Samenvatting Software Engineering

1. DevOps

Richtlijnen BB

Schriftelijk, gesloten boek:

Aangeven wat DevOps is, hoe het ontstaan is, de noodzaak, enz.

DevOps principes kunnen uitleggen

Kennis van de slides en het bijhorende E-book

Slides

DevOps focust zich op het opleveren van snelle IT-services door het gebruik van agile, lean practices.

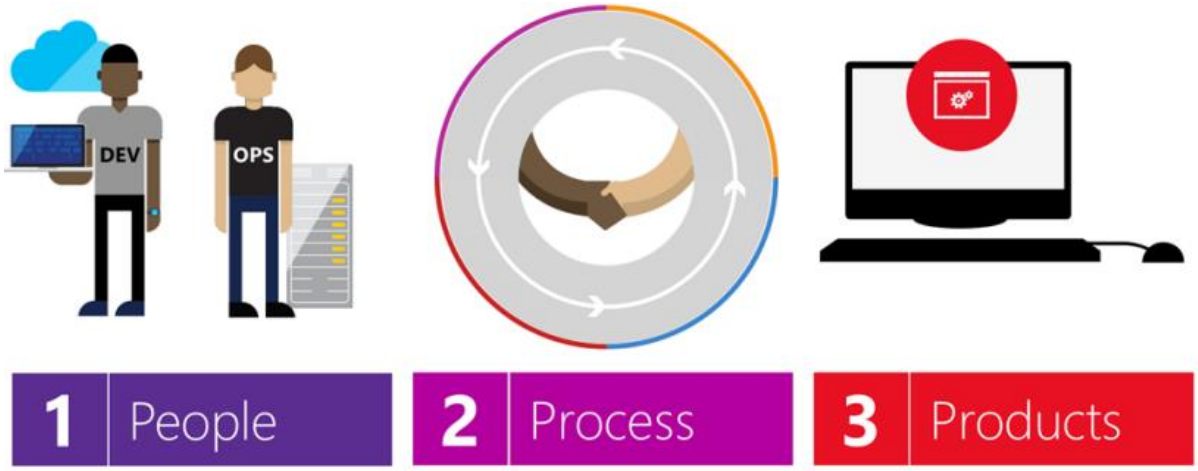
DevOps zoekt de samenwerking op van development en operations teams.

DevOps implementaties gebruiken automatisatie tools, die een stijgend programmeerbaar en dynamisme infrastructuur creëren ten opzichte van een life cycle perspectief.

DevOps is geen product, maar eerder een cultuur van mensen die een proces volgen dat mogelijk gemaakt wordt door producten om zo meer waarde te kunnen leveren aan de end user.

DevOps is de samenwerking van mens, proces en product om continuïteit te kunnen garanderen aan de end user.

DevOps is een ‘three stage conversation’



E-book

Wat is DevOps?

- Het woord DevOps werd voor het eerst gebruikt in 2009 door Patrick Debois
- Het is geen proces, technologie of standard. Eerder een cultuur.
- De term 'DevOps movement'
    - Wordt gebruikt als het praten over adaptation rates en trends.
- De term 'DevOps environment'
    - Wordt gebruikt als we praten IT-organisaties die een DevOps cultuur aanneemt.

Waar komt DevOps vandaan?

2 primaire grondleggers

Enterprise Systems Management(ESM)

- Systeem administrators
- Brachten ESM best practices naar DevOps
    - Configuration management
    - System monitoring
    - Automated provisioning
    - Toolchain approach

Agile Development

- DevOps is een extensie van Agile principes dat verder gaat als de code en service.
- Agile schrijft een goeie samenwerking voor met klanten, product management, developers en QA om sneller te itereren naar een beter product.
- Service delivery en de interactie tussen app en systemen is een fundamenteel deel van de waarde naar een klant toe.

DevOps in a nutshell: High-Quality software, snellere releases, gelukkigere klanten.

Problemen met DevOps?

Developers

- Willen sneller en sneller code pushen

Operations

- Willen een stabiel systeem

➔ Dilemma

Oplossing?

- het opstellen van verwachtingen en prioriteiten.
- Samenwerking binnen en tussen teams voor 'problem solving'.
- Het automatiseren van herhalende processen om meer tijd en 'higher-level work' te creëren.
- Feedback integreren
- Deze data delen met iedereen die betrokken is in het project.

Hoe werkt DevOps

- Collaboration
    - Niet enkel Dev en Ops, maar alle teams. Zoals testing, product management, executives,…
- Automation
    - Tools
    - Hangt af van toolchains om grote delen van end-to-end software development en deployment processen te automatiseren.
- Continuous Integration
    - Het continue integreren van code
    - Blijft source code updates van het team mergen in een shared mainline.
    - Continual merging voorkomt catastrofale merge conflicts
- Continuous Testing
    - Het continue automatisch testen van code
    - Developers ➔ test data sets
    - QA ➔ Automation test cases
    - Ops ➔ Monitoring tools, Test environments
    - Elimineert “bottlenecks”
- Continuous Delivery
    - Het continue afleveren van “releases”
    - Verbeterd release frequentie.
    - Elke update is kleiner en kleiner, de kans dat een update de release doet falen daalt met elke release.
- Continuous Monitoring
    - Het continue vinden en fixen van failures
    - Server monitoring
    - Application Performance monitoring

Wie neemt DevOps aan?

74% van IT-organisaties implementeert DevOps op een of andere manier.

Waarom DevOps?

- Developers
    - Automatisatie ➔ Big win
    - Geen papierwerk, geen approval cycles… geen verloren tijd.
    - Meer creativiteit, meer opties, verschillende scenario’s, beter testen.
    - Snellere probleemoplossing
- Operations
    - Developers verbeteren system stability
    - Lager risico van catastrofistisch falen
    - Automation elimineert human errors
- Test Engineers
    - Test environment dat identiek is aan productie environment
    - Gemakkelijker om performantie te voorspellen van releases.
- Product Manager
    - Snellere feedback
    - Meer responsivenes
    - Verminderd risico

- Executives
    - Helpt organisatie met het leveren van high-quality producten
    - High-Quality personeel.

2. Desgin Patterns

Richtlijnen BB

Schriftelijk, gesloten boek

Een Design Pattern kunnen definiëren: wat is het? UML-klassendiagram tekenen en de verschillende klassen verklaren.

Een Design Pattern kunnen herkennen aan de hand van voorbeeldcode en/of een voorbeeld UML diagram.

Aan de hand van een probleembeschrijving een oordeelkundige keuze kunnen maken tussen Design Patterns: welke zou je hier het best toepassen en waarom?

Praktisch, laptop (C#)

Een oefening waarbij via een probleembeschrijving een Design Pattern dient gekozen (zie voorbeeldvraag)

Voorbeeldcode herwerken (refactoren) tot een (eventueel opgelegde) Design Pattern gebruikt wordt. Bijvoorbeeld: herwerk dit switch-statement zodat je een Command patroon gebruikt.

