieve test die alle mogelijke inputparameters, alle combinaties en alle mogelijke pre-condities voorziet, bestaat niet (behalve voor triviale test objecten).

Bijgevolg is testen altijd een combinatie van risico beheersen, prioriteiten stellen en keuzes maken.

Test vroeg en  
regelmatig

De testactiviteiten dienen zo vroeg mogelijk opgestart te worden in het software ontwikkelingsproces en dienen regelmatig herhaald te worden.

Hoe vroeger een fout opgespoord is hoe eenvoudiger en goedkoper ze op te lossen.

## Accumulatie van fouten

Er is geen gelijkmatige verdeling van fouten binnen een testobject. Op de plaats waar men één fout terugvindt, is het waarschijnlijk dat er meerdere aanwezig zijn.

Het testproces moet daarom flexibel zijn en hierop kunnen anticiperen.

## Afnemende effectiviteit

De effectiviteit van testen nemen af na een tijd. Indien testcases enkel herhaald worden, zullen ze geen nieuwe fouten blootleggen.

Fouten die in niet geteste functies achterblijven worden op deze manier niet ontdekt.

Testcases dienen dus regelmatig aangepast worden.

Testen zijn  
contextgevoelig

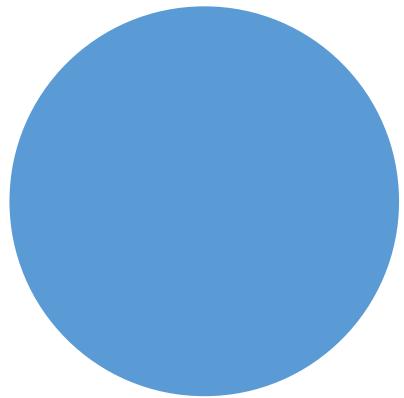
Geen twee systemen zijn dezelfde en kunnen bijgevolg op dezelfde manier getest worden.

De intensiteit van de tests, de definitie van de eindcriteria, etc moeten voor ieder systeem bepaald worden, aangepast aan de context.

Bijvoorbeeld: E-commerce website vs online-banking app

# Absence- of-errors fallacy

- Het vinden en oplossen van defecten helpt niet als het systeem niet bruikbaar is en/of niet voldoet aan de noden en verwachtingen van de eindgebruikers.



# Software design & Quality Assurance

Johan van den Broek

# Software testing



- "*Quality is never an accident; it is always the result of intelligent effort.*" -  
- [John Ruskin](#)

# Wanneer testen

---

# Testen doorheen de Software Life Cycle

Testen gebeurt niet in isolatie.

Test activiteiten zijn gerelateerd tot software development activiteiten.

Ieder softwareontwikkelingsmodel heeft nood aan zijn eigen benadering.

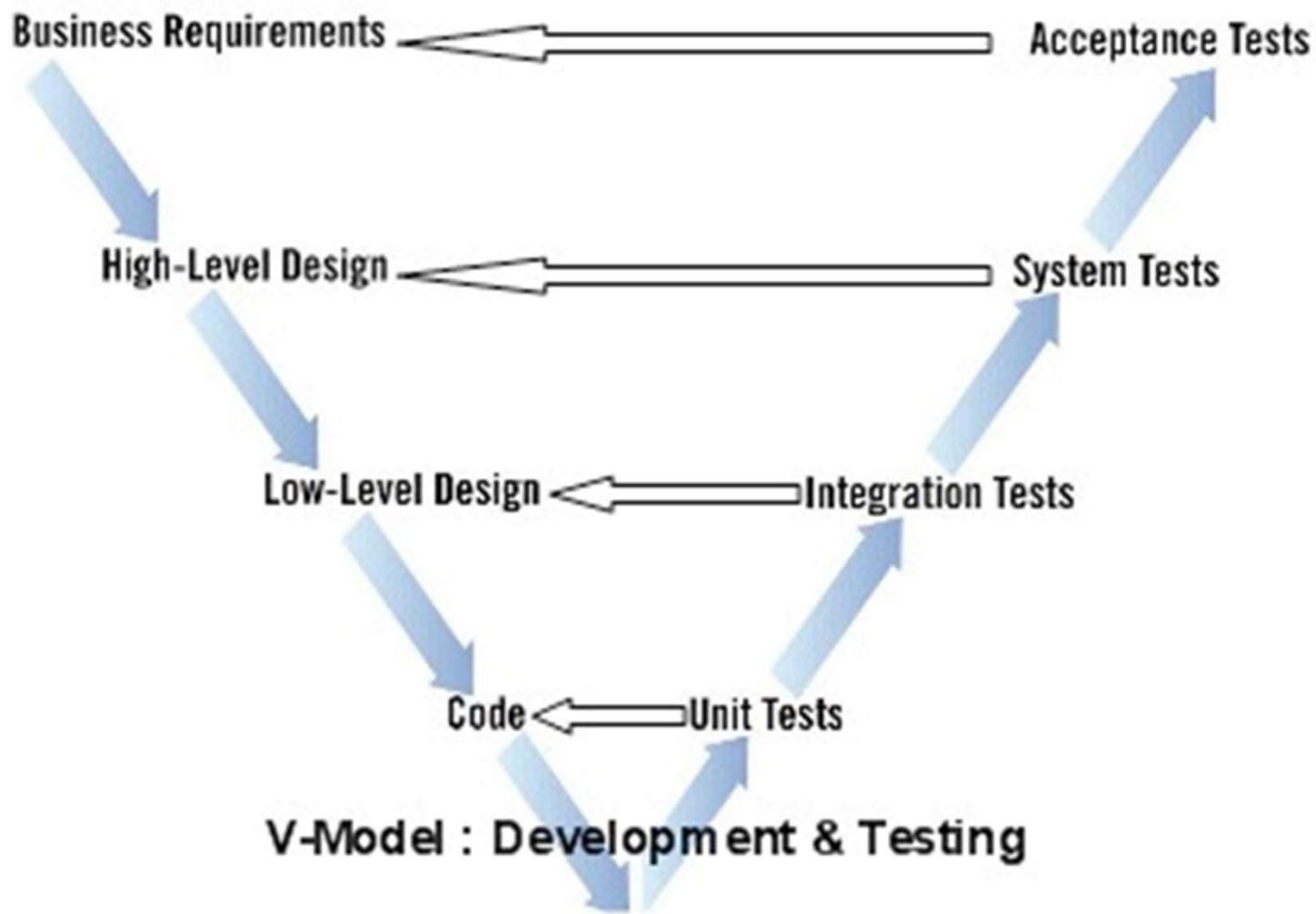
# V-model

Het V-model wordt beschouwd als een uitbreiding van het waterval model.

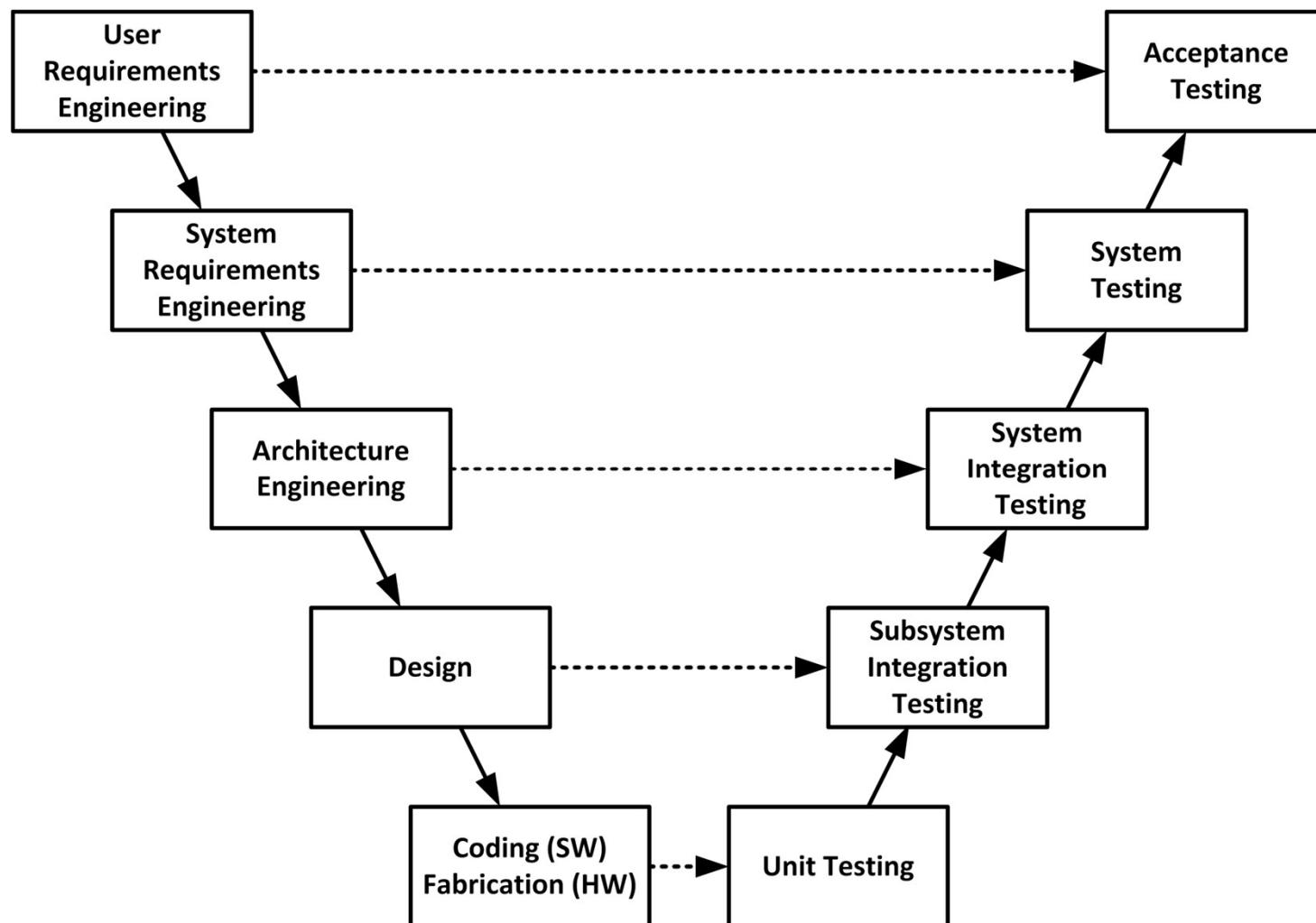
Is een term die verwijst naar een hele reeks van softwareontwikkelingsmodellen

Het is een conceptueel model dat werd ontwikkeld om op een eenvoudige wijze de complexiteit van het softwareontwikkeling weer te geven.

# V-model (v1)



# V-model (v2)



# Testlevels

---

## Testlevels

Testen worden vaak gegroepeerd op basis van de plaats waar zij worden uitgevoerd in het softwareontwikkelingsproces.

Testlevels zijn groepen van testactiviteiten die samen beheerd worden.

## Testlevels

- Testlevel is niet hetzelfde als een testfase. In een incrementeel of iteratief ontwikkelingsmodel zal men bijvoorbeeld in verschillende fasen van een project telkens opnieuw taken uitvoeren die behoren tot een welbepaald testlevel.

4 Testlevels

Module testing

Integratie testing

System testing

Acceptance testing

# Terminologie: testbasis

---

Een belangrijk aspect bij testen is de vraag: wat is correct?

---

Bij computerprogramma's kan dit zijn vastgelegd in één of meerdere documenten, waaronder een technisch of functioneel ontwerp, een interactie-ontwerp of ontwerp van de grafische gebruikersinterface.

---

Deze documenten vormen samen de testbasis, het geheel van producten waarop de verwachtingen, oftewel het testontwerp, wordt gebaseerd.

## Terminologie: testobject

- Belangrijk bij het testen van software is weten wat er getest gaat worden: het testobject

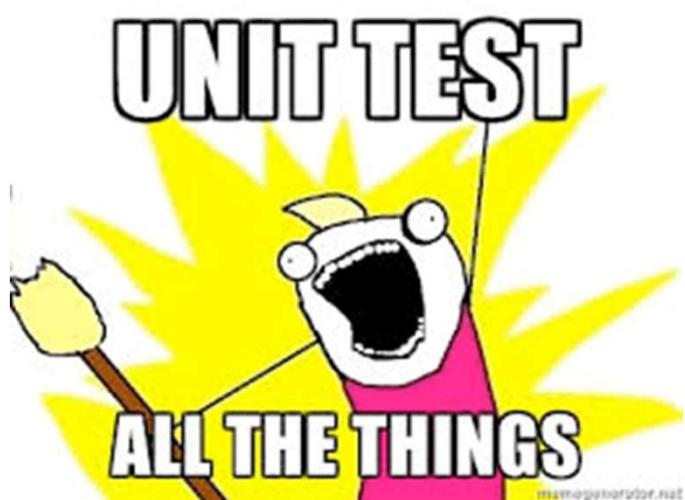
## Voor we starten ...

- Reviews van vereisten vormen een uitstekend middel om vroeg en kost-efficiënt fouten te detecteren.

# Module testing

---

# Module testing



- Ook bekend als component, unit of program testing
- Unitests worden gebruikt om zeker te zijn dat de individuele componenten correct werken.

# Module testing

## Testbasis:

- Requirements van het component
- Details van het design
- Code

## Testobjecten:

- Componenten
- Programma

## Wat gaan we testen?

Functionaliteit of niet functionele eigenschappen zoals hoe er met resources wordt omgesprongen (memory leaks)

Correct functioneren van de module

Structurele testen (bv decision coverage)

Testcases  
worden  
afgeleid van

De specificaties  
van de component

Software design

Data model

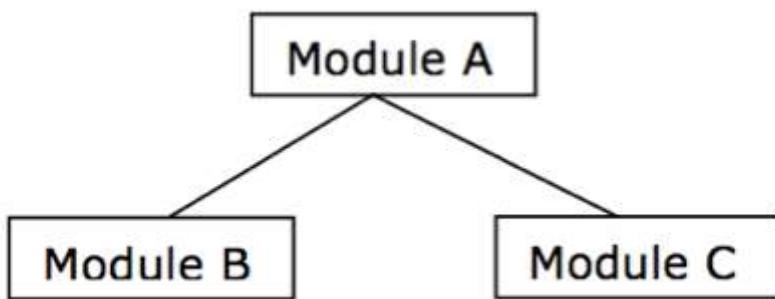
## Module testing

- Normaliter heeft de tester toegang tot de code en voert hij testen uit met behulp van een unit test framework.
- Een mogelijkheid: Test-driven development

# Module testing

- Unitests zijn onafhankelijk van elkaar en moeten dan ook allen afzonderlijk kunnen uitgevoerd worden.

# Probleem?



- Modules B en C worden aangeroepen door A

# Integratietesten

---

## Integratietesten

- Integratietesten testen interfaces tussen componenten en interacties met andere onderdelen van een systeem zoals het OS, bestandssysteem en hardware en interfaces tussen systemen.

# Integratietesten

## Testbasis:

- Software en systeemontwerp
- Architectuur
- Workflows
- Use cases

## Testobjecten:

- Subsystemen
- Infrastructuur
- Interfaces
- Systeemconfiguratie en configuratiedata

## Integratietesten: 2 vormen

### Component integratietesten

- Testen na de integratie van verschillende software componenten met als doel de interactie tussen de componenten te testen. Uitgevoerd na de module testen.

### Systeem integratietesten

- Testen de interacties tussen verschillende systemen of tussen hardware en software en kan uitgevoerd worden na de systeemtests.

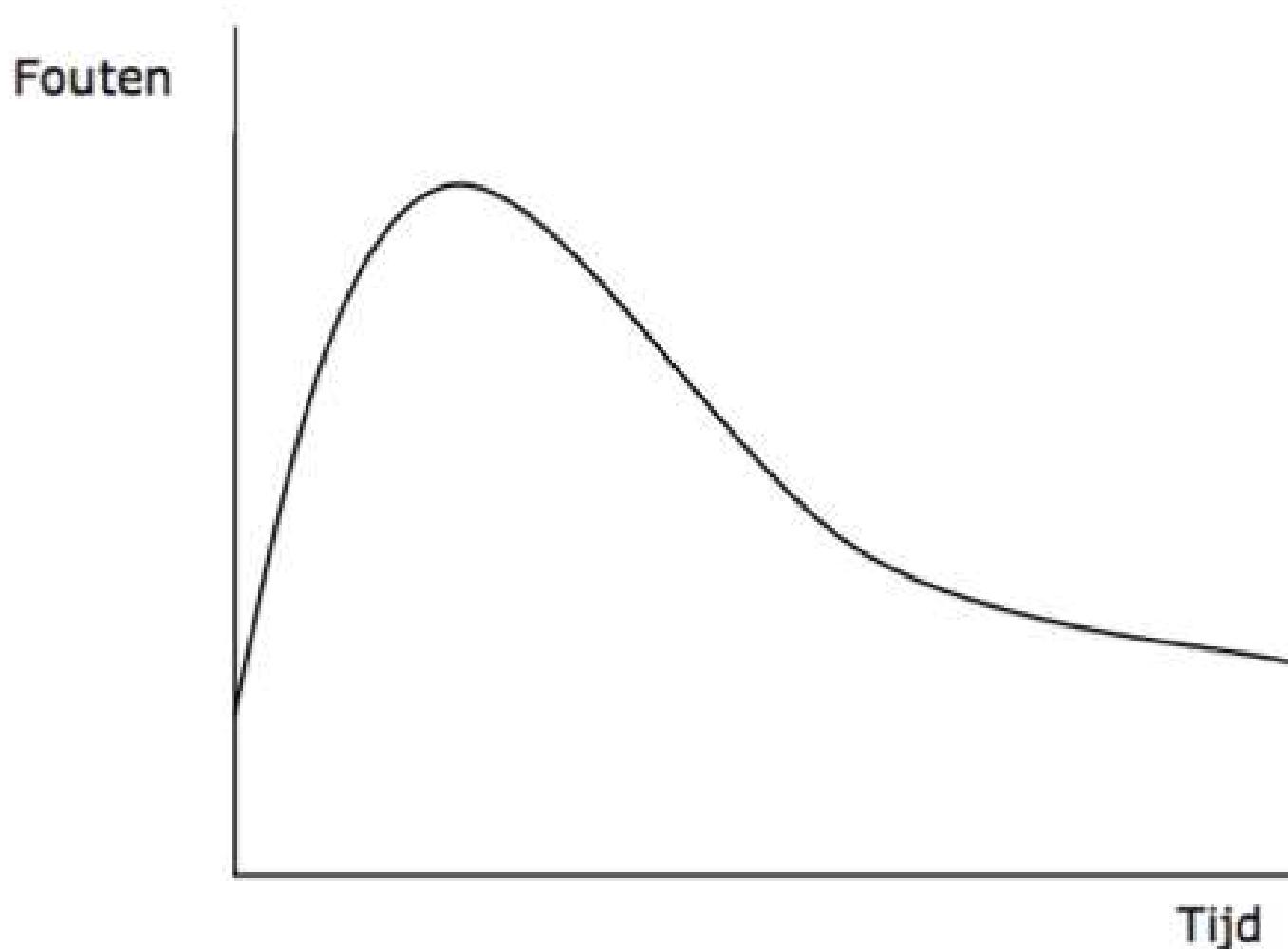
## Integratietesten

Zowel niet functionele eigenschappen als functionele eigenschappen kunnen getest worden (bv performance)

Testers moeten zich concentreren op de communicatie tussen de verschillende componenten niet op de functionaliteit van iedere individuele component

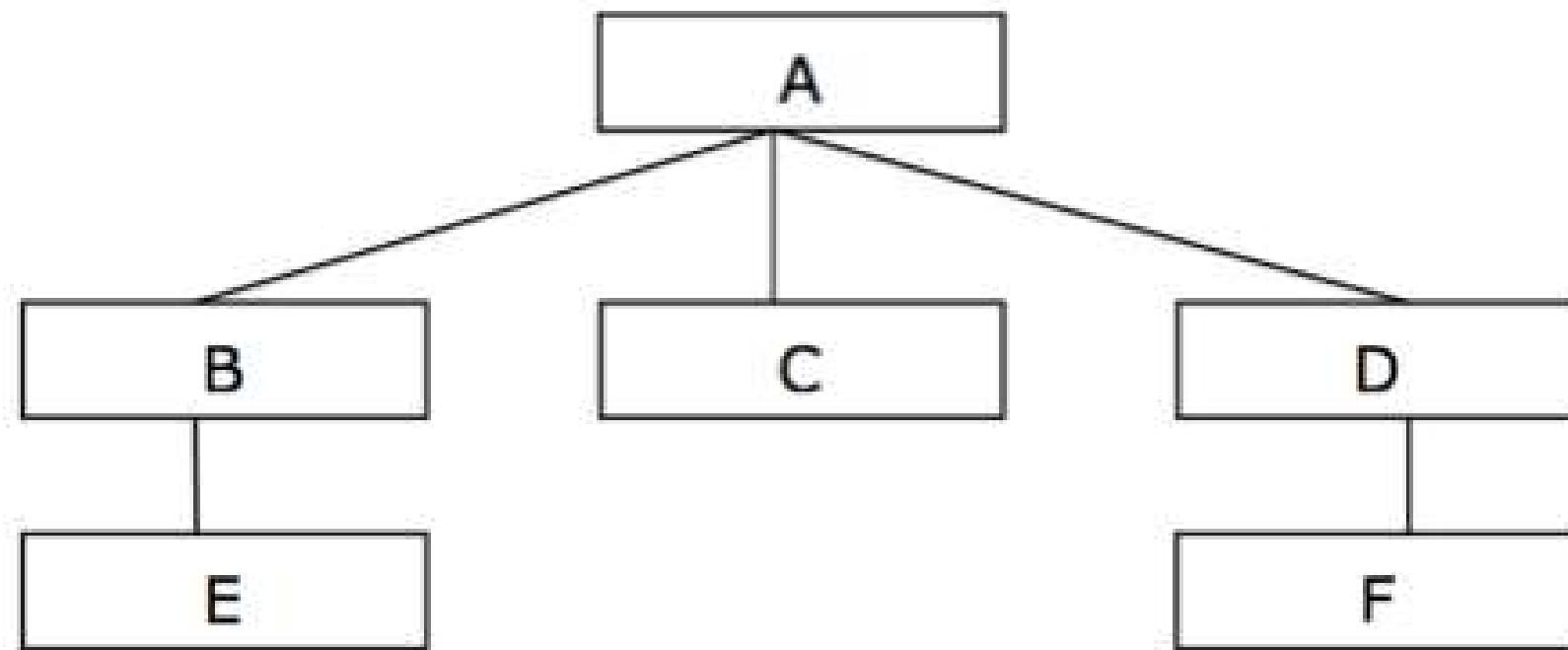
# Niet-incrementeel testen

Unit tests en nadien samenvoegen.



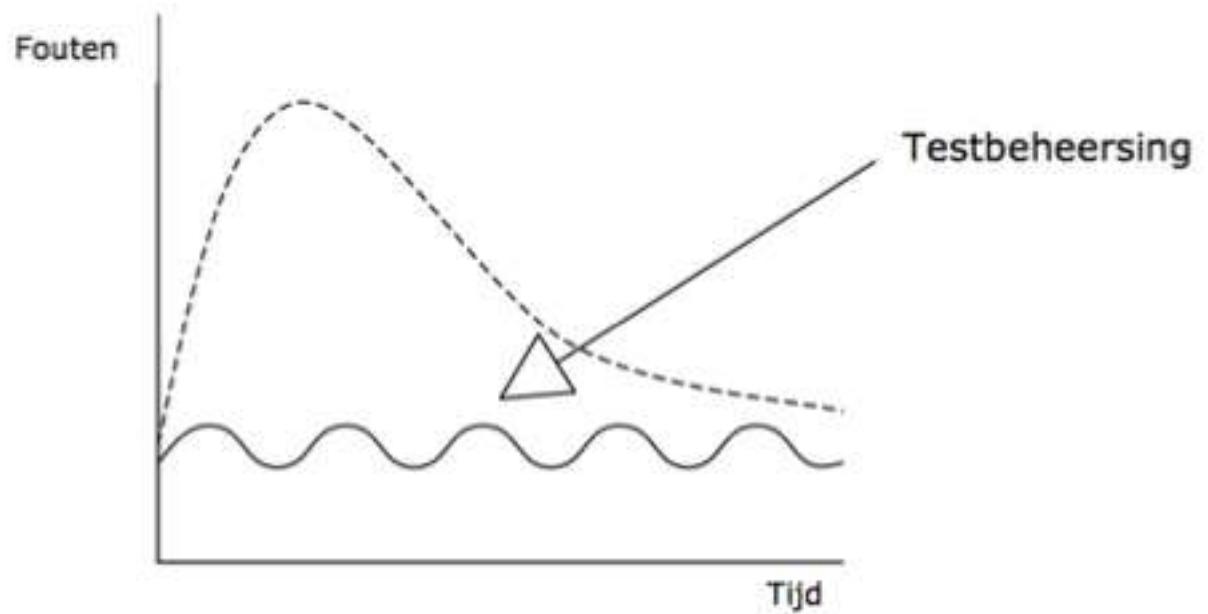
# Niet-incrementeel testen

Nadeel: Stubs en drivers voor iedere module



# Incrementeel testen

- De testen module samenvoegen met reeds geteste modules



# Incrementeel testen

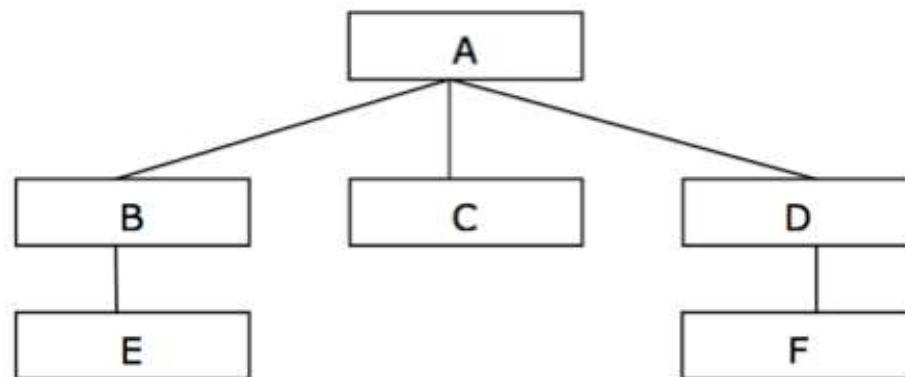
## Top-down

- Boven gelegen modules (bv user interface)
- Wanneer?
  - Complexiteit aan de ‘bovenkant’

# Incrementeel testen

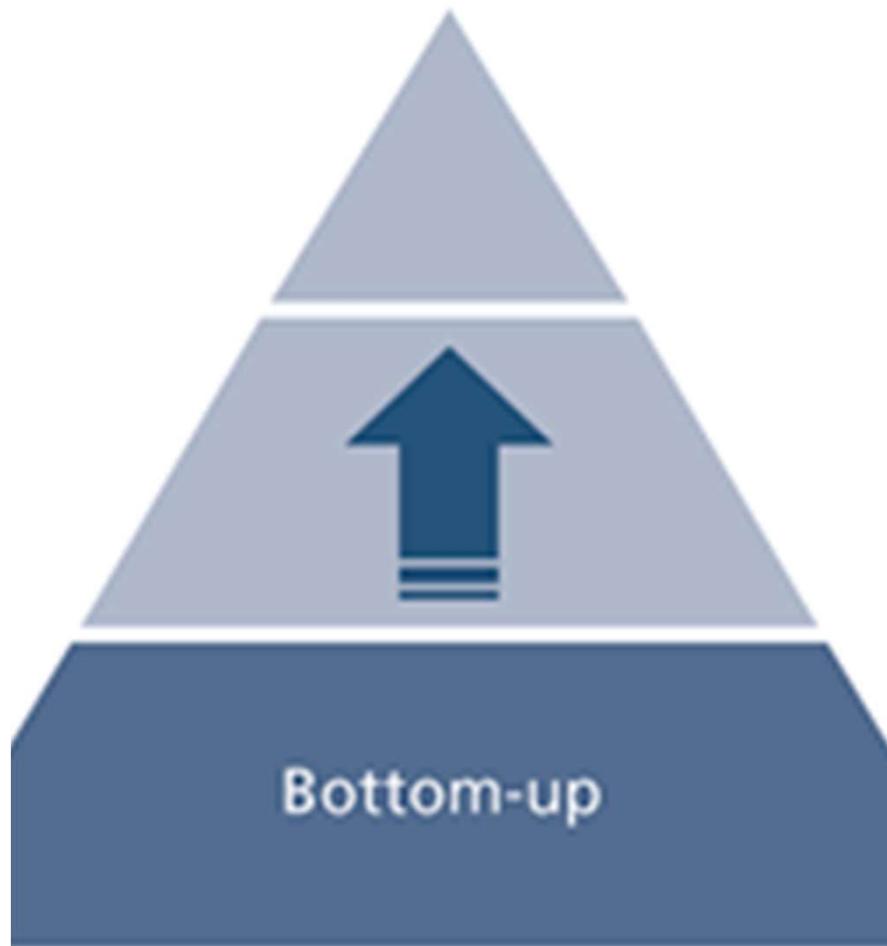
## Top-down

- Voordeel
  - Relatief korte termijn kan men een werkende versie voorstellen aan de klant
- Nadeel
  - Welke data wordt doorgegeven? Welke transformatie? Hoe testen?
- Meeste kritieke delen worden het uitvoerigst getest!



# Incrementeel testen

## Bottom-up



- Starten met de onderste modules
- Wanneer de belangrijkste functionaliteit zich in deze modules bevindt

# Incrementeel testen

## Bottom-up

- Voordeel
  - Je dient enkel drivers te ontwikkelen (eenvoudiger)
  - Testcases eenvoudiger dan met top-down (testdata!)
- Nadeel
  - Men kan geen programma opleveren aan de eindgebruiker totdat alles getest is
- Meest kritieke delen worden het uitvoerigst getest

# Systeemtests

---

## Systeemtests

Bij de systeemtests willen we het gedrag van een volledig systeem, zoals gedefinieerd in de vereisten, testen.

Voor systeemtests is geen kennis vereist van het inwendige design of logica van het systeem

# Systeemtests

- Bij de systeemtests moet de omgeving zoveel mogelijk overeenkomen met de finale productieomgeving om het risico op omgeving specifieke defecten te minimaliseren.

# Systeemtests

## Testbasis:

- Requirementspecificaties
- Use cases
- Functionele specificatie
- Risicospecificaties

## Testobjecten:

- System, user and operation manuals
- System configuration and configuration data

# Systeemtests

- Doelstelling van de testuitvoering is dat de leverancier overtuigd is dat het systeem voldoet aan alle gestelde vereisten en klaar is om aan de klant te worden overgedragen.

# Systeemtests

Vele soorten tests ...

## Systeemtest - Facility testing

Alle objectieven één voor één overlopen en na te gaan of het programma voldoet.

In bepaalde gevallen is het mogelijk deze test uit te voeren door de objectieven te vergelijken met de gebruikersdocumentatie.

## Systeemtest – Volume testing

Deze testcases zullen bepalen of het programma grote hoeveelheden data aankan.

Het doel van deze tests is aantonen dat het programma de hoeveelheden data die in de objectieven beschreven staan, aankan.

## Systeemtest – Stress testing

grote hoeveelheden  
data aan het  
programma overmaken  
in een korte periode.

Er zal dus nagegaan  
worden of het  
programma hoge  
pieken aankan.

# Systeemtest – Usability testing

- Deze tests zullen de gebruiksvriendelijkheid van de interface van het programma nagaan.



Developer watching videotape of usability test.

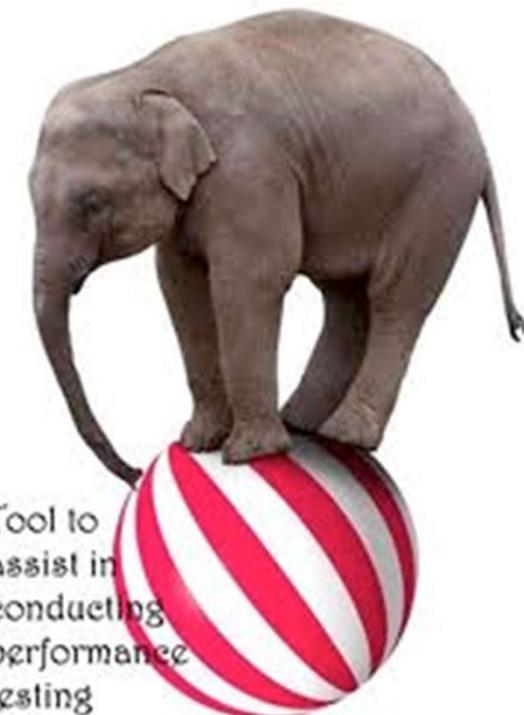
# Systeemtest - Security testing

- Bij security testing zal je trachten de beveiliging van het systeem te omzeilen.



# Systeemtest – Performance test

- Indien er in de objectieven zaken werden vermeld over de performantie en de efficiëntie van het programma, dienen deze zaken ook afgetoetst te worden.



Tool to  
assist in  
conducting  
performance  
testing

# Systeemtest – Storage testing

- Objectieven bevatten vaak ook clauses over de eigenschappen van de opslag van een programma, zoals het totale geheugen dat door het programma mag worden gebruikt. Indien dit zo is, moet er worden nagegaan het programma hieraan beantwoordt.



# Systeemtest – Configuration testing

Het programma test met elk mogelijk (ondersteund) type hardware en dit telkens met een minimale en maximale configuratie.

Als het programma zelf kan worden geconfigureerd, dien je ook elke mogelijke configuratie van het programma te testen.

# Systeemtests: Compatibility/Conversiontesting

- Programma's zijn vaak een vervanging voor een oudere versie. Je moet in dat geval ook rekening houden met de objectieven van deze oudere versie.

## Systeemtests: Reliability testing

- Controleren van objectieven aangaande betrouwbaarheid.
- Recoverability testing
  - Vaak zijn er objectieven aanwezig die het herstel van programmafouten of het falen van hardware omschrijven.



# Systeemtests: Serviceability testing

- Deze objectieven bevatten de hulpprogramma's die bij een systeem geleverd worden. Ze omvatten eveneens de procedures voor onderhoud.

# Systeemtests: Documentation testing

- Een systeemtest omvat eveneens het testen van de gebruikersdocumentatie.



# Acceptatietests

---

## Acceptatietests

Acceptatietests zijn vaak de verantwoordelijkheid van de klanten of (eind)gebruikers van een systeem. Andere stakeholders kunnen hier ook bij betrokken worden.

Het doel van acceptatietesten is om vertrouwen in het systeem, delen van het systeem of specifieke niet functionele eigenschappen te krijgen.