Design Principes

"Indentify the aspects of your application that vary and separate them from what stays the same."

"Program to an interface, Not an implementation."
Interfaces zijn contracten of signaturen zijn en niks weten van implementaties

"Favor composition over inheritance."

"Strive for loosely coupled designs between objects that interact."

Note: Voor uit gecodeerde voorbeelden, zie bijlage DesignPatterns.sln

http://www.dofactory.com/net/design-patterns

Strategy

Behavioral Pattern

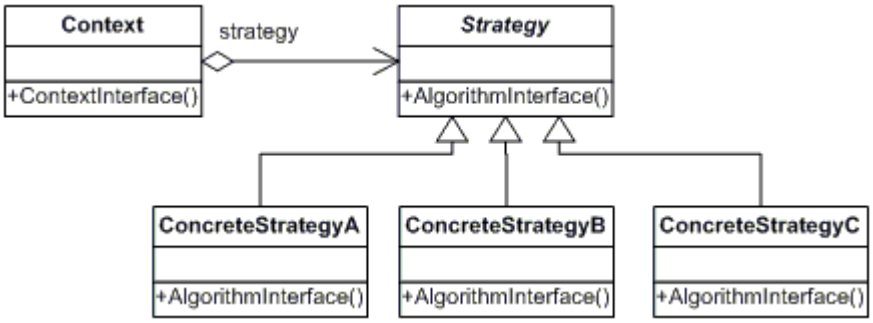
Verschil met Template Method:

Run -time algoritme selectie door containment.

Definitie

Definieert een familie van algoritmes, omhult deze en maakt ze uitwisselbaar. Met het strategy pattern kan je algoritmes onafhankelijk maken van de clients die ze gebruiken.

UML class diagram



- Strategy (SortStrategy.cs)
    - Definieert een gemeenschappelijke interface naar alle algoritmes. Context gebruikt deze interface om het algoritme aan te roepen dat gedefinieerd is door ConcreteStrategy
- ConcreteStrategy (Quicksort.cs, ShellSort.cs, MergeSort.cs)
    - Implementeert het algoritme gebruik makende van het Strategy Interface.
- Context (SortedList.cs)
    - Is geconfigureerd met een ConcreteStrategy Object
    - Heeft een reference naar het Strategy object.
    - Kan een interface definiëren wat ervoor zorgt dat Strategy aan zijn data kan.

Observer

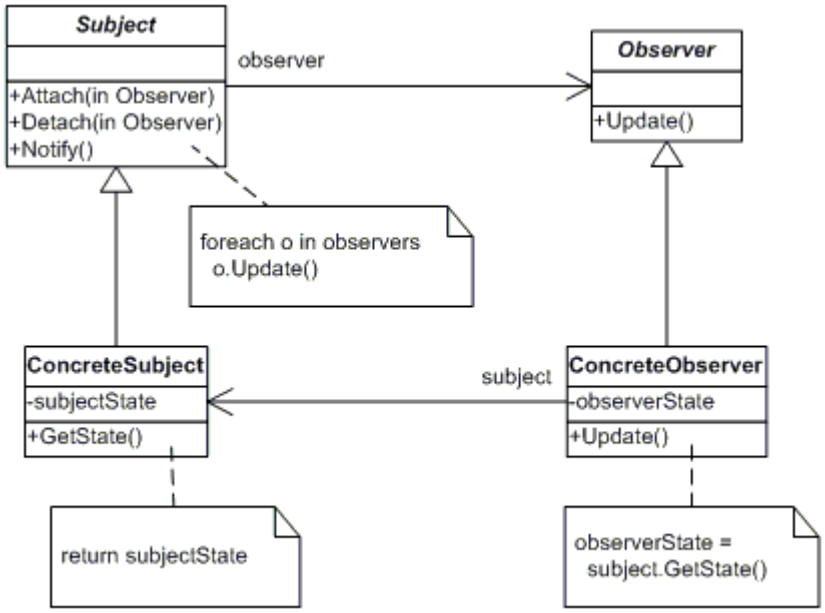
Behavioral Pattern

Definitie

Definieert een one-to-many dependency tussen objecten. Wanneer de staat van een object verandert, worden alle dependants automatisch verwittigd en geüpdatet. (PubSub) (Events, Delegates…)

Loose Coupling (➔ Veranderingen in Subject heeft geen effect op Observers en vise versa, Subject weet enkel dat Observer een interface implementeert, Subject moet niet veranderen bij het toevoegen van observers, observers kunnen altijd toegevoegd worden en het herbruiken van de 2 objecten is onafhankelijk van elkaar)

UML class diagram



- Subject (Stock.cs)
  - Kent de observers. X aantal Observer objecten kunnen subject observeren.
- ConcreteSubject (IBM.cs)
  - Steekt relevante state in ConcreteObserver.
  - Stuurt notificatie naar zijn observers als zijn state veranderd.
- Observer (IInvestor.cs)
  - Definieert een updatende interface voor objecten die moeten verwittigd worden als een subject veranderd
- ConcreteObserver (Investor.cs)
  - Houdt een reference naar een ConcreteSubject object
  - Houdt state bij dat consistent moet blijven bij de subjects
  - Implementeert de Observer (updatende interface) om zijn status consistent te houden met de subjects.

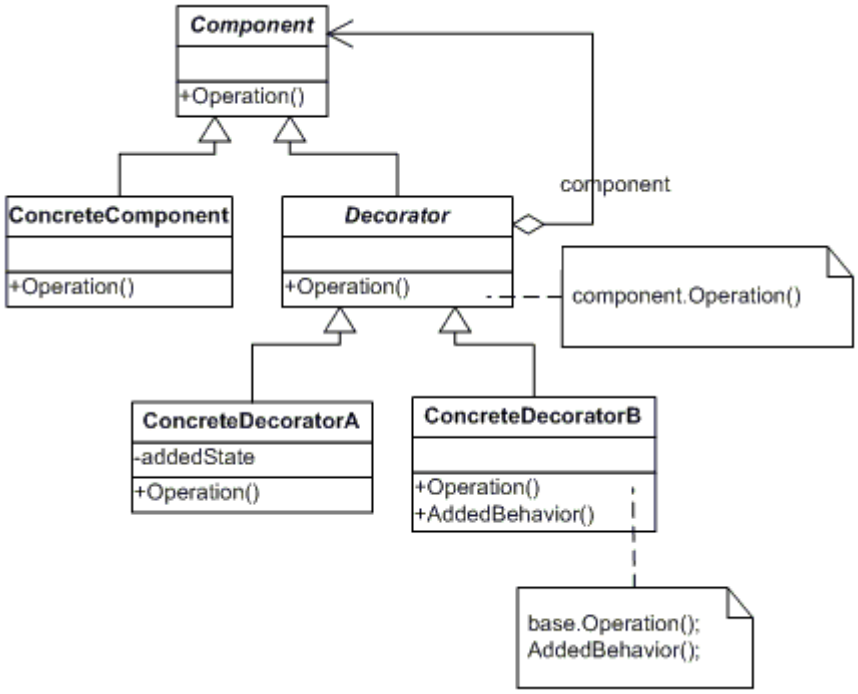
Decorator

Structural Pattern

Definitie

Het dynamisch toevoegen van verantwoordelijkheden aan een object. Een decorator zorgt voor een flexiebel alternatief voor subclassing om functionaliteit te extenden.

UML class diagram



- Component (LibraryItem.cs)
    - Definieert de interface voor objecten die dynamisch extra functionaliteit willen ontvangen.
- ConcreteComponent (Book.cs, Video.cs)
    - Definieert een object waar extra functionaliteit aan kan gehecht worden.
- Decorator (Decorator.cs)
    - Houdt een referentie naar een Component object en definieert een interface dat overeenkomt met de interface van deze component.
- ConcreteDecorator (Borroable.cs)
    - Voegt functionaliteit toe aan component.

Abstract Factory

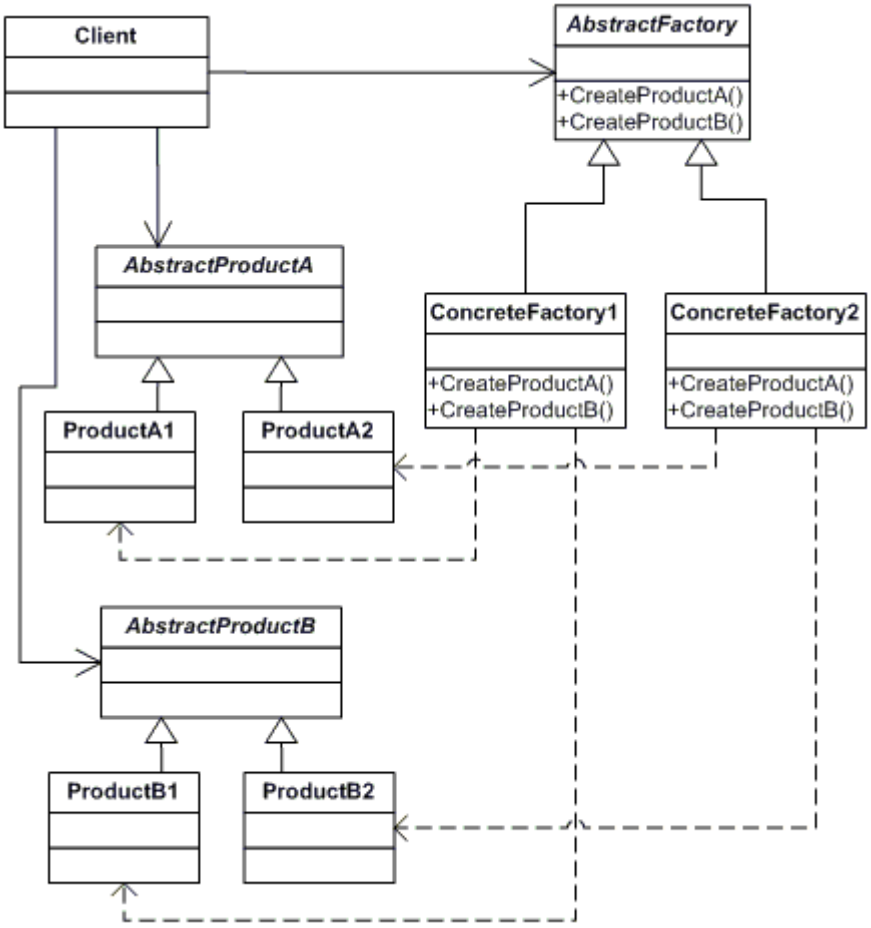
Creational Pattern

(Kan zijn dat je deze niet moet kennen, heb ze er toch maar bijgezet voor de zekerheid)

Definitie

Voorziet een interface voor het creëren van families van gerelateerde of dependant object zonder hun concrete klassen te specifiëren

UML class diagram



- AbstractFactory (ContinentFactory.cs)
  - Declareert een interface voor operaties die een abstract procuct maken.
- ConcreteFactory (AfricaFactory.cs, AmericaFactory.cs)
  - Implementeerd de operatie om concrete product objecten te creëren.
- AbstractProduct (Herbivore.cs, Carnivore.cs)
  - Declareerd een interface voor een type van product object.
- Product (Wildebeest.cs, Lion.cs, Bison.cs, Wolf.cs)
  - Definieert een product object dat gemaakt wordt door het corresponderend concrete factory.
  - Implementeerd de AbstractProduct interface.
- Client (AnimalWorld.cs)
  - Gebruikt interfaces die gedeclareerd zijn door de AbstractFactory en AbstractProduct klassen.

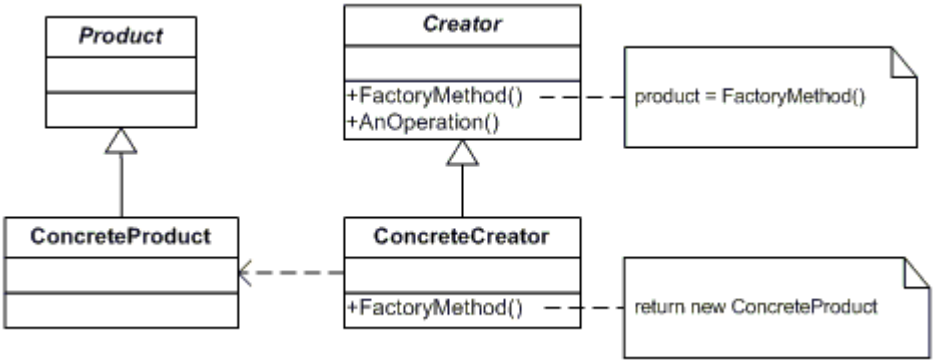
Factory Method

Creational Pattern

Definitie

Definieert een interface voor het creëren van een object, maar laat subklasse bepalen welke klasse er geïnstantieerd wordt. De Factory Method laat een klasse de instantiatie overdragen naar zijn subklassen.

UML class diagram



- Product (Page.cs)
  - Definieert de interface van objecten die gemaakt worden door de factory method.
- ConcreteProduct (Skillspage.cs, EducationPage.cs, ExperiencePage.cs,…)
  - Implementeerd de Product interface.
- Creator (Document.cs)
  - Declareert de factory method. Deze zal een object van het type Product terug geven. De Creator mag ook een default implementatie va.n de factory method definiëren die een default ConcreteProduct object returnt
- ConcreteCreator (Report.cs, Resume.cs)
  - Overschrijft de factory method om een instantie van een ConcreteProduct te returnen.

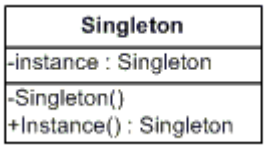
Singleton

Creational Pattern

Definitie

Een class waarvan maar 1 instantie van kan bestaan en waar globaal access naartoe is.