Het vinden van defecten is niet het hoofddoel van acceptatietesten. Met de acceptatietests kunnen we de gereedheid voor gebruik van een systeem nagaan.

# Acceptatietests

- Er kunnen verschillende vormen van acceptatietests voorkomen. Bijvoorbeeld:
  - Acceptatietesten van de bruikbaarheid van een component kan gebeuren tijdens het unittesten.
  - Acceptatietesten van een nieuwe functionele verbetering kan gebeuren voor de systeemtests.



# Elementen van acceptatietests



## Gebruikersacceptatietests

de paraatheid verifiëren van het gebruik van een systeem door business gebruikers.



## Operationele acceptatietests

de acceptatie van het systeem door de systeem administrators, bevat tests van backup/restore, disaster recovery, user management, maintenance tasks, etc.

# Elementen van acceptatietests

- **Contract- en richtlijntests (conformitetest)**
  - de acceptatiecriteria moeten afgesproken worden tijdens de contractbesprekingen. De acceptatie wordt dan gedaan tegenover deze criteria voor het produceren van software. Hierin zijn ook standaarden (government, legal, security, etc.) inbegrepen.

# Elementen van acceptatietests

- Zonder specifieke afspraken moet het systeem voldoen aan alle gestelde vereisten zoals beschreven in het lastenboek en bijhorende documenten evenals aan de vereisten beschreven in alle meer technische documenten.

# Elementen van acceptatietests

- **Alfa- en betatests**

- het krijgen van feedback van potentiele of bestaande klanten vooraleer het product op de markt komt. Alfa testen gebeurt binnen de organisatie van de ontwikkelaar.
- Betatesten (field testen) gebeurt door de mensen op hun eigen locatie.

# Test types

---

# Functionele tests

Ze beschrijven “wat” het systeem doet.

Functionele tests verifiëren de functies die een systeem, subsysteem of component moeten uitvoeren.

Functionele tests kunnen uitgevoerd worden op elk testlevel.

Niet  
functionele  
tests

Ze testen “hoe” het  
systeem werkt.

Niet functionele  
tests kunnen  
uitgevoerd worden  
op elk test level.

# Niet functionele tests

- Niet functionele tests omvatten performance tests, load tests, stresstests, usability tests, maintainability tests, reliability tests, etc.

# Tests gelinkt aan wijzigingen

- Confirmatietesten
  - Wanneer een defect gevonden en opgelost is moet de software opnieuw getest worden om te bevestigen dat het oorspronkelijke defect succesvol verwijderd is. Men spreekt in dit verband ook van hertesten.
  - Opmerking: Het debuggen van code is een ontwikkelingsactiviteit, geen testactiviteit.

# Tests gelinkt aan wijzigingen

Regression:  
"when you fix one bug, you  
introduce several newer bugs."

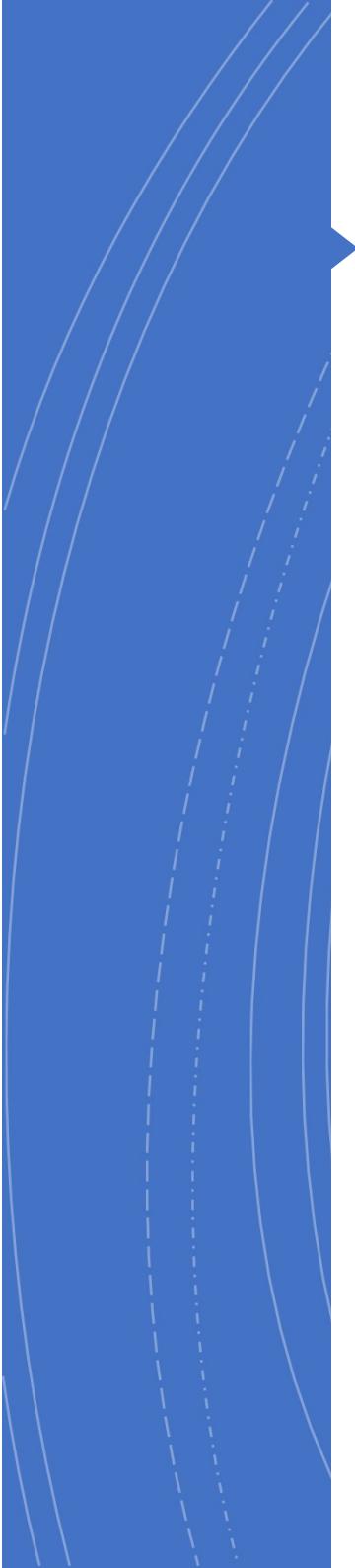


# Tests gelinkt aan wijzigingen



- Regressietesten
  - het opnieuw testen van een reeds geteste programmacode na wijzigingen om zo defecten te ontdekken (in niet gewijzigde aspecten van de software) die veroorzaakt worden door wijzigingen aan de software of aan de omgeving.

# CLASSIFICATIE VAN TESTTECHNIEKEN



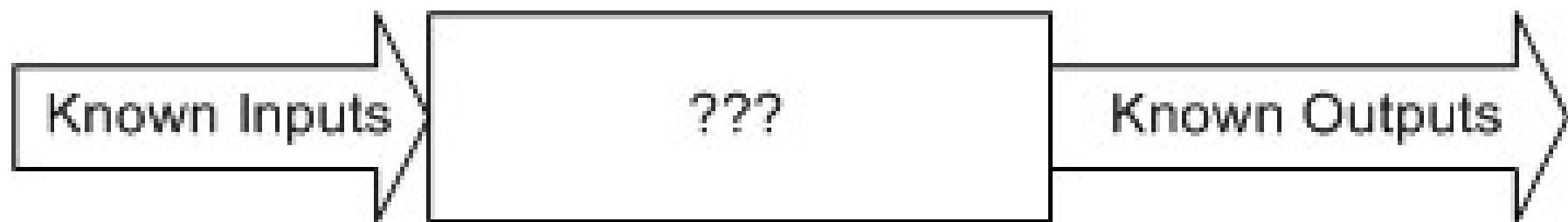
# Classificatie van testtechnieken

- Testtechnieken zijn formele werkwijzen om test gevallen af te leiden van documenten, van modellen van de software of van code.
- We kunnen de verschillende soorten testtechnieken opdelen in 3 groepen:
  - Whiteboxtests (structuurgebaseerd)
  - Blackboxtests (specificatiegebaseerd)
  - Ervaringsgebaseerde tests

# Blackboxtests

---

# Blackboxtests



Black Box Test is volledig gebaseerd op de functionele specificatie en kwaliteitseisen

# Blackboxtests

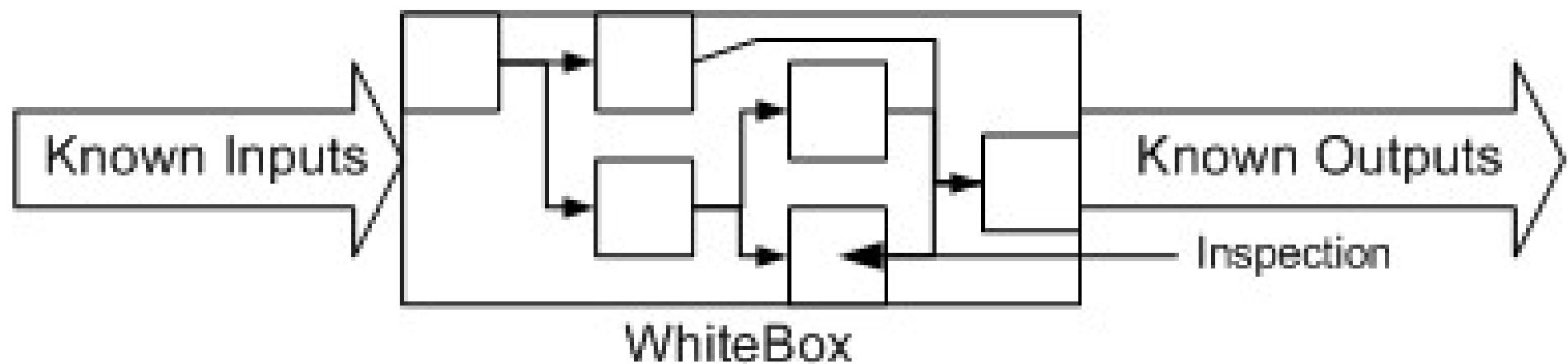


- De software wordt gezien als een “zwarte doos” zonder enige kennis van de interne implementatie of de interne structuur
- Kunnen op ieder testlevel gebruikt worden (typisch op hogere levels)

# Whiteboxtests

---

# Whiteboxtests



Bij whiteboxtesting gaan we kijken hoe de code opgebouwd is en aan de hand hiervan testcases opstellen.

## Whiteboxtests

Ze zijn gebaseerd op programmacode, programmabeschrijvingen of het technisch ontwerp.

Er wordt nadrukkelijk gebruik gemaakt van kennis over de interne structuur van het systeem. Deze tests worden typisch uitgevoerd door de ontwikkelaars.

Kunnen op alle level gebruikt worden (typisch op lagere testlevels)

# Whiteboxtests

- Code coverage tools: gebruikt om na te gaan hoeveel vd code effectief uitgevoerd wordt.
  - Statement coverage
    - Ieder statement moet door de testcases minstens éénmaal worden uitgevoerd
  - Decision coverage
    - Statistieken gaan na hoeveel beslissingen van een ja/nee beslissing door de tests uitgevoerd worden
  - Path coverage
    - Metingen gaan na hoeveel paden er door de tests uitgevoerd worden
    - Ieder pad wordt onafhankelijk van elkaar bekijken

## Whiteboxtests

White box testing wordt niet gegegaan of er werd afgeweken van de specificatie (elementen werden toegevoegd/ontbreken?).

White box = enkel de eigenschappen die de code ‘heeft’, niet de eigenschappen die de code ‘zou moeten hebben’.

# Classificatie van testtechnieken

- Bij BB testing mag de code een ongelooflijk rommeltje zijn, zolang ze maar doet wat ze zou moeten doen.
- Bij BB is het mogelijk dat er delen code nooit uitgevoerd zullen worden → Je bent nooit zeker hoeveel van het te testen systeem er nu daadwerkelijk getest is geweest.

# Ervaringsgebaseerde tests

---

## Ervaringsgebaseerde tests

tests opgebouwd op basis van de competenties, intuïtie en ervaring van de tester

Effectiviteit afhankelijk van de ervaring van de test

Bv Error guessing, Exploratory testing

# Testcases

---

# Testcases

- Wanneer je een testcase ontwerpt, moet je de volgende zaken identificeren:
  - Doelstelling van de test
  - Precondities voor de uitvoer van de test
  - Test data requirements (invoerwaarden voor de testcase maar ook data die eventueel noodzakelijk is in het systeem)
  - Verwachte resultaten
  - Postcondities na de uitvoer van de test (data aangepast? Staat van het systeem?)

# Verwachte resultaten

- Test oracle
  - softwarepakket, proces of data die informatie verschaft aan de testcase ontwikkelaar over de verwachte uitvoer van een testcase

# Verwachte resultaten

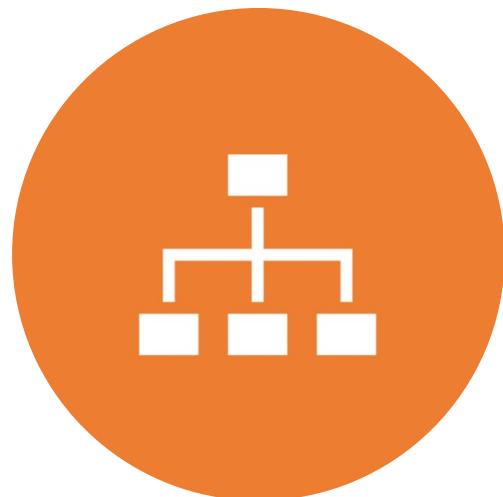
- Kiddie oracle
  - Voer tests uit, kijk naar het resultaat
- Regression oracle
  - Vergelijk de uitvoer met de uitvoer van eerdere versies vh programma
- Validated data oracle
  - Vergelijk de resultaten met een standaard zoals een tabel/formule of andere gestandaardiseerde modellen

# Outputs achterhalen

- Purchased test suite oracle
  - Vergelijk resultaten met die van een aangekocht goedgekeurd en gevalideerde test suite. Bijvoorbeeld: compilers, SQL processors en web browsers.

# Volgorde van de testcases

---



**CASCADING**



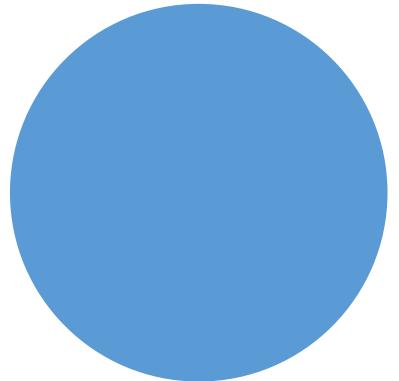
**INDEPENDANT**

# Cascading

- = de eerste testcase test een bepaalde functionaliteit en laat het programma in een dergelijke staat zodat de tweede test case uitgevoerd kan worden
  - Vb. DB: Create a record, Read the record, Update the record, Read the record, Delete the record, Read the deleted record
- Voordeel: elke testcase kan klein gehouden worden (opdeling testcases in stappen)
- Nadeel: Indien één test case faalt, zullen er problemen zijn voor de volgende test cases

# Independant

- = Elke test case staat compleet op zich
- Voordeel = De test cases kunnen in elke mogelijke volgorde uitgevoerd worden
- Nadeel = Elke test case zal groter en complexer zijn en bijgevolg moeilijker om te ontwerpen/implementeren/onderhouden.



# Software design & Quality Assurance

Johan van den Broek

# Blackbox testontwerptechnieken

---

# Equivalence Partitioning

---

# Equivalence Partitioning



- Probleem
  - Invoerveld: 1 tot 500
  - alle mogelijke invoerwaarden testen?
- Oplossing
  - invoerwaardes indelen in equivalentie klassen
- Doel
  - aantal testcases verminderen zonder verlies van test coverage

# Equivalence Partitioning

- Equivalentie klasse = verzameling waarden die door het systeem op een gelijkaardige wijze zullen worden behandeld
- Bv: programma 1-500
  - Iedere waarde wordt op dezelfde manier behandeld
    - Welke equivalentieklassen?
- Bv: programma: bus, vrachtwagen, auto, motor
  - Welke equivalentieklassen?

# Equivalence Partitioning

- Voorbeeld: in dienst nemen adhv leeftijd
  - 0 – 16 → niet aannemen
  - 16 – 18 → aannemen als part-time
  - 18 – 65 → aannemen als full-time
  - 65 – 99 → niet aannemen
- Equivalentieklassen?

# Boundary Value Analysis

---



## Boundary Value Analysis

- Ervaring wijst uit dat testcases die gebruik maken van grenswaarden meer kans op slagen hebben.
- Bijvoorbeeld:
  - Programmeefouten waarbij een teller net één waarde te ver of net niet ver genoeg telt

# Boundary Value Analysis

- Voorbeeld:

```
If (applicantAge >= 0 && applicantAge <= 16)
 hireStatus="NO";
If (applicantAge >= 16 && applicantAge <= 18)
 hireStatus="PART";
If (applicantAge >= 18 && applicantAge <= 55)
 hireStatus="FULL";
If (applicantAge >= 55 && applicantAge <= 99)
 hireStatus="NO";
```

# Boundary Value Analysis

- We selecteren testcases die op de grenzen van de equivalentie klassen liggen.
- Dit wil zeggen
  - een testcase net op de grens
  - vervolgens een testcase net boven de grens
  - één net onder deze grens

# Boundary Value Analysis

- Voorbeeld
  - Waarde die tussen -1.0 en 1.0 moet liggen → testcases voor -1.0, 1.0, -1.1, -0.9, 0.9 en 1.1
  - Euro's
  - Aantal waardes → test cases voor minimum en maximum aantal en voor een beetje minder en een beetje meer
  - Leeftijd?



# Boundary Value Analysis

- Nadat je een aantal maal eenzelfde type invoervariabele hebt moeten testen, zal je opmerken dat dezelfde tests terugkomen → **standaard matrix** opstellen waarin typische tests voorkomen voor bepaald invoerveld.
- Hergebruik!