UML class diagram



- Singleton (LoadBalancer.cs)
    - Definieert een instantie waarvan clients deze unieke instantie kunnen aanroepen. Deze instantie is een class operation.
    - Verantwoordelijk voor het aanmaken en behouden van zijn eigen unieke instantie.

Command

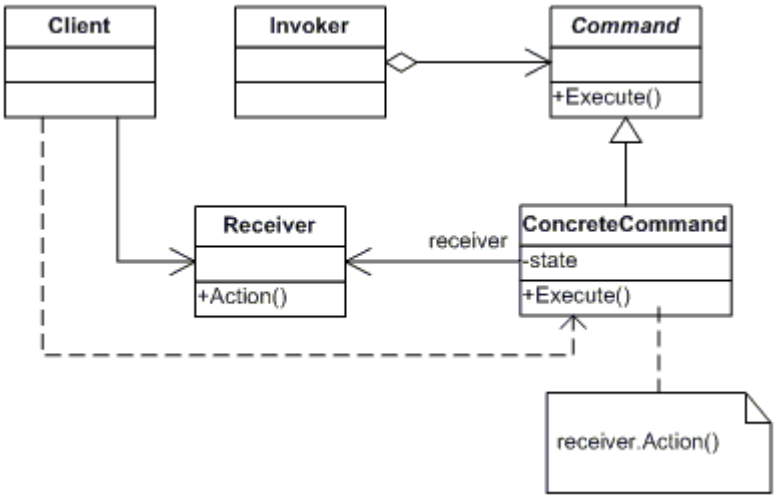
Behavioral Pattern

Definitie

Encapsuleert een request als een object, waarmee je clients kan parametizeren met verschillende requests, queue of log request. En unduable operations kan supporten

(MVVM commands, …)

UML class diagram



- Command (Command.cs)
  - Declareert een interface voor het uitvoeren van een operatie.
- ConcreteCommand (CalculatorCommand.cs)
  - Definiëert een binding tussen een receiver object en een actie.
  - Implementeerd Execute door de corresponderende operatie te invoken op de Receiver.
- Client (Program.cs)
  - Creëert een ConcreteCommand object en set zijjn receiver.
- Invoker (User.cs)
  - Vraagt het commando op om het request uit te voeren.
- Receiver (Calculator.cs)
  - Weet hoe hij de operaties moet uitvoeren die geassocieert zijn met het uitvoeren van de request.

Template Method

Behavioral Pattern

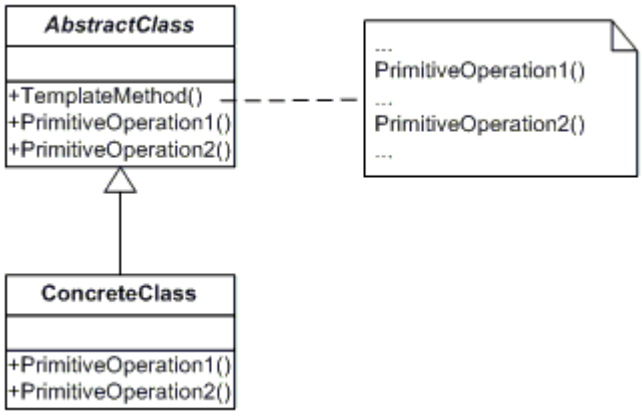
Verschil met Strategy:

Compile-time algoritme selectie door sub classing.

Definitie

Definieert een skelet van een algoritme in een operatie, waarbij sommige stappen worden overgedragen naar subklassen. Template Method laat subklassen bepaalde stappen van een algoritme herdefiniëren zonder de structuur van dit algoritme te veranderen.

UML class diagram



- AbstractClass (DataExporter.cs)
    - Definieert een primitief abstractie operatie dat concrete subclasses definieert om de stappen van een algoritme te implementeren.
    - Implementeert een template method die een skelet van een algoritme definieert. De template method roep primitieve operaties aan, en zowel operaties die gedefiniëert zijn in AbstractClass of die van andere objecten.
- ConcreteClass (PDFExporter.cs, ExcelExporter.cs)
    - Implementeerd de primitieve operaties die de subclass-specifieke stappen van het algoritme uitvoeren.

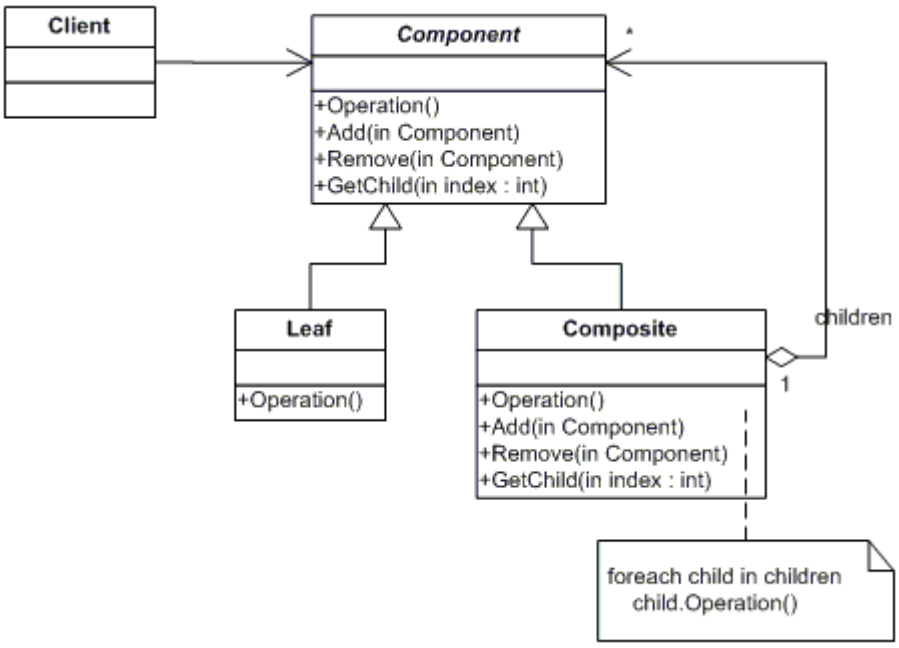
Composite

Structural pattern

Definitie

Schikt objecten in boomstructuren om “part-whole hierarchies” te representeren. Composite laat clients elk individueel object en compositie van objecten uniform behandelen.

UML class diagram



- Component (DrawingElement.cs)
  - Declareert de interface voor objecten in de compositie
  - Implementeert een default behavior voor de interface gemeenschappelijk aan alle classes.
  - Declareert een interface voor de toegang naar en het managen van child components.
  - (Optioneel) Definieert een interface voor de toegang naar een component zijn parent
- Leaf (PrimitiveElement.cs)
  - Representeert Leaf objecten in de compositie, een leaf heeft geen child objecten.
  - Definieert een behavior voor primitieve objecten in de compositie.
- Compostie (CompositeElement.cs)
  - Definiëert een behaviour voor componenten die child objects hebben.
  - Houdt child components bij
  - Implementeerd child-related operaties in de Component interface.
- Client (CompositeApp.cs)
  - Manipuleert objecten in de composite vanuit het de Component interface.