# Boundary Value Analysis

## ■ Test Matrix

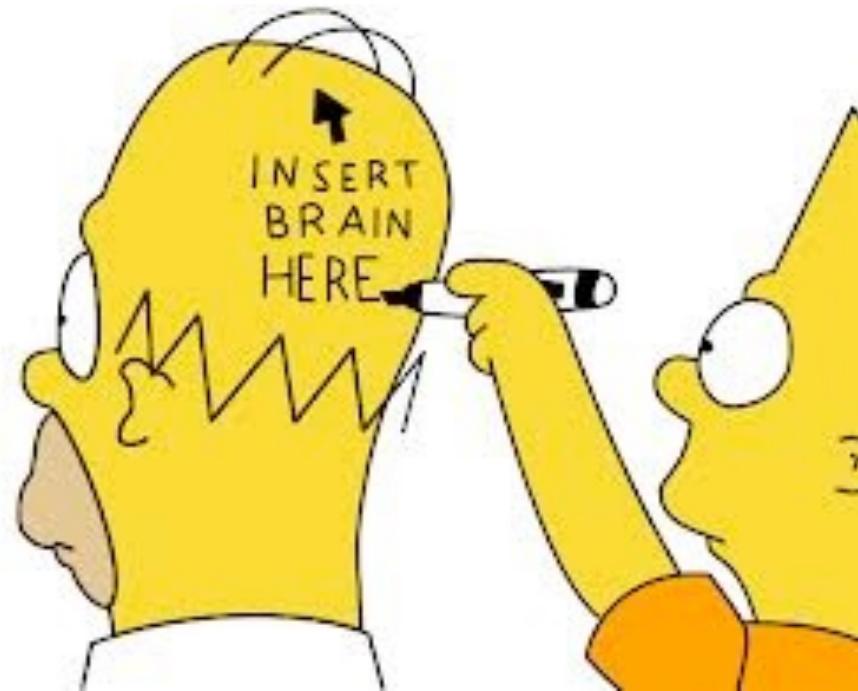
| Numeric (Integer) Input Field |                                         |
|-------------------------------|-----------------------------------------|
|                               | Nothing                                 |
|                               | LB of value                             |
|                               | UB of value                             |
|                               | LB of value - 1                         |
|                               | UB of value + 1                         |
| 0                             |                                         |
|                               | Negative                                |
|                               | LB number of digits or chars            |
|                               | UB number of digits or chars            |
|                               | Empty field (clear the default value)   |
|                               | Outside of UB number of digits or chars |
|                               | Non-digits                              |

# Boundary Value Analysis

- Zwaktes equivalentieklassen:
  - Er zijn een aantal blind spots:
    - bugs die zich niet op de grenzen of in de speciale gevallen bevinden
    - Het eigenlijke domein is niet altijd gekend
- ➔ Nog steeds creativiteit van de tester vereist

## Voorbeeld

- Matrix opstellen voor file handling?



# Voorbeeld

- Volle lokale schijf/netwerkdisk
- Bijna volle lokale schijf/netwerkdisk
- Lokale disk/netwerkdisk met schrijfbeveiliging
- Beschadigde lokale schijf/netwerkdisk
- Ongeformatteerde lokale schijf/netwerkdisk
- Verwijder de lokale schijf/netwerkdisk nadat de file geopend is
- Timeout bij het wachten tot de lokale schijf/netwerkdisk terug online is
- I/O van het toetsenbord of de muis terwijl er naar lokale schijf/netwerkdisk gesaved wordt
- Andere interrupts terwijl er naar de lokale schijf/netwerkdisk geschreven wordt
- Stroomuitval tijdens het schrijven naar de lokale schijf
- Lokale stroomuitval tijdens het schrijven naar de netwerkdisk
- Netwerkstroomuitval tijdens het schrijven naar de netwerkdisk

# Whitebox testontwerptechnieken

---

# Code coverage

- Hoeveel code is reeds getest?
  - Statement coverage
  - Decision coverage
  - Loop coverage
  - Path coverage

# Statement coverage

- Indien alle statements in de code ten minste één keer worden uitgevoerd = een statement coverage van 100%.

```
public boolean isMeerderjarig(int a) {
 boolean meerderjarig = false;
 if (a>18)
 meerderjarig = true;
 return meerderjarig;
}
```

# Decision coverage

- if-statement werkelijk testen betekent dat je twee tests moet uitvoeren: één keer met de conditie true en één keer met de conditie false
- Hetzelfde geldt voor een while-statement, een for-statement of een switch-statement.

# Loop coverage

- een test waarbij de body van de loop twee of meer keren wordt uitgevoerd;
- een test waarbij de body één keer wordt uitgevoerd
- een test waarbij de body niet wordt uitgevoerd (als dat mogelijk is).

# Path coverage

- In code met beslissingen (if, if-else) en herhalingen (while, do while, for) zijn er veel mogelijke paden
- Om inzicht te krijgen in de paden door de code neem je een activiteitendiagram als uitgangspunt.
- Zoek naar lineair onafhankelijke paden
  - Basispaden = combineerbaar tot elk ander pad
  - Een basispad mag niet bestaan uit een combinatie van andere basispaden

# Ervaringsgebaseerde tests

---

# Error guessing

---

# Error Guessing



- Een techniek gebaseerd op de mogelijkheid van de tester om te steunen op zijn ervaring, kennis en intuïtie om te voorspellen waar fouten in software kunnen gevonden worden

# Exploratory testing

---

# Exploratory testing

- “Exploratory testing is simultaneous learning, test design, and test execution.”

# Exploratory testing

## Wanneer toepassen?

- Geen of onvoldoende specificaties
- Te weinig tijd voor een volledig testontwerp
- Snelle feedback vereist
- Goede aanvulling op formele testtechnieken

## Voorwaarde

- Testteam met veel systeem- en domeinkennis

# Exploratory testing – hoe?

test design & uitvoering worden gelijktijdig gedaan

Adhv de testresultaten zal er dieper ingegaan worden op een bepaald gebied (explore)

de productieve gebieden van testen worden onmiddellijk uitgebreid

On the fly

Dit wil NIET zeggen dat het testen ongestructureerd/ongepland gebeurd

Exploratory testing is mooie uitbreiding van gestructureerd testen = effectief alle test cases uitschrijven



## Exploratory testing

- Succesvolle gebieden uitbreiden (explore!)
- Bij ET is er sprake van een gedetailleerde procedure waarin specifieke taken, aanpak, doelstellingen en producten zijn vastgelegd
- Verschil met Error Guessing!

# Exploratory testing

- Nadelen
  - Ontbreken van testontwerp als statische techniek
    - Opstellen van een testontwerp leidt tot het vinden van fouten in de specificatie!!
  - Te veel nadruk op testuitvoering
    - In praktijk lopen de meeste projecten uit en testuitvoering is de enige testfase die zich op het kritieke pad bevindt
    - Het maken van een testontwerp (testspecificaties en testscript) is een voorbereiding om de fase testuitvoering zo efficiënt mogelijk te kunnen laten verlopen

# Exploratory testing



- Nadelen:
  - Door het ontbreken van resultaatvoorspellingen worden fouten niet ontdekt
    - ‘the eye seeing what it wants to see’
  - Exploratory testen eist ervaren testers

# SMART Testcases

---

# SMART testcases

- **Specifiek**
  - Duidelijk
    - Testcases moeten niet dubbelzinnig zijn. er moeten geen meerdere interpretaties mogelijk.
  - Bondig
    - Testcases bevatten net genoeg detail zodat een tester, die bekend is met het functioneel gebied waarop de test van toepassing is, de test kan uitvoeren.
  - Volledig
    - Testcases definiëren alle verwachte resultaten, testdata en afhankelijkheden.

# SMART testcases

- **Meetbaar**
  - Het resultaat van iedere test moet objectief kunnen geëvalueerd worden zodat men weet of de test geslaagd is of faalt.
  - Alle testcases moeten gekoppeld worden aan de vereisten (requirements) zie zij af te oetsen zodat het mogelijk is om na te gaan wat de impact is van een verandering van deze vereisten.

# SMART testcases

- **Aanvaardbaar**
  - Testcases moeten, voor uitvoering, worden goedgekeurd door de stakeholders van de vereisten die gecontroleerd worden door testcase. Iedere wijziging aan de testcase veroorzaakt door een wijziging aan de vereisten, moet ook worden goedgekeurd.

# SMART testcases

- **Realistisch**

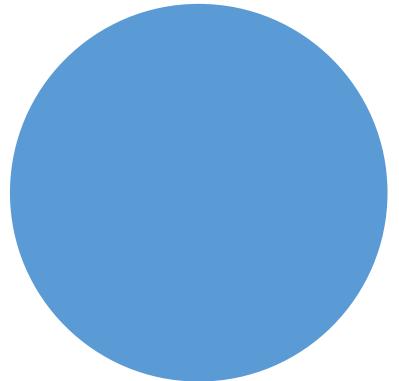
- Realistische testcases controleren het functioneren van het systeem bij normaal gebruik, maar ook bij abnormaal gebruik/misbruik.
- Realistische testcases controleren functies en eigenschappen die werden opgenomen in de vereisten.
- Realistische testcases kunnen uitgevoed worden op de platformen en configuraties die beschikbaar zijn in de testomgeving.
- Realistische testcases controleren GEEN functies en vereisten die niet werden goedgekeurd (niet aanwezig in de vereisten).

## SMART testcases



- **Tijdsgebonden**

- Testcases moeten kunnen afgerond worden binnen de planning van het project.



# Software design & Quality Assurance

Johan van den Broek

# Mocking

---

## Testing en mockobjecten

- Unit testing = het testen van aparte klassen of methoden
- MAAR klassen zijn vaak afhankelijk van andere klassen





# Testing en mockobjecten

- Unit tests worden uitgevoerd op kleine onderdelen vh systeem. Het is niet de bedoeling dat héél het systeem of grote delen vh systeem geïnitialiseerd worden om uw unit tests uit te voeren.
  - Dit vraagt immers veel tijd!
  - Je tests zijn op deze manier aan te veel componenten gekoppeld. Het is juist de bedoeling dat componenten op zich worden getest, en niet dat de componenten in interactie met andere componenten getest worden

# Mock objecten



- Mockobjecten zijn objecten die speciaal worden aangemaakt om het gedrag van andere objecten te simuleren.
- Een mockobject kan je vergelijken met een crashtest-dummy tijdens het testen van auto's.

# Mock-objects

- Imitateer echte objecten zodat je je beter kan concentreren op het testen van je eigen code.
- Een klassiek voorbeeld van een mockobject is wanneer we beroep doen op een data uit een databank. Tijdens het testen simuleert een mockobject de database en zorgt er op deze manier voor dat de randvoorwaarden steeds dezelfde blijven.

## Mocking: Wanneer?

Onvoorspelbaar of moeilijk uit te lokken gedrag

Moeilijke setup

Traag object

User Interface

Onbestaand object

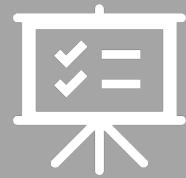
# Praktijk

---

# Mock objecten



Mock objecten zijn dummy implementaties van interfaces of klassen waarin je de output van bepaalde methoden definieert.



Mock objecten maken het mogelijk om een unit test te schrijven voor een klasse zonder dat de uitkomst van deze test beïnvloed wordt door andere klassen.

# Mock objecten

- Modules werken niet geïsoleerd, maar werken samen met andere onderdelen
- Om onze modules in een geïsoleerde omgeving te testen, maken we mock objecten aan.

# Mock objecten

- Twee opties
  - Je kan mock objecten steeds zelf implementeren.
  - Je kan een framework gebruiken zodat het niet nodig is om zelf mock objecten te voorzien
- Wij maken gebruik van het framework EasyMock
- Easymock is een populair framework dat gemakkelijk gebruikt kan worden met JUnit.

# EasyMock



- Download EasyMock en voeg easymock.jar toe aan het classpath

# EasyMock

- In de JUnit test
  - static import van org.easymock.EasyMock

```
import static org.easymock.EasyMock.*;
```

- Static import?
  - double r = Math.cos(Math.PI \* theta);
  - import static java.lang.Math.\*;
  - double r = cos(PI \* theta);

# EasyMock

- Hoe maken we een mock object aan?
  1. Maak een Mock Object aan voor de interface die we willen simuleren
  2. Registreer het verwachte gedrag (RECORD)
  3. Plaats het object in de replay modus (PLAYBACK)
  4. Controle of alle gedrag werd afgespeeld (optioneel)

# Stap 1: maak het mock object aan

- Identificeer welke klassen je wil mocken. We kunnen een klasse mocken door gebruik te maken van zijn interface.
- We maken gebruik van de createMock methode om een mock object aan te maken

```
SomeInterface mock = createMock(SomeInterface.class);
```

## Stap 2: Imiteer de methoden

- In EasyMock wordt het gedrag **eerst geregistreerd** en vervolgens terug afgespeeld (RECORD/REPLAY)
- Tijdens de registratie (RECORD) **trainen we de mock** en maken we duidelijk welke methode oproepen het mock object mag verwachten.
- Tijdens het afspelen, zal hij **na een verwachte methode oproepen de aangeleerde waarden** terugsturen. Wanneer een methode aangeroepen wordt die het mock object niet verwacht, zal een Exception worden gegooid zodat de test faalt.
- Het mock object zal dus niet alle acties opnieuw afspelen, maar het object wordt in een neutrale staat geplaatst zodat wanneer de aangeleerde methoden worden opgeroepen, het object de gewenste reacties kan geven.

```
stmtMock.executeQuery("SELECT * FROM survey"));
stmtMock.execute();
```

## Stap 3: Plaats in replay modus

- Plaats in replay modus en voer de te testen code uit

```
stmtMock.replay();
rsMock.replay();

assertEquals(55,idao.getNoOfColumns());
```

## Stap 4: Controle (optioneel)

- Als laatste stap kan je laten nagaan of daadwerkelijk alle aangeleerde methoden werden opgevraagd.
- Indien je deze controle niet voorziet, is het dus niet verplicht dat het mock object alle aangeleerde methoden terug afspeelt.
- Indien een methode niet werd opgeroepen, gooit EasyMock een exception.

```
verify(stmtMock);
```

# Stap voor stap

```
// Maak een mock object aan
SomeInterface mock = createMock(SomeInterface.class);

// Registreer gedrag (RECORD)
mock.doStuff("argument");

// Replay gedrag (PLAYBACK)
replay(mock);

// Voer de code uit die we wensen te testen
String newValue=test.perform()
// de methode perform() roept de methode doStuff() aan

// Controle van gedrag
verify(mock);
```

# EasyMock

- Voorbeeld 1

```
@Before
public void setUp() {
 mock = createMock(Collaborator.class); // 1
 classUnderTest = new ClassUnderTest();
 classUnderTest.addListener(mock);
}

@Test
public void testRemoveNonExistingDocument() {
 // 2 (we verwachten niets!)
 replay(mock); // 3
 classUnderTest.removeDocument("Does not exist");
}
```

# EasyMock

- Voorbeeld 2

```
@Before
public void setUp() {
 mock = createMock(Collaborator.class); // 1
 classUnderTest = new ClassUnderTest();
 classUnderTest.addListener(mock);
}

@Test
public void testAddDocument() {
 mock.documentAdded("New Document"); // 2
 replay(mock); // 3
 classUnderTest.addDocument("New Document", new byte[0]);
}
```

# EasyMock

- Voorbeeld 3

```
@Before
public void setUp() {
 mock = createMock(Collaborator.class); // 1
 classUnderTest = new ClassUnderTest();
 classUnderTest.addListener(mock);
}

@Test
public void testAddDocument() {
 mock.documentAdded("New Document"); // 2
 replay(mock); // 3
 classUnderTest.addDocument("New Document", new byte[0]);
 verify(mock); // 4
}
```

# Meerdere oproepen van een methode

```
@Test
public void testAddAndChangeDocument() {
 mock.documentAdded("Document");
 mock.documentChanged("Document");
 mock.documentChanged("Document");
 mock.documentChanged("Document");
 replay(mock);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 verify(mock);
}
```

# Meerdere oproepen van een methode

- Indien we meerdere oproepen van een methode verwachten, kunnen we dit aanleren aan ons Mock object door één eenvoudige opdracht

```
@Test
public void testAddAndChangeDocument() {
 mock.documentAdded("Document");
 mock.documentChanged("Document");
 expectLastCall().times(3);
 replay(mock);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 classUnderTest.addDocument("Document", new byte[0]);
 verify(mock);
}
```

# Meerdere oproepen van een methode

Indien de methode niet exact zo vaak wordt aangeroepen als voorzien, gooit het mockobject een AssertionError.

Het spreekt voor zich dat we dit steeds doen in combinatie van een controle: **verify(mock)**

# Return values

- Indien we het mockobject willen aanleren return values terug te sturen na een methode oproep dan maken we gebruik van expect()

```
@Test
public void testVoteAgainstRemoval() {
 expect(mock.documentAdded("Document")).andReturn(true);
 replay(mock);
 classUnderTest.addDocument("Document", new byte[0]);
 // ...
 verify(mock);
}
```

# Return values

- Twee mogelijkheden

```
expect(mock.voteForRemoval("Document")).andReturn(true);
```

of

```
mock.voteForRemoval("Document");
expectLastCall().andReturn(true);
```

# Werken met Exceptions

- Wanneer we wensen dat na een bepaalde methode oproep het mockobject een exception gooit, kunnen we gebruik maken van de methode `andThrow(Throwable trowable)`
- Opmerking
  - Unchecked exceptions (`RuntimeException`, `Error` en subklassen) kunnen vanuit iedere methode worden gegooid
  - Checked exceptions kunnen enkel gegooid worden uit methoden die dit aangeven.

```
expect(mock.storeUser("Jo", "Janssens", "go"))
 .andThrow(new RuntimeException());
```

# Verschillend gedrag voor dezelfde methode

- Het is mogelijk om veranderend gedrag voor een methode vast te leggen.
- Bijvoorbeeld
  - Een methode geeft eerst tweemaal true terug
  - Vervolgens driemaal een RuntimeException
  - Vervolgens false.

```
expect(mock.eenMethode("test"))
 .andReturn(true).times(2)
 .andThrow(new RuntimeException(), 3)
 .andReturn(false);
```

# Aantal oproepen specifiëren

- Om een aantal calls te simuleren zijn er meerdere methoden. We gebruikten tot hiertoe
  - times(int count).
- Andere mogelijkheden:

```
times(int min, int max)
 // verwacht een aantal oproepen tussen min en max
atLeastOnce()
 // verwacht tenminste één oproep
anyTimes()
 // verwacht een ongelimiteerd aantal oproepen
once() en times(1)
 // indien er geen aantal vermeld wordt,
 // wordt één oproep verwacht
 // je kan dit echter ook expliciet angeven
```

# Strict Mocks

- Bij een standaard Mock object wordt de volgorde van de oproepen niet gecontroleerd.
- Indien je dit wil controleren, zal je een strict Mock Object moeten aanmaken

```
mock = EasyMock.createStrictMock();
```

# Software Design & Quality Assurance

---

Johan van den Broek



```
mirror_mod = modifier_obj
mirror object to mirror
mirror_mod.mirror_object
operation = "MIRROR_X":
 mirror_mod.use_x = True
 mirror_mod.use_y = False
 mirror_mod.use_z = False
operation == "MIRROR_Y":
 mirror_mod.use_x = False
 mirror_mod.use_y = True
 mirror_mod.use_z = False
operation == "MIRROR_Z":
 mirror_mod.use_x = False
 mirror_mod.use_y = False
 mirror_mod.use_z = True

selection at the end -add
ob.select= 1
ler_ob.select=1
ntext.scene.objects.active
("Selected" + str(modifier))
rror_ob.select = 0
bpy.context.selected_objects
ta.objects[one.name].sel
int("please select exactly one ob
- OPERATOR CLASSES -->
types.Operator):
 X mirror to the selected ob
ject.mirror_mirror_x"
rror X"
context):
 ntext.active_object is not
```

# What is Software Engineering?

- Software engineering is defined as a process of analyzing user requirements and then designing, building, and testing software application which will satisfy those requirements.
- IEEE, in its standard 610.12-1990, defines software engineering as the application of a systematic, disciplined, which is a computable approach for the development, operation, and maintenance of software.

# Why Software Engineering?

- It was in the late 1960s when many software projects failed.
- Many software became over budget. Output was an unreliable software which is expensive to maintain.
- Larger software was difficult and quite expensive to maintain.
- Lots of software not able to satisfy the growing requirements of the customer.
- Complexities of software projects increased whenever its hardware capability increased.
- Demand for new software increased faster compared with the ability to generate new software.

# Need of Software Engineering

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

# Software Development Methodologies

---

# Waterfall model

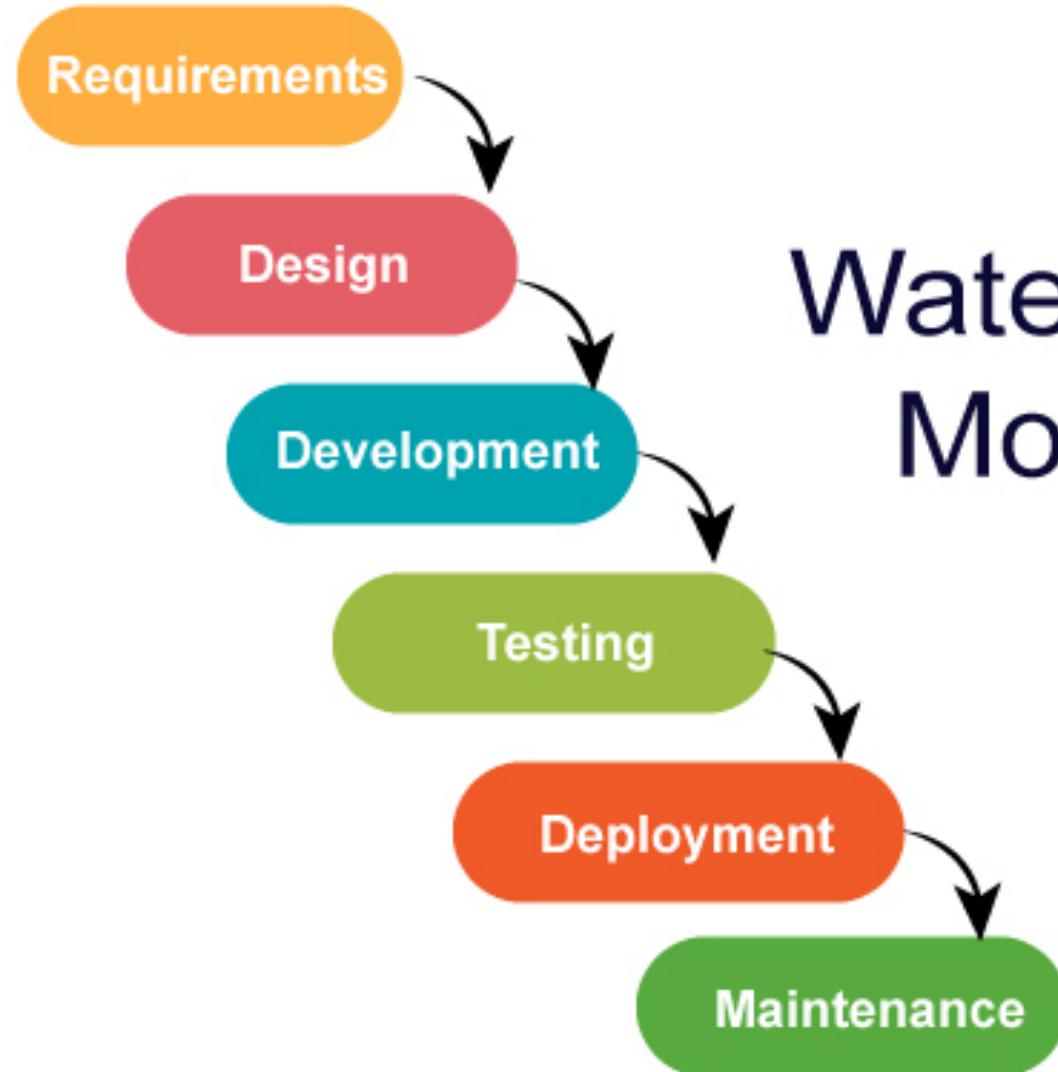
---

# Waterfall model

- Waterfall model is an example of a sequential model. In this model, the software development activity is divided into different phases and each phase consists of a series of tasks and has different objectives.
- Waterfall model is the pioneer of the Software Development Life Cycle (SDLC) processes.
- It was the first model which was widely used in the software industry. It is divided into phases and output of one phase becomes the input of the next phase. It is mandatory for a phase to be completed before the next phase starts. In short, there is no overlapping in the Waterfall model

# Waterfall model

---



# Waterfall Model

# Advantages of using the Waterfall model

Simple and easy to understand and use.

For smaller projects, the waterfall model works well and yield the appropriate results.

Since the phases are rigid and precise, one phase is done one at a time, it is easy to maintain.

The entry and exit criteria are well defined, so it is easy and systematic to proceed with quality.

Results are well documented.

# Disadvantages of using Waterfall model

Cannot adopt the changes in requirements

It becomes very difficult to move back to the phase. For example, if the application has now moved to the testing stage and there is a change in requirement, It becomes difficult to go back and change it.

Delivery of the final product is late as there is no prototype which is demonstrated intermediately.

For bigger and complex projects, this model is not good as a risk factor is higher.

## Disadvantages of using Waterfall model

Not suitable for the projects where requirements are changed frequently.

Does not work for long and ongoing projects.

Since the testing is done at a later stage, it does not allow identifying the challenges and risks in the earlier phase so the risk mitigation strategy is difficult to prepare.

# When to use the waterfall model

This model is used only when the requirements are very well known, clear and fixed.

Product definition is stable.

Technology is understood.

There are no ambiguous requirements

Ample resources with required expertise are available freely

The project is short.

# When to use the waterfall model

---



In Waterfall model, almost no customer interaction is involved during the development of the product. Once the product is ready, only then can it be demonstrated to the end users.



Once the product is developed and if any failure occurs then the cost of fixing such issues are very high, because we need to update everything from document till the logic.



In today's world, Waterfall model has been replaced by other models like agile.

# Agile

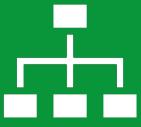


# What is Agile

---



Agile project management is a project philosophy or framework that takes an iterative approach towards the completion of a project.



A project management methodology is characterized by building products using short cycles of work that allow for rapid production and constant revision.

# Agile vs Waterfall

- Waterfall project management is another popular strategy that takes a different approach to project management than Agile. While Agile is an iterative and adaptive approach to project management, Waterfall is linear in nature and doesn't allow for revisiting previous steps and phases.
- Waterfall works well for small projects with clear end goals, while Agile is best for large projects that require more flexibility. Another key difference between these two approaches is the level of stakeholder involvement. In Waterfall, clients aren't typically involved, whereas **in Agile, client feedback is crucial.**

# Agile Methodology

---



The Agile methodology is a way to manage a project by breaking it up into several phases.



It involves constant collaboration with stakeholders and continuous improvement at every stage.



Once the work begins, teams cycle through a process of planning, executing, and evaluating.



Continuous collaboration is vital, both with team members and project stakeholders.

# Agile

---



Agile Methodology involves continuous iteration of development and testing in the Software Development Life Cycle process. This software development method emphasizes **on iterative, incremental, and evolutionary development.**



Agile development process breaks the product into smaller pieces and integrates them for final testing. It can be implemented in many ways, including Kanban, Scrum, XP (eXtreme Programming), etc.

# Agile's Core Values

<https://agilemanifesto.org>

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

# 12 principles from Agile Manifesto

---

Satisfying the customer  
is the top priority

Welcome changing  
requirements, even late  
in development

Deliver working  
software frequently

Development and  
business must work  
together

Build projects around  
motivated people

Face-to-face  
communication is the  
most efficient and  
effective method of  
conveying information

The primary measure of  
success is working  
software

Agile processes  
promote sustainable  
development

Maintain continuous  
attention to technical  
excellence and good  
design

Simplicity is essential

The best architectures,  
requirements, and  
designs emerge from  
self-organizing teams

Regularly reflect on  
work, then tune and  
adjust behavior

# Agile & Evolutionary Model



Evolutionary model is a combination of Iterative and Incremental model of software development life cycle.



The Evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle.

# Application of Evolutionary Model

---

It is used in large projects where you can easily find modules for incremental implementation. Evolutionary model is commonly used when the customer wants to start using the core features instead of waiting for the full software.

---

Evolutionary model is also used in object oriented software development because the system can be easily portioned into units in terms of objects.

# Evolutionary Model

## Advantages:

- In evolutionary model, a user gets a chance to experiment partially developed system.
- It reduces the error because the core modules get tested thoroughly.

## Disadvantages:

- Sometimes it is hard to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented and delivered.

# Scrum





## Difference Between Agile and Scrum

---

- The key difference between Agile and Scrum is that while Agile is a project management philosophy that utilizes a core set of values or principles, Scrum is a specific Agile methodology that is used to facilitate a project.

# Scrum

- A methodology in which a small team is led by a Scrum master, whose main job is to clear away all obstacles to completing work. Work is done in short cycles called sprints, but the team meets daily to discuss current tasks and roadblocks.

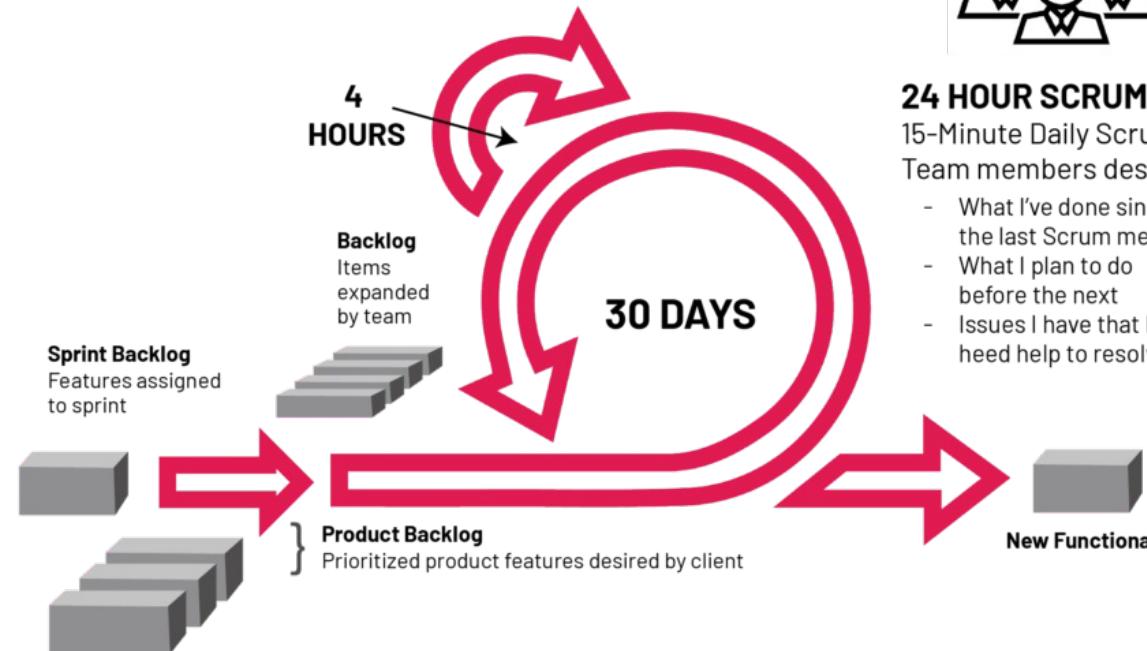
# What is Scrum?

- Scrum is a subset of Agile. It is a lightweight process framework for agile development, and the most widely-used one.
  - A “process framework” is a particular set of practices that must be followed in order for a process to be consistent with the framework. (For example, the Scrum process framework requires the use of development cycles called Sprints, the XP framework requires pair programming, and so forth.)
  - “Lightweight” means that the overhead of the process is kept as small as possible, to maximize the amount of productive time available for getting useful work done.

# Scrum process

- A Scrum process is distinguished from other agile processes by specific concepts and practices, divided into the three categories of Roles, Artifacts, and Events.

## SCRUM PROCESS



### 24 HOUR SCRUM

15-Minute Daily Scrum Meeting  
Team members describe:

- What I've done since the last Scrum meeting
- What I plan to do before the next
- Issues I have that I need help to resolve

# Roles



## Developers

Anyone on the team that is delivering the work. The name may be misleading but includes any team with members in roles outside of software development



## Product Owner

Holds the vision for the product and prioritizes the product backlog



## Scrum Master

Helps the team best use Scrum to build the product

# Artifacts

---

| Product Backlog                                                                                                                                                                                | Sprint Backlog                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Potentially Releasable Product Increment                                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>The product backlog is an ordered list of everything that is known to be needed in a product. It is constantly evolving and is never complete.</li></ul> | <ul style="list-style-type: none"><li>The sprint backlog is a list of everything that the team commits to achieve in a given sprint. Once created, no one can add to the sprint backlog except the development team.</li><li>If the development team needs to drop an item from the sprint backlog, they must negotiate it with the Product Owner. During this negotiation, the Scrum Master should work with the development team and Product Owner to try to find ways to create some smaller increment of an item rather than drop it altogether.</li></ul> | <ul style="list-style-type: none"><li>At the end of every sprint, the team must complete a product increment that is potentially releasable, meaning that meets their agreed-upon definition of done. (An example might be fully tested and fully approved.)</li></ul> |