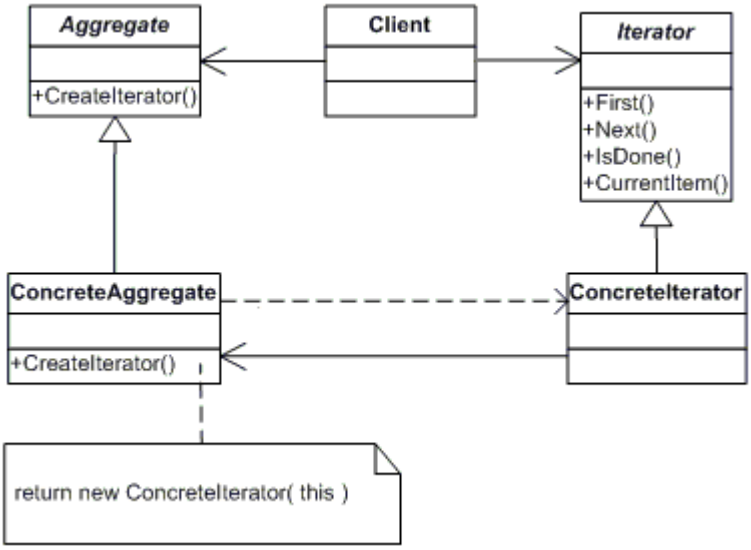
Iterator

Behavioral Pattern

Definitie

Stelt een manier voor om toegang te verkrijgen tot de elementen van een aggregate object. Dit sequentieel en zonder de onderliggende representatie bloot te stellen.

UML class diagram



- Iterator (AbstractIterator.cs)
  - Definieert een interface voor het toegang krijgen tot een element.
- ConcreteIterator (Iterator.cs)
  - Implementeert de iterator interface
  - Houdt de huidige positie bij van het aggregate object.
- Aggregate (AbstractCollection.cs)
  - Definieert een interface voor het aanmaken van een Iterator object
- ConcreteAggregate (Collection.cs)
  - Implementeert de iterator creation interface om een instantie van de juiste ConcreteIterator terug te geven

3. Git

Richtlijnen BB

Schriftelijk, gesloten boek

Kort kunnen situeren waarom git bedacht is

Git principes kunnen uitleggen: staging, commit, push, pull, merge, rebase, etc.

Kunnen schetsen welke workflows voor teams mogelijk zijn als je git gebruikt

Praktisch, laptop

Op het examen zou je een git repo gegeven kunnen hebben, waarover je een aantal vragen moet kunnen beantwoorden. Hierbij zal je dagdagelijkse git commando's moeten gebruiken (je moet het gebruikte commando noteren). Bijvoorbeeld:

- Hoeveel branches zitten in deze repo?
- Hoeveel commits tel je?
- Wie heeft de meest recente commit gedaan?
- Merge branch A en branch B, wat voor soort merge was dit?
- enz.

Design goals

- Speed
- Simplicity
- Strong branch/merge support
- Distributed
- Scales well for large projects

Why distributed

- Reliable branching/merging
    - Feauture branches
    - Altijd met Version Control
    - Fixes toepassen op meerdere branches
- Full local history
    - Snel lokaal repository statistieken genereren:
    - Analyze regressions: opzoeken wanneer een bug ontstond
- New ideas
    - Deployment

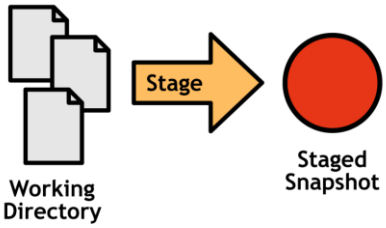
Working locally

Working Directory

- Dit is (lokaal) waar je bestanden aanpast, code compileert en in het algemeen het project ontwikkelt

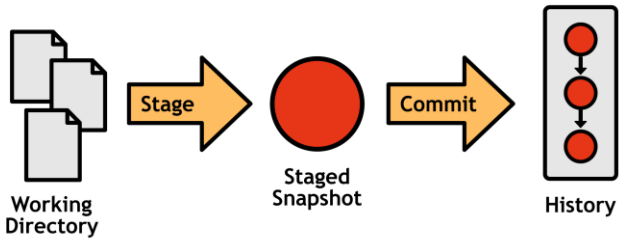
Staging Area

- Een tussenstap tussen de working directory (lokaal) en de project geschiedenis
- Ipv te forceren alle aanpassingen onmiddellijk te committen, laat git u deze groeperen in changesets
- Staged changes zijn nog geen onderdeel van de project history



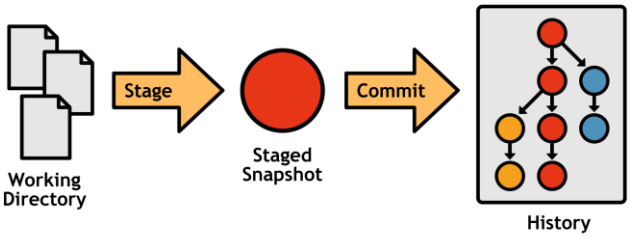
Committed history

- Na de changes in de staging area te steken kan je committen naar de project history waar het zal blijven als een “safe” revision
- Safe wilt zeggen dat git ze nooit zelf zal aanpassen, maar je kan wel zelf de project history aanpassen



Development branches

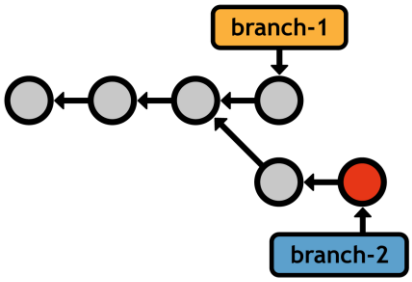
- Branches maken het mogelijk om onafhankelijke features parallel te maken door de project history te vertakken
- Git branches zijn niet zoals branches van een gecentraliseerd versie control systeem.
    - Simple to make
    - Simple to merge
    - Easy to share
    - Worden dus voor alles gebruikt



Branch, merge rebase

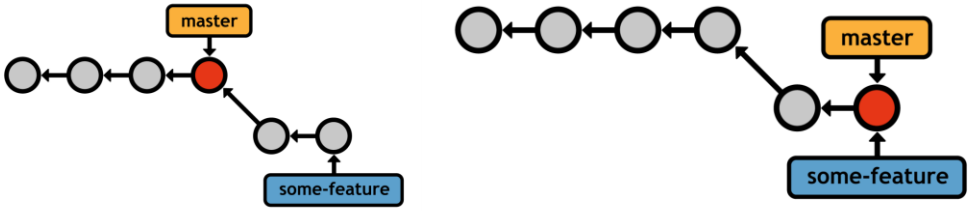
Branches

- Vermenigvuldigen de basis functionaliteit aangeboden door commits, door gebruikers toe te staan hun geschiedenis te vertakken
- Is verwant aan een nieuwe ontwikkel omgeving aanvragen, volledig met geisoleerde working directory, staging area en project history
- Non-linear workflow: de mogelijkheid om ongerelateerde features in parallel te ontwikkelen
- Git branches werken op commit level, een branch is dus gewoon een pointer naar een commit.