# Events

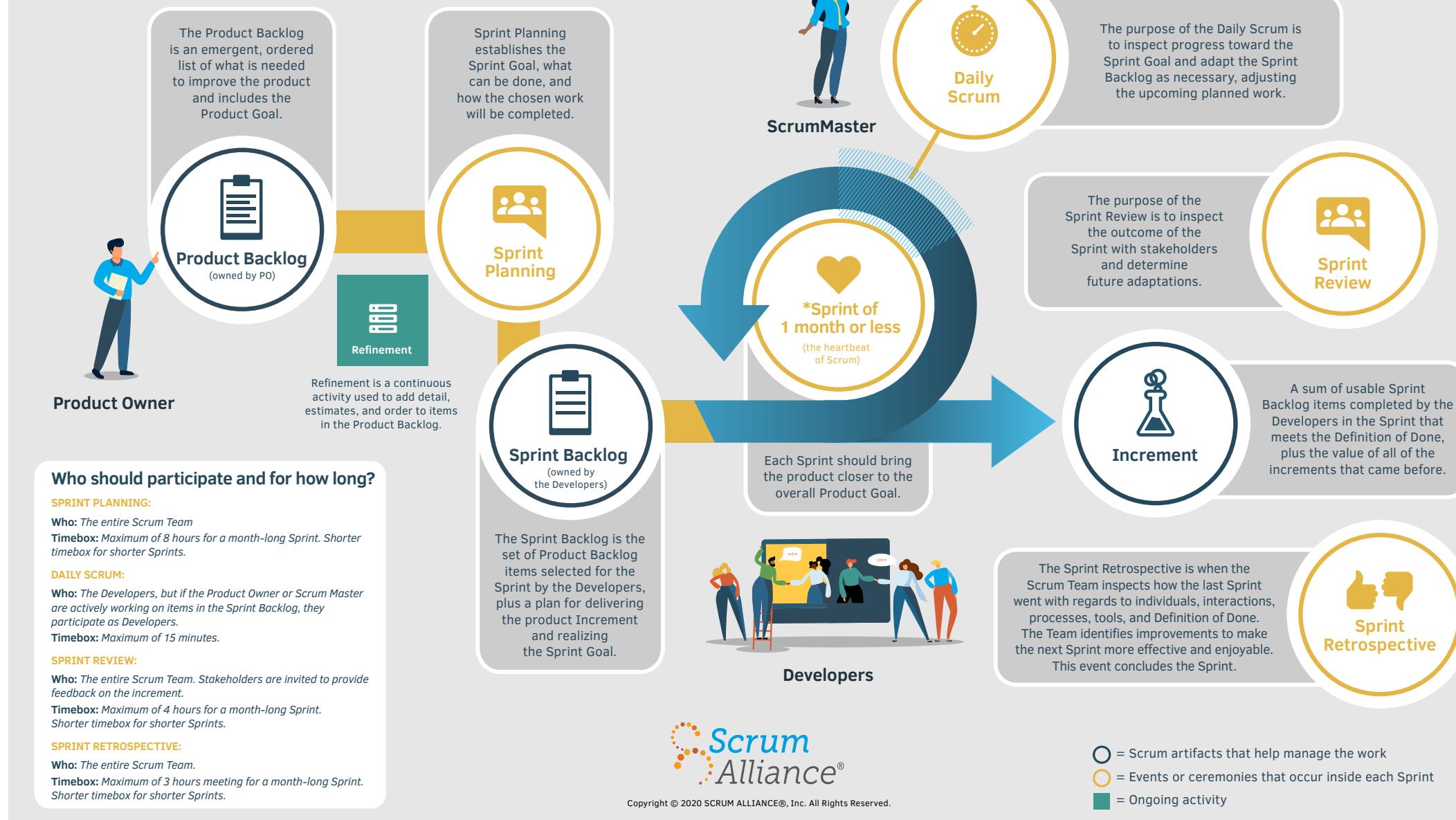
---

|                      |                                                                                                                                                                                                                                                                                                                                               |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| The Sprint           | The heartbeat of Scrum. Each sprint should bring the product closer to the product goal and is a month or less in length.                                                                                                                                                                                                                     |
| Sprint Planning      | The entire Scrum team establishes the sprint goal, what can be done, and how the chosen work will be completed. Maximum of 8 hours for a month-long Sprint. Shorter timebox for shorter sprints.                                                                                                                                              |
| Daily Scrum          | The Developers inspect the progress toward the sprint goal and adapt the sprint backlog as necessary, adjusting the upcoming planned work. May include Product Owner or Scrum Master if they are actively working on items in the sprint backlog. Maximum of 15 minutes each day.                                                             |
| Sprint Review        | The entire Scrum team inspects the sprint's outcome with stakeholders and determines future adaptations. Stakeholders are invited to provide feedback on the increment.                                                                                                                                                                       |
| Sprint Retrospective | The Scrum team inspects how the last sprint went regarding individuals, interactions, processes, tools, and definition of done. The Team identifies improvements to make the next sprint more effective and enjoyable. This is the conclusion of the sprint. Maximum of 3 hours for a month-long sprint, shorter timebox for shorter sprints. |

# Scrum in practice

- The Product Owner defines a vision using information from stakeholders and users. They identify and define pieces of value that can be delivered to move closer towards the Product Goal. Before the Developers can work on any pieces of value, the Product Owner must order the backlog so that the team knows what is most important. The team can help the Product Owner further refine what needs to be done, and the Product Owner may rely on the Developers to help them understand requirements and make trade-off decisions - this is where refinement becomes an important tool for the Scrum team.
- During sprint planning, the Developers pull a chunk from the top of the product backlog and decide how they will complete it. The team has a set time frame, the sprint, to complete their work. They meet at the daily scrum to inspect progress towards the sprint goal and plan for the upcoming day. Along the way, the Scrum Master keeps the team focused on the sprint goal and can help the team improve as a whole.
- At the end of the sprint, the work should be potentially shippable and ready to be used by a user or shown to a stakeholder. After each sprint, the team conducts a sprint review on the Increment and a retrospective on the process. Then they choose the next chunk of the backlog and the cycle repeats.

# The Scrum Framework At a Glance



# Lean Development

---

# Lean development

- Lean development is the application of Lean principles to software development. Lean principles got their start in manufacturing, as a way to optimize the production line to minimize waste and maximize value to the customer.

# 7 Lean Development Principles

- Eliminate waste
  - Build quality in
  - Create knowledge
  - Defer commitment
  - Deliver fast
  - Respect people
  - Optimize the whole
-

# Eliminate waste

Unnecessary code or functionality: Delays time to customer, slows down feedback loops

Starting more than can be completed: Adds unnecessary complexity to the system, results in context-switching, handoff delays, and other impediments to flow

Delay in the software development process: Delays time to customer, slows down feedback loops

Unclear or constantly changing requirements: Results in rework, frustration, quality issues, lack of focus

Bureaucracy: Delays speed

# Eliminate waste

Slow or ineffective communication: Results in delays, frustrations, and poor communication to stakeholders which can impact IT's reputation in the organization

Partially done work: Does not add value to the customer or allow team to learn from work

Defects and quality issues: Results in rework, abandoned work, and poor customer satisfaction

Task switching: Results in poor work quality, delays, communication breakdowns, and low team morale

# Build quality in

Pair programming: Avoid quality issues by combining the skills and experience of two developers instead of one

Test-driven development: Writing criteria for code before writing the code to ensure it meets business requirements

Incremental development and constant feedback

Minimize wait states: Reduce context switching, knowledge gaps, and lack of focus

Automation: Automate any tedious, manual process or any process prone to human error

# Create knowledge

- Pair programming
- Code reviews
- Documentation
- Wiki - to let the knowledge base build up incrementally
- Thoroughly commented code
- Knowledge sharing sessions
- Training
- Use tools to manage requirements or user stories

# Defer commitment



Not plan (in excessive detail) for months in advance



Not commit to ideas or projects without a full understanding of the business requirements



Constantly be collecting and analyzing information regarding any important decisions

# Deliver fast

The question isn't why teams want to deliver fast, but rather, what slows them down.



Thinking too far in advance about future requirements



Blockers that aren't responded to with urgency



Over-engineering solutions and business requirements

# Respect for people



Communicating proactively and effectively



Encouraging healthy conflict



Surfacing any work-related issues as a team



Empowering each other to do their best work

# Optimize the whole

- Two vicious cycles into which Lean development teams often fall:
  - The first is releasing sloppy code for the sake of speed. When developers feel pressured to deliver at all costs, they release code that may or may not meet quality requirements. This increases the complexity of the code base, resulting in more defects. With more defects, there is more work to do, putting more pressure on developers to deliver quickly... so the cycle continues.
  - The second is an issue with testing. When testers are overloaded, it creates a long cycle time between when developers write code and when testers are able to give feedback on it. This means that developers continue writing code that may or may not be defective, resulting in more defects and therefore requiring more testing.

# Optimize the whole

- As the antidote to suboptimization, optimizing the whole is a Lean development principle that encourages Lean organizations to eliminate these sorts of vicious cycles by operating with a better understanding of capacity and the downstream impact of work.
- After identifying how value flows through their teams, many organizations decide to organize their software development teams to be complete, multi-disciplined, co-located product teams, which enables them to have everything they need to deliver a request from start to finish, without reference to other teams.
  - Example Spotify Model (Lean & Agile)

# Lean Development

Lean Software Development is an agile framework based on optimizing development time and resources, eliminating waste, and ultimately delivering only what the product needs.

The Lean approach is also often referred to as the Minimum Viable Product (MVP) strategy, in which a team releases a bare-minimum version of its product to the market, learns from users what they like, don't like and want to be added, and then iterates based on this feedback.

# What is a Minimum Viable Product?

A minimum viable product, or MVP, is a product with enough features to attract early-adopter customers and validate a product idea early in the product development cycle.

The MVP can help the product team receive user feedback as quickly as possible to iterate and improve the product.

# Purpose of a Minimum Viable Product?

- A company might choose to develop and release a minimum viable product because its product team wants to:
  - Release a product to the market as quickly as possible
  - Test an idea with real users before committing a large budget to the product's full development
  - Learn what resonates with the company's target market and what doesn't



# Examples of the Minimum Viable Product

---



## Airbnb

With no money to build a business, the founders of Airbnb used their own apartment to validate their idea to create a market offering short-term, peer-to-peer rental housing online. They created a minimalist website, published photos and other details about their property, and found several paying guests almost immediately.

---

# Foursquare

---

The location-based social network Foursquare started as just a one-feature MVP, offering only check-ins and gamification rewards. It wasn't until they had validated the idea with an eager and growing user base that the Foursquare development team began adding recommendations, city guides, and other features.



## Amazon

Jeff Bezos collected orders on his own, bought the books, and sent them to the clients. A large number of orders was a confirmation that people needed this service. So, he added more books to his website and bought warehouses.



# Dropbox

- The CEO of Dropbox, Drew Houston, knew that there were a lot of companies providing cloud-storage services. He decided to create an explanatory video that would describe the advantages of the service. The number of views exceeded 70,000 in just one night, and it was a success.

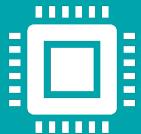


# DevOps



# Devops

---



DevOps is a collaboration between Development and IT Operations to make software production and deployment in an automated & repeatable way.



DevOps helps to increase the organization's speed to deliver software applications and services. The word 'DevOps' is a combination of two words, 'Development' and 'Operations.'

# What is IT Operations?

---

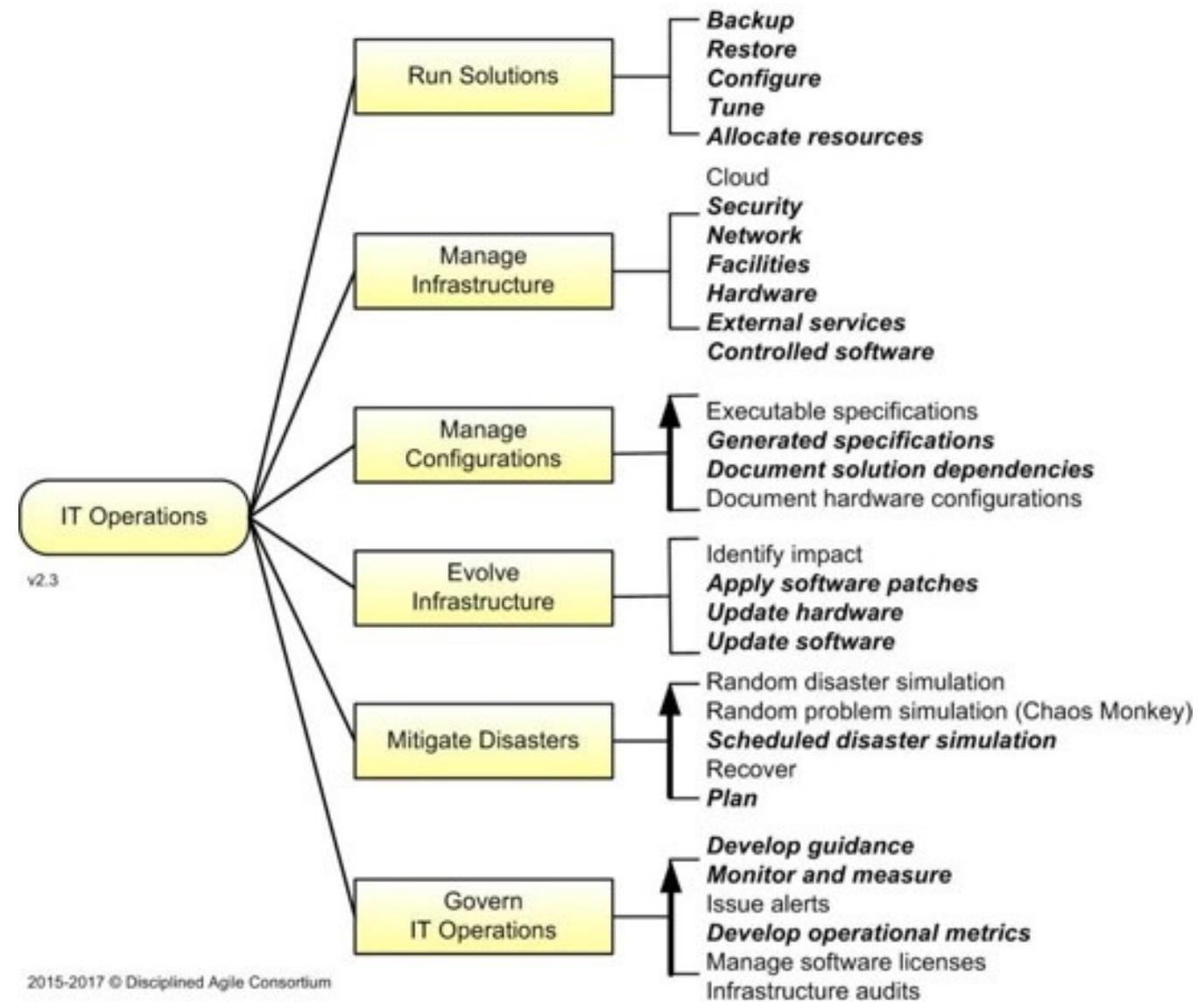


Information Technology Operations, or IT operations, are the set of all processes and services that are both provisioned by an IT staff to their internal or external clients and used by themselves, to run themselves as a business.



Important roles of the IT operations team include tech management, quality assurance, infrastructure management and confirmation that finished products meet all the customer's needs and expectations.

# IT Operations



# Why DevOps

- Software no longer merely supports a business; rather it becomes an integral component of every part of a business.
- Companies interact with their customers through software delivered as online services or applications and on all sorts of devices. They also use software to increase operational efficiencies by transforming every part of the value chain, such as logistics, communications, and operations.

# DevOps Philosophy



At its simplest, DevOps is about removing the barriers between two traditionally siloed teams, development and operations.



With DevOps, the two teams work together to optimize both the productivity of developers and the reliability of operations. They strive to communicate frequently, increase efficiencies, and improve the quality of services they provide to customers.



Organizations using a DevOps model have teams that view the entire development and infrastructure lifecycle as part of their responsibilities.

# DevOps Practices Explained



One fundamental practice is to perform very frequent but small updates. This is how organizations innovate faster for their customers.



These updates are usually more incremental in nature than the occasional updates performed under traditional release practices.



Frequent but small updates make each deployment less risky. They help teams address bugs faster because teams can identify the last deployment that caused the error.

# DevOps best practices

# DevOps best practices

---

Continuous  
Integration

Continuous  
Delivery

Microservices

Infrastructure as  
Code

Monitoring and  
Logging

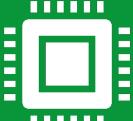
Communication  
and  
Collaboration

# Continuous Integration

---



Continuous integration is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.



The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

# Continuous Integration

the practices

Maintain a single source repository

Automate the build

Make your build self-testing

Every commit should build on an integration machine

Keep the build fast

Test in a clone of the production environment

Make it easy for anyone to get the latest executable version

Everyone can see what's happening

Automate deployment

# Continuous Integration

## How to do it

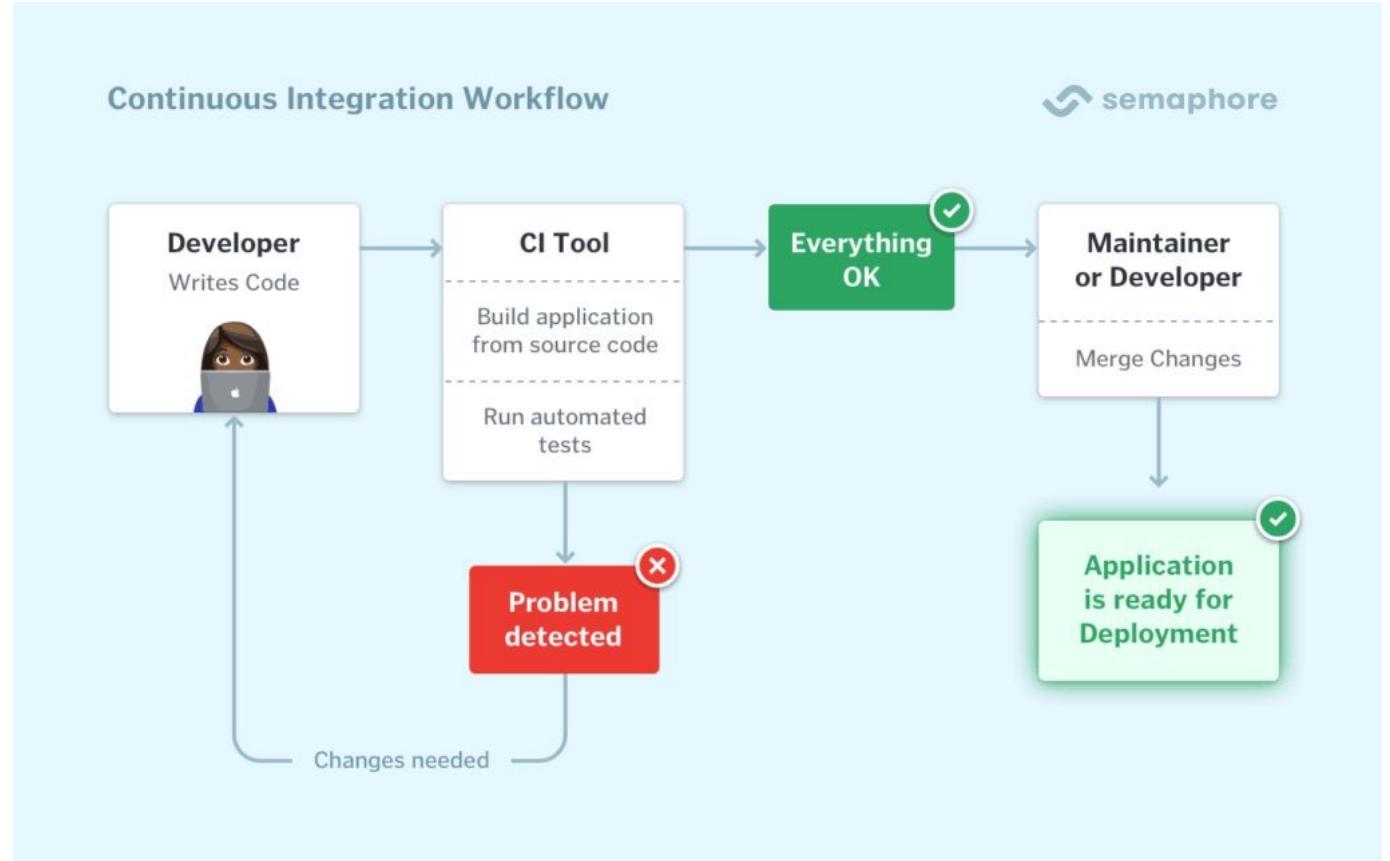
- Developers check out code into their private workspaces
- When done, commit the changes to the repository
- The CI server monitors the repository and checks out changes when they occur
- The CI server builds the system and runs unit and integration tests
- The CI server releases deployable artefacts for testing
- The CI server assigns a build label to the version of the code it just built
- The CI server informs the team of the successful build
- If the build or tests fail, the CI server alerts the team
- The team fixes the issue at the earliest opportunity
- Continue to continually integrate and test throughout the project

# Continuous Integration

Team responsibilities

- Check in frequently
- Don't check in broken code
- Don't check in untested code
- Don't check in when the build is broken
- Don't go home after checking in until the system builds

# Continuous Integration



# Continuous Delivery

---



Continuous delivery is a software development practice where code changes are automatically built, tested, and prepared for a release to production.

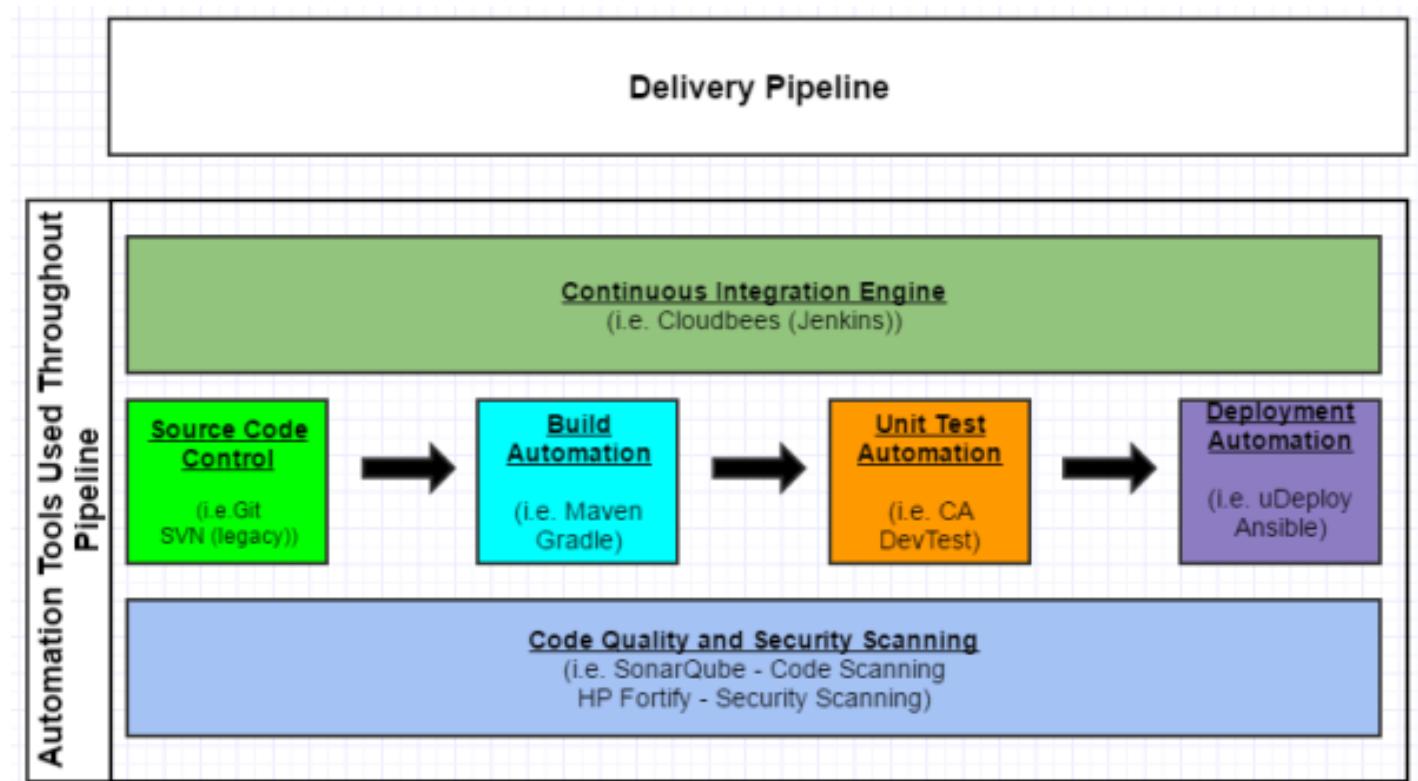


It expands upon continuous integration by deploying all code changes to a testing environment and/or a production environment after the build stage.



When continuous delivery is implemented properly, developers will always have a deployment-ready build artifact that has passed through a standardized test process.

# Continuous Delivery



# What is a CI/CD Pipeline?



A CI/CD pipeline is a series of orchestrated steps with the ability to take source code all the way into production.



The steps include building, packaging, testing, validating, verifying infrastructure, and deploying into all necessary environments.

# What is a CI/CD Pipeline?

Depending on organizational and team structures, there might be multiple pipelines required to achieve this goal.

A CI/CD pipeline can be triggered by some sort of event, such as a pull request from a source code repository (ie: a code change), the presence of a new artifact in an artifact repository, or some sort of regular schedule to match a release cadence.

# Microservices

---



The microservices architecture is a design approach to build a single application as a set of small services.



Each service runs in its own process and communicates with other services through a well-defined interface using a lightweight mechanism, typically an HTTP-based application programming interface (API).



Microservices are built around business capabilities; each service is scoped to a single purpose. You can use different frameworks or programming languages to write microservices and deploy them independently, as a single service, or as a group of services.

# Microservices @ Amazon

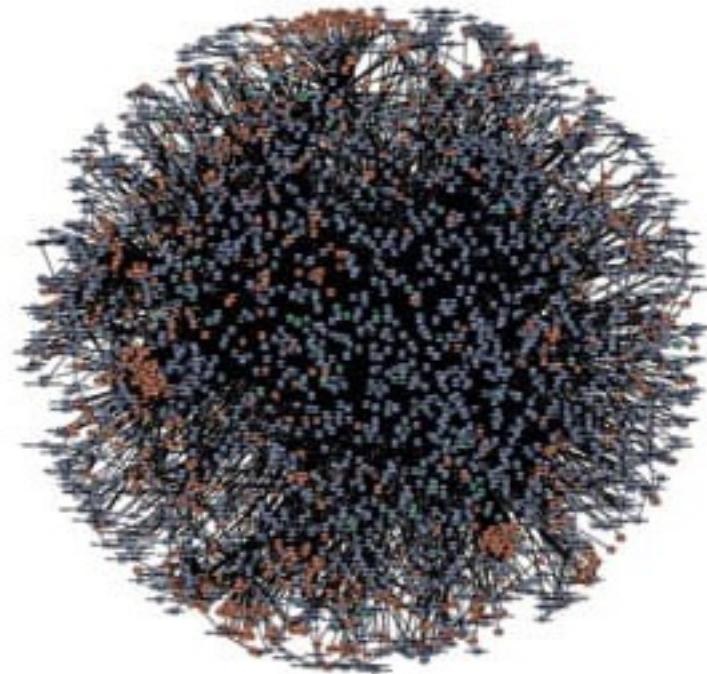
- In 2001, development delays, coding challenges, and service interdependencies inhibited Amazon's ability to meet the scaling requirements of its rapidly growing customer base. Faced with the need to refactor their system from scratch, Amazon broke its monolithic applications into small, independently-running, service-specific applications.

# Microservices @ Amazon

- Developers analyzed the source code and pulled out units of code that served a single, functional purpose.
- They wrapped these units in a web service interface.
- For example: They developed a single service for the Buy button on a product page, a single service for the tax calculator function, and so on.

# Microservices @ Amazon

Amazon's microservices  
infrastructure, a.k.a., the  
Death Star



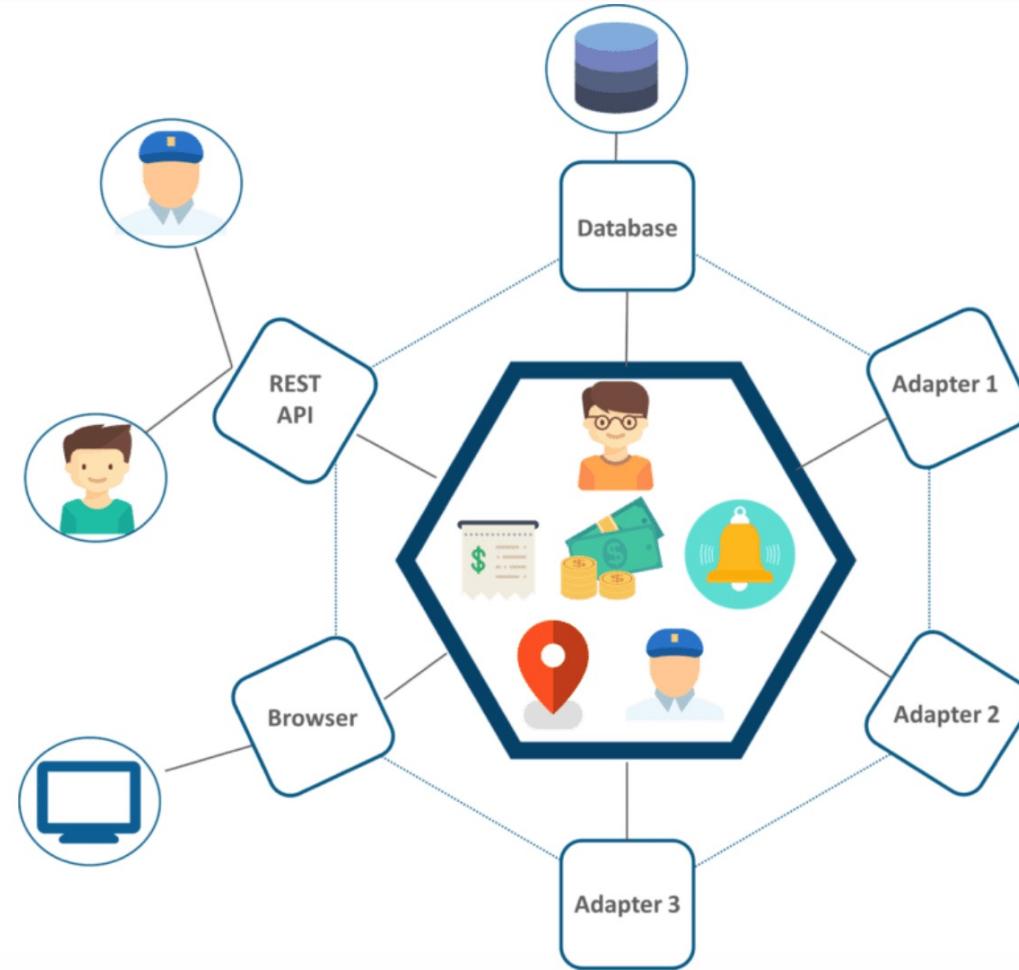
**amazon.com**

# Microservices @ Uber

- The platform struggled to efficiently develop and launch new features, fix bugs, and integrate their rapidly-growing, global operations.
- The complexity of Uber's monolithic application architecture required developers to have extensive experience working with the existing system - just to make minor updates and changes to the system.

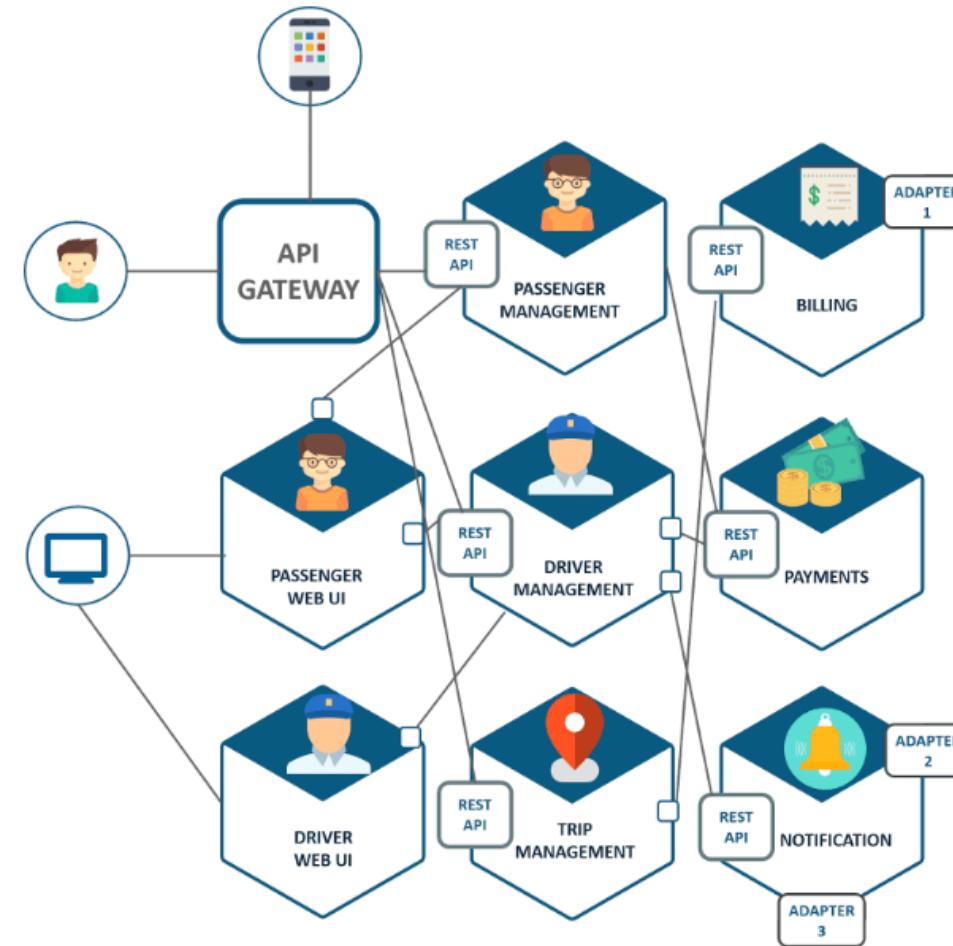
# Microservices @ Uber

Uber's original monolith



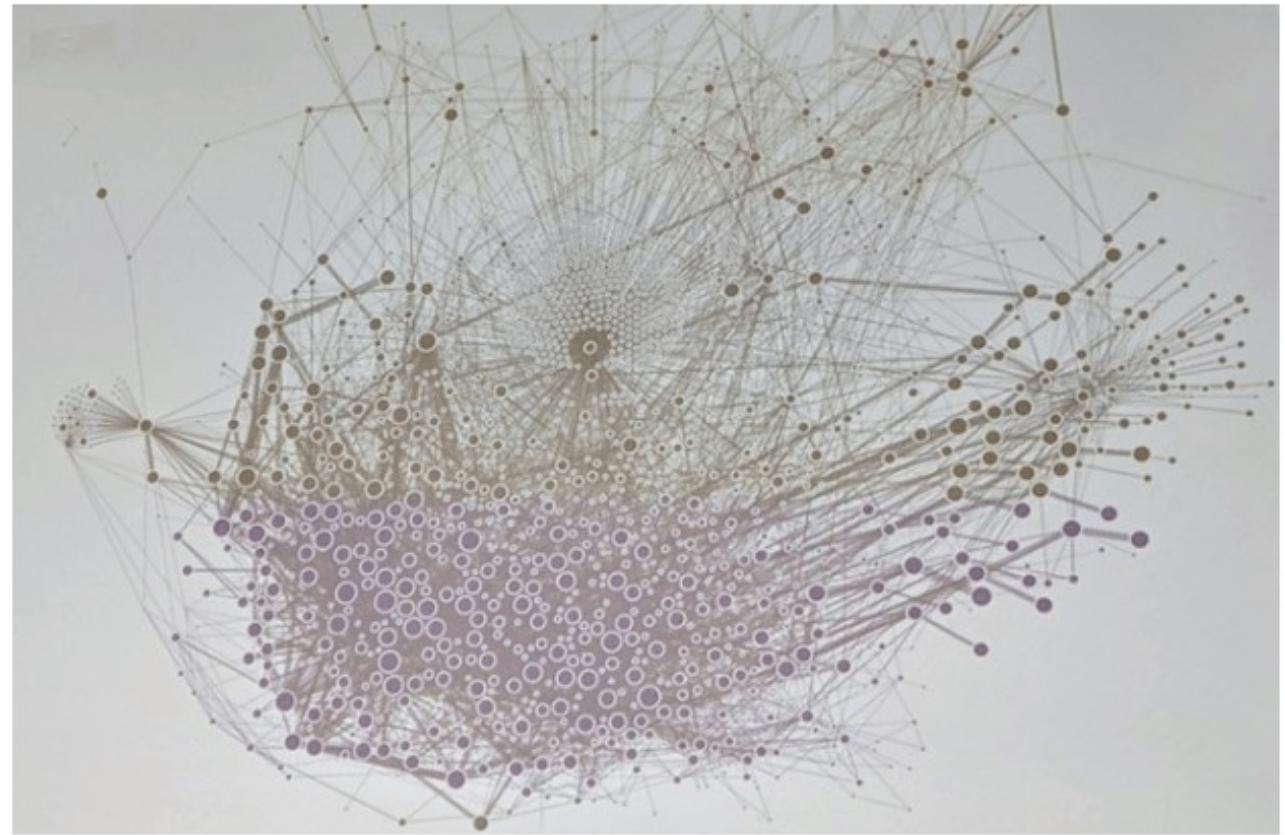
# Microservices @ Uber

Uber's first microservices architecture



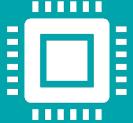
# Microservices @ Uber

Uber's microservices  
architecture from 2019



# Infrastructure as Code

---



Infrastructure as code is a practice in which infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration.