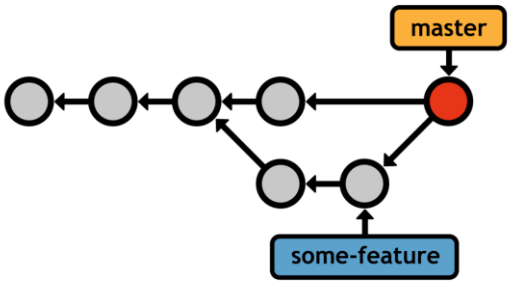
Fast-forward merge

- Je maakt een branch om een nieuwe feature te ontwikkelen, die klaar is om geintegreerd te worden in de main code base
- Ipv de 2 missing commits van de master te herschrijven, zal git de master branch zijn pointer “fast-forwarden” naar de locatie van “some feature”
- Na de merge heeft de master branch alle gewenste geschiedenis
- Je had hier de feature op de branch kunnen ontwikkelen, maar door een aparte branch te maken creeer je een veilige omgeving om te experimenteren. Als het fout gaat delete je gewoon de branch, and no harm is done



3-way merge

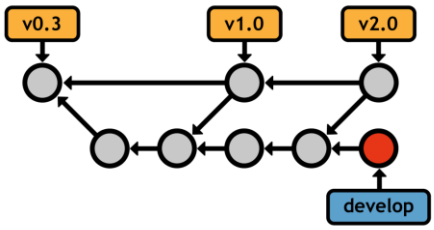
- Je hebt nog aanpassingen gedaan op de master branch voor je de feature wilt mergen
- De feature branch nu mergen in de master branch resulteert in een 3-way merge
- Git genereert een nieuwe merge commit die een gecombineerde snapshot voorstelt van beide branches
- Deze nieuwe commit heeft 2 parent commits, waardoor het toegang heeft tot beide geschiedenissen.



Branching Workflows

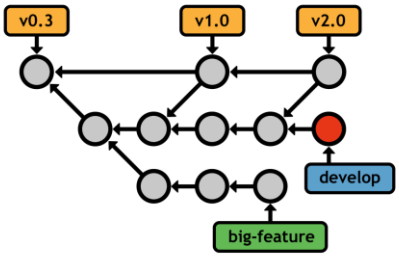
Permanent Branches

- De master branch wordt enkel gebruikt voor stabiele code, je commit nooit rechtstreeks op de master branche
- Master branche is een integratie branch voor afgewerkte features die gemaakt zijn in hun eigen branche
- Hier bovenop wordt soms nog een extra integratie branch gemaakt voor een extra laag van abstractie zoals in onderstaand voorbeeld. De master branch wordt dan enkel gebruikt voor releases, en de develop branch wordt gebruikt als interne integratie branch



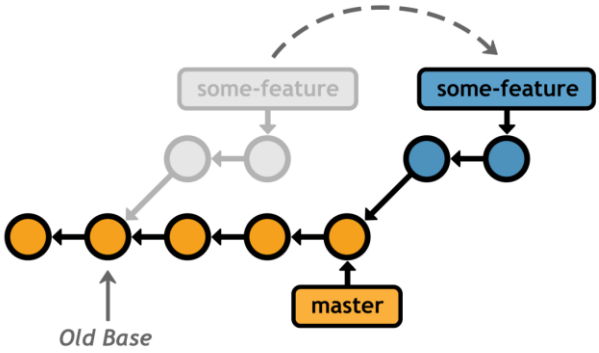
Topic branches

- Tijdelijke branches die een nieuwe feature afscheiden, en hierdoor het main project beschermen van untested code
- Stammen meestal af van een andere feature branch of de integratie branch, maar nooit de “super stable” of master branch



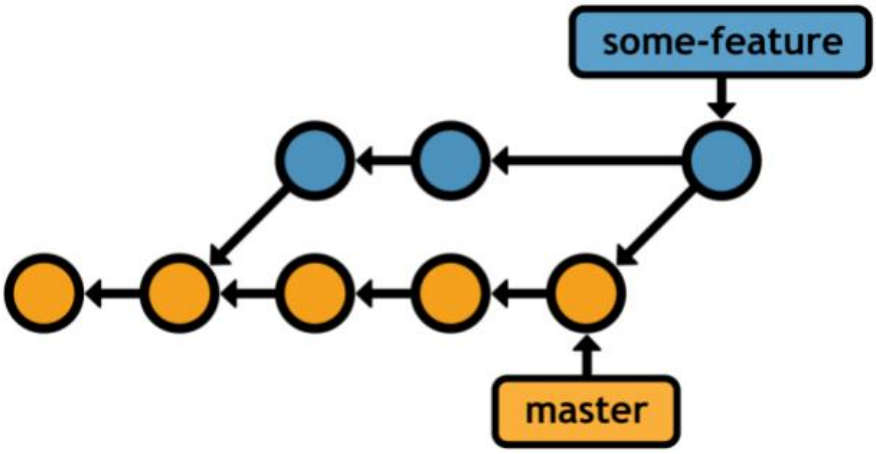
Rebasing

- Het proces van het verplaatsen van een branch naar een “new base”
- Na de rebase is de feature branch een lineaire extensie van master
- Dit is een mooiere manier om veranderingen van 1 branch in een andere te integrereren (zeker in vergelijking met 3way merge)



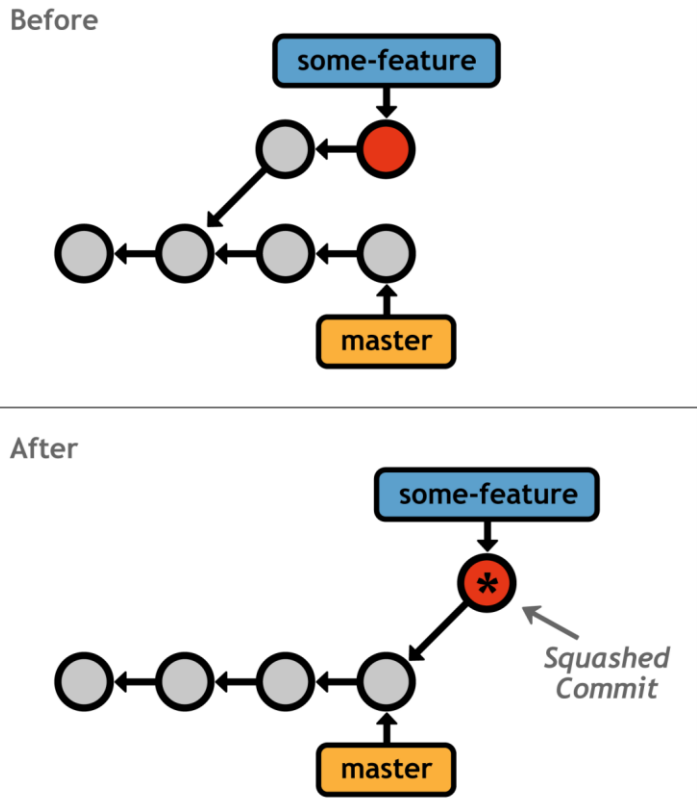
Rebasing vs 3 way commit

Vergelijk...