The cloud's API-driven model enables developers and system administrators to interact with infrastructure programmatically, and at scale, instead of needing to manually set up and configure resources.



Engineers can interface with infrastructure using code-based tools and treat infrastructure in a manner similar to how they treat application code. Because they are defined by code, infrastructure and servers can quickly be deployed using standardized patterns, updated with the latest patches and versions, or duplicated in repeatable ways.

# Monitoring and Logging

Organizations monitor metrics and logs to see how application and infrastructure performance impacts the experience of their product's end user.

By capturing, categorizing, and then analyzing data and logs generated by applications and infrastructure, organizations understand how changes or updates impact users, shedding insights into the root causes of problems or unexpected changes.

Active monitoring becomes increasingly important as services must be available 24/7 and as application and infrastructure update frequency increases.

# Communication and Collaboration

- Increased communication and collaboration in an organization is one of the key cultural aspects of DevOps.

# Communication and Collaboration

- The use of DevOps tooling and automation of the software delivery process establishes collaboration by physically bringing together the workflows and responsibilities of development and operations.
- Building on top of that, these teams set strong cultural norms around information sharing and facilitating communication through the use of chat applications, issue or project tracking systems, and wikis. This helps speed up communication across developers, operations, and even other teams like marketing or sales, allowing all parts of the organization to align more closely on goals and projects.

# Agile vs DevOps

# Agile vs DevOps

Stakeholders and communication chain in a typical IT process.



# Agile vs DevOps

Agile addresses gaps in Customer and Developer communications



# Agile vs DevOps

DevOps addresses gaps in Developer and IT Operations communications



# Agile vs DevOps

- Key difference
  - DevOps is a practice of bringing development and operations teams together whereas Agile is an iterative approach that focuses on collaboration, customer feedback and small rapid releases.
  - DevOps focuses on constant testing and delivery while the Agile process focuses on constant changes.
  - DevOps requires relatively a large team while Agile requires a small team.
  - DevOps leverages both shifts left and right principles, on the other hand, Agile leverage shift-left principle.
  - The target area of Agile is Software development whereas the Target area of DevOps is to give end-to-end business solutions and fast delivery.
  - DevOps focuses more on operational and business readiness whereas Agile focuses on functional and non-function readiness.

# Shift Left process involves

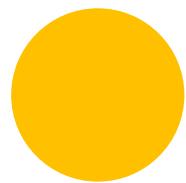
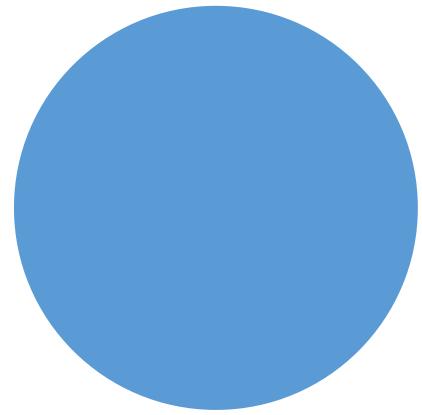
- Testing early and testing often to reduce the overall cost of project and maintain quality
- Testing continuously with a shorter feedback loop to avoid resolving defects in the end
- To automate everything and improve time to market
- To design for customer requirements and improve the overall customer experience

# Shift Right Approach involves

- Testing in the production environment to ensure product stability and performance in real word scenarios
- You can receive feedback and reviews from targeted users to ensure customer satisfaction
- Ability to test usage scenarios and real load levels that aren't possible to create in test environment

# References

- <https://aws.amazon.com/devops/what-is-devops/>
- <https://agilemanifesto.org>
- <https://resources.scrumalliance.org/Article/overview-scrum-framework>
- <https://www.guru99.com/agile-vs-devops.html>
- <https://www.softwaretestinghelp.com/devops-tutorials/>
- [https://cio-wiki.org/wiki/IT\\_Operations\\_\(Information\\_Technology\\_Operations\)](https://cio-wiki.org/wiki/IT_Operations_(Information_Technology_Operations))
- <https://www.guru99.com/devops-tutorial.html>
- <https://www.productplan.com/glossary/minimum-viable-product/>



# Software design & testing

Johan van den Broek

# Software Version Control

---

# De belangrijkste taken van version control

Bijhouden wat er gewijzigd werd

Bijhouden wie een wijziging heeft aangebracht

Bijhouden waarom een wijziging werd aangebracht

# Gebruik

- Version control systemen bewijzen hun nut in tal van beroepen, gaande van developers/designers over schrijvers/producers tot artiesten en componisten.
- Het bijhouden van het pad dat je tijdens eender welk creatief proces (bv. programmeren) hebt doorlopen is dan ook zeer belangrijk.

# Redenen

**Peace of mind:** alles wat we schrijven wordt geback-upt.

**Geschiedenis:** alles wat je gewijzigd hebt en vooral *waarom* wordt bijgehouden. Hierbij kunnen we bij elke wijziging een kort bericht toevoegen waarin we beschrijven wat er gewijzigd werd.

**Undo:** gemakkelijk naar vorige versies teruggaan.

**Experimenteren:** in een version control systeem is het eenvoudig om een sandbox op te zetten waarin je allerlei zaken kan uitproberen, zonder te vrezen dat er dingen mislopen.

# Als team

**Synchronisatie:** elke team member blijft up-to-date doordat iedereen toegang heeft tot de gemeenschappelijke inhoud

**Accountability:** wanneer iemand van het team een wijziging doorvoert wordt dit door het systeem bijgehouden en kan deze persoon er op aangesproken worden als er iets misloopt.

**Conflicten detecteren:** het version control systeem kan ervoor zorgen dat de *build* clean blijft door ervoor te zorgen dat er bijvoorbeeld geen bestanden of wijzigingen in bestanden van andere mensen worden overschreven.

# Version control concepten

|                                      |                                                                                                                                                  |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1. Repository</b>                 | Een soort database waarin je mappen, bestanden en de geschiedenis ervan wordt bijgehouden. Als het ware de core van het version control systeem. |
| <b>1. Working set (working copy)</b> | Een lokale versie met mappen en bestanden op je harde schijf, deze bevat wijzigingen die mogelijk nog niet in de repository zitten.              |
| <b>1. Add</b>                        | Nieuwe bestanden aan de working set toevoegen.                                                                                                   |
| <b>1. Check-in (commit)</b>          | De wijzigingen van een working set in de repository invoegen.                                                                                    |
| <b>1. Check-out (update)</b>         | De wijzigingen van de repository binnenhalen in de working set.                                                                                  |
| <b>1. Tag (label)</b>                | De huidige toestand beschrijven aan de hand van tekst/documentatie zodat er later gemakkelijk naar kan verwijzen.                                |
| <b>1. Revert (rollback)</b>          | Overschrijven van bestanden in de working set met een specifieke versie die typisch afkomstig is van de repository.                              |

## Gecentraliseerde version control

Hierbij is er een centrale server waarop de gedeelde versie(s) van het project worden bijgehouden.

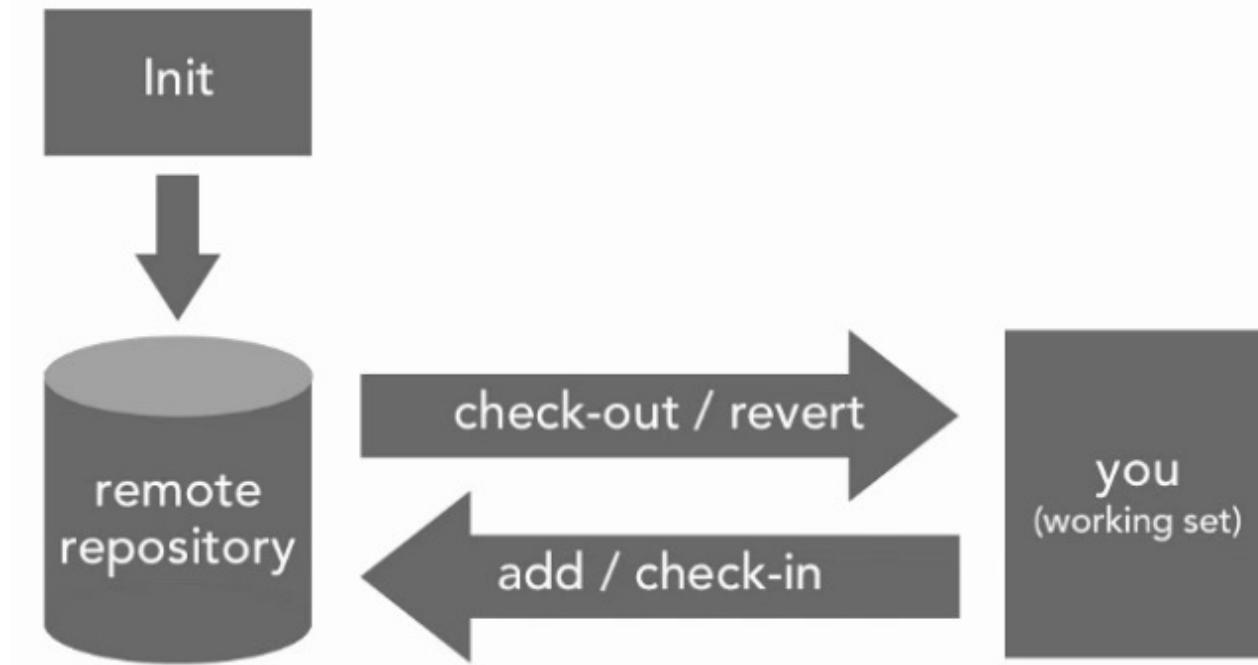
Het gaat uit van het principe dat een gebruiker eerst een check-out doet van de laatste versie van een project. Nadat een gebruiker wijzigingen heeft aangebracht doet hij een check-in, wat de aangepaste bestanden weer naar de centrale server doorstuurt.

# Gecentraliseerde version control

Het grote nadeel van deze techniek is dat de server online moet zijn om een check-in te kunnen uitvoeren van de code. Stel dat een programmeur een hele dag geen toegang heeft tot de server (of bijvoorbeeld op een vliegtuig werkt), dan kan hij enkel zijn wijzigingen bewaren maar geen tussentijdse versies inchecken (en dus beheren) op het version control systeem. Alle wijzigingen zullen dus als 1 grote changeset worden bekijken.

Een voordeel van een gecentraliseerd version control systeem is dat back-ups maken van de repository eenvoudiger wordt omdat de gegevens minder verspreid staan.

# Gecentraliseerde version control



# Gedistribueerde version control

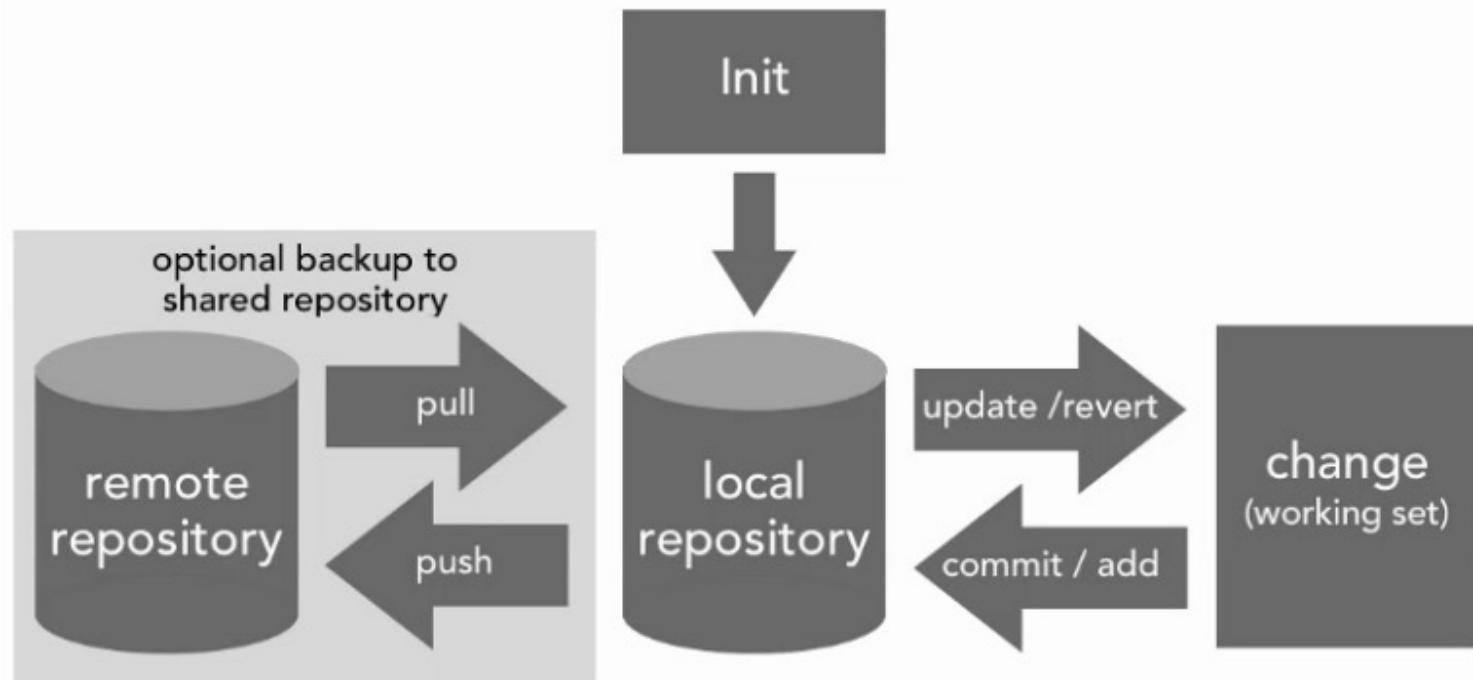
- Hierbij wordt gebruik gemaakt van 2 repositories.
  - Enerzijds is er een **lokale repository** op de computer van de gebruiker in kwestie. Op deze lokale repository kan een gebruiker te allen tijde versies inchecken en vorige versies van de bestanden bekijken.
  - Daarnaast is er een **remote repository**, waar een gebruiker zijn lokale repository mee kan synchroniseren.

# Gedistribueerde version control

Het grote voordeel aan dit soort systemen is dat ze de gebruiker toelaten om verder te werken met de version control, ook al is er op dat moment geen verbinding met de remote repository. Elke gebruiker heeft dus lokaal toegang tot de full history van het project. Deze kopij heeft bevatten alle metadata (check-in berichten, tags, ...) van de originele versie.

In vele gevallen wordt er gewerkt met een Cloud-gebaseerde remote repository waar verschillende gebruikers hun versies met elkaar kunnen delen. Een voorbeeld hiervan is GitHub, dat gebruikt kan worden als remote repository voor het Git version control systeem.

# Gedistribueerde version control



# Build Tools en Continuous Integration

---

# Build tools

---

# Build tool: Maven

- **Het bouwproces van software vereenvoudigen.** Maven haalt de nood niet weg om te weten wat er onder de motorkap gebeurt, maar het schermt wel veel details af.
- **Een eenvormig bouwsysteem aanleveren:** Maven laat toe om een project op te bouwen aan de hand van een project object model (POM) en een verzameling gedeelde plugins die door verschillende projecten worden gebruikt, waardoor er een uniform build systeem wordt aangeboden. Eens je gewoon bent aan 1 Maven project kan je deze kennis gemakkelijk herbruiken voor andere projecten en begrijp je meteen wat er in die andere projecten juist gebeurt.
- **Het leveren van kwalitatieve projectinformatie:** Maven kan voorzien in uitgebreide project informatie die opgenomen wordt in de POM en gegenereerd wordt vanuit de project source. Voorbeelden hiervan zijn: change log documenten, mailing lists, dependency lists, unit test rapporten (inclusief coverage van testen)

# Build tool: Maven

- **Het verstrekken van richtlijnen voor best practices rond softwareontwikkeling:** Maven probeert best practices voor ontwikkeling in te bedden in het aangeboden systeem en duwt op die manier Maven projecten in de goede richting. Zo zijn bijvoorbeeld specificatie, uitvoer en rapportering van unit testen een deel van de normale build cycle in maven.
- **Eenvoudige migratie naar nieuwe features mogelijk maken:** Maven biedt een gemakkelijke manier om maven-installatie makkelijk te updaten zodat ze gebruik kunnen maken van vernieuwingen die binnen Maven werden ontwikkeld.