Interactive Rebasing

- Staat u toe om volledig te branch geschiedenis te herschrijven naar uw specificaties
- Je kan zoveel tussentijdse commits doen als je wilt, achteraf herwerk je ze in 1 commit

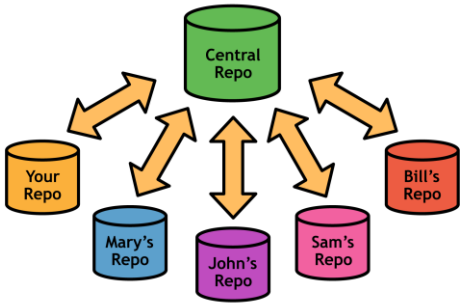


NOOIT EEN BRANCH REBASEN DIE GEPUSHED IS NAAR EEN PUBLIEKE REPOSITORY

Working remotely

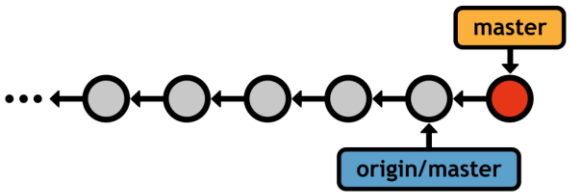
Centralized workflow

- Meest geschikt voor kleine teams waar elke ontwikkelaar write access heeft op de repository
- Staat samenwerking toe door gebruik te maken van 1 centrale repository
- Individuele ontwikkelaars werken in hun eigen lokale repository

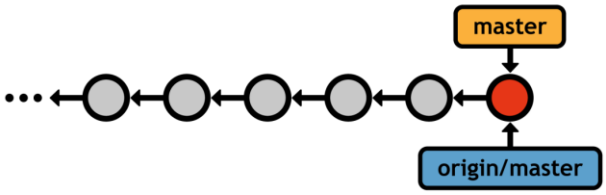


- Als ze een feature afhebben, kunnen ze het integreren in hun lokale master, en pushen naar de centrale repository (origin)

Mary's Repository, Before Pushing



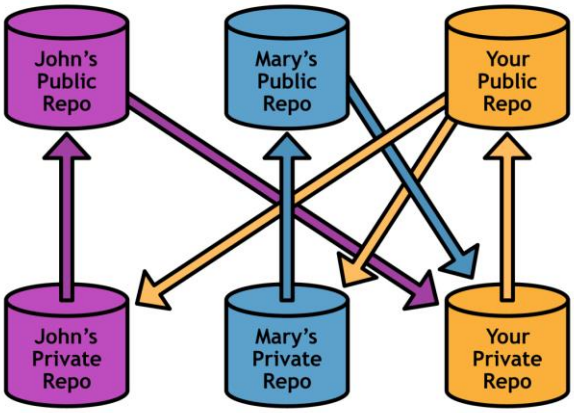
Mary's Repository, After Pushing



- Daarna kan iedereen de nieuwe commits fetchen en opnemen in hun lokaal project
- Nadeel: Je kan niet met 2 tegelijk committen

Integrator workflow

- Individuele gebruikers hebben zowel een private, als een publieke repository
- De bijdrager pushed zijn veranderingen naar zijn eigen publieke repo. Dan kan men deze pullen naar hun eigen private repository, om te controleren. Als je zijn bijdrages goed vind merge je ze in een lokale branch en push je het naar de main repository (public).
- Je moet wel overeenkomen over een “officiele” repository, anders kan iedereen vrij snel “out of sync” geraken



Command Cheatsheet

Groene commands stonden in de slides, de rest is extra van een andere samenvatting.
!!!Commando’s van in het gastcollege zitten hier NIET in)!!!

Difference & Logging

```
Git log (inspect commit)
Git log –oneline –graph
Git log –format=short OF Git logshort
Git config –global alias.lga “log –graph –oneline –all –decorate”
Git diff <<eerste 6 charsa van commit>>..<<eerste 6 charsa van commit>>
Git diff HEAD~1..HEAD (laatste commit – 1 tot laatste commit het verschil zien)
Git tag (tag a commit)
```

Van en naar staging halen

```
Git add –u (voegt alleen updated files toe)
Git add –A (voegt ALLE files toe, ook niet tracked files)
Git add . (voegt gewoon alle tracked files toe)
Git checkout file1 (reset de changes van een file/map in u working copy)
Git checkout HEAD <<file>> Undo modified file, not staged
Git clean –f (get rid of untracked files)
```

Resetting

Git reset –hard (alles changes resetten in u working copy)
Git reset --soft HEAD~1 (reset de laatste commit + alles in stage staat er nog)
Git reset –hard HEAD~1 (reset de laatste commit + alles in stage verwijderd)
Git reset HEAD <<file>> (undo modified file, staged)
Git reset HEAD --hard (Undo stage and modified files)
Git reset HEAD~1 (Creates problematic commits in isolation)
Git revert <<commit id>> Creates a new commit with the undoing info

Werken met een remote repository

Git init (Initialiseren van git repository)
Git remote add origin <<url>>
Git remote rm origin
Git remote add <name> <url>
Git remote -v

Push & Pull

Git fetch (alle changes van de remote halen, kan aanvullen met naam, aka origin)
Git merge origin/master (alle changes ophalen en de local & remote branch mergen)
Git pull
Git push
Git push –tags (standaard pusht git geen tags)

Viewing

Git branch (toont alle branches, zelfde met tags)
Git branch –r (toont alle remote branches)
Git remote (toont original remote naam)
Git remote –v (toont push en fetch url)
Git reflog (toont alle posities van HEAD)

Branching

Git branch <<branchName>> (maakt een branch aan)
Git checkout <<branchName>> (switchen naar die branch)
Git branch <<branchName>> <<commitNr>> (een branch maken van een commit)
Git branch –m <<branchName1>> <<branchName2>> (rename van branch)
Git branch –d <<branchName>> (verwijder merged branch, -D is force delete)
Git checkout –b <<branchName>> (branch maken en switchen)
Git branch <<branchName>> <<reflogNr>> (terug halen van deleted branch)

Stashing

Git stash (zet alles van working copy naar een stash en huidige wc reset)

Config

Git config –system (system-lvl)
Git config –global (user-lvl)
Git config (repository lvl)

Amending

Git commit –amend (Add current stage to previous commit)

4. Solid

Richtlijnen BB

Schriftelijk, gesloten boek

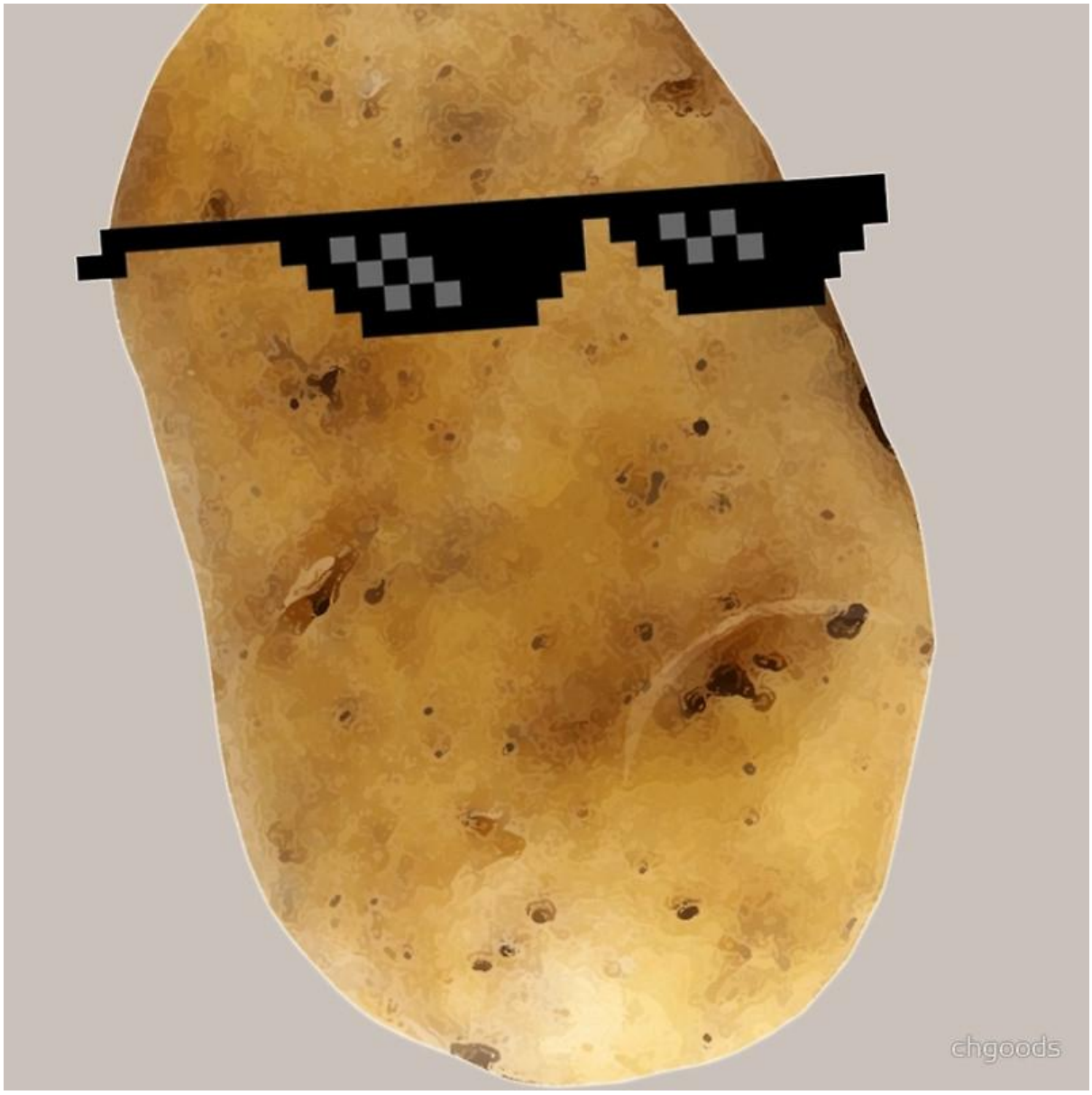
De SOLID principes kunnen opsommen en kort beschrijven

Op basis van gedrukte broncode op papier aangeven welk van de SOLID principes geschonden zijn en hoe je dit kan oplossen

Praktisch, laptop (C#)

Een oefening zoals de voorbeelden uit de cursus, waarbij een refactoring dient gemaakt te worden op basis van één van de SOLID-principes
(zie DesignPatterns.sln voor demos, heb het daar gewoon bijgezet.)

SRP	Single Responsibility	***Iedere methode, klasse of variabele moet maar één functie uitoefenen en dus maar één verantwoordelijkheid hebben.*** Dit heeft als gevolg dat door het wijzigen van deze methode, klasse of variabele maar op één enkele functie geraakt wordt. Zo zou een methode voor het toevoegen van een record aan de database, niet ook moeten zorgen voor het loggen van fouten. Dit moet in een aparte klasse/methode gebeuren.
OC	Open Closed	***Je klassen moeten open zijn voor uitbreiding van gedrag, maar gesloten voor het wijzigen van gedrag.*** Met dit principe in het achterhoofd kunnen je functies verrijkt worden, maar kan je er ook altijd vanuit gaan dat het basisgedrag functioneert zoals het bedoeld is. Overerving is de meest logische manier om dit toe te passen. Een voorbeeld hiervan is een interface Vorm welke de eigenschap 'Oppervlakte' definieert, verschillende vormen zullen deze interface implementeren en hun eigen berekening voor de oppervlakte implementeren zonder dat er iets veranderd aan de 'Vorm' interface.
LSP	Liskov Subsitution	***LSP zegt dat wanneer een klasse overerft van een andere klasse of een interface implementeert, je er vanuit moet kunnen gaan dat de klasse deze functies heeft.*** Het komt hier eigenlijk op neer: gebruik interfaces en zorg ervoor dat je ook naar de interfaces verwijst in plaats van naar de klassen. Een voorbeeld hiervan is het gebruik maken van een log interface, deze is dan altijd te vervangen door bijvoorbeeld een logger op de database of een logger naar tekstbestanden. Er hoeft dan niets veranderd te worden op de plaatsen waar de logger wordt aangesproken.
ISP	Interface Segregation	***ISP staat voor het scheiden van interfaces, zodat er nooit klassen zullen zijn die niet alle methoden binnen de interface implementeren.*** Maak bijvoorbeeld in plaats van een CRUD interface een aparte interface voor Create, Update en Delete. Hiermee voorkom je dat je methoden moet gaan implementeren welke je helemaal niet wil gebruiken. Bijvoorbeeld wanneer een entiteit niet verwijderd mag worden, deze zal dus niet de Delete methode uit de CRUD interface moeten implementeren.
DI	Dependency Inversion	***DIP staat voor het scheiden van abstractie niveaus binnen de code. Zorg ervoor dat complexe logica niet in dezelfde klasse staat als low-level code.*** Bijvoorbeeld het aansturen van hardware of het schrijven naar een log binnen in een berekening moet voorkomen worden. De berekenings logica kan dan beter een instantie van een log-interface aanspreken, waardoor de low-level code makkelijk te vervangen is of verbeterd kan worden. Hierdoor blijft de berekening zeker hetzelfde wanneer er van logger veranderd wordt.



Let's end this summary with le potato of good luck.