SWOP
2020

Wouter

June 24, 2020

1 Nota & wiki

Deze samenvatting is gebaseerd op het Larman boek, de pdfs van de proffen, samenvattingen van andere studenten, Wikipedia, StackOverflow en de Wina wiki.

Je moet voor dit vak heel wat design-principes/patterns kunnen toepassen, maar die worden niet (of in heel geringe mate) behandeld in de les. Alle theorie moet je kunnen toepassen vanaf deel 1 van het project. In het begin zul je dus veel zelfstudie moeten doen. Hier is een gids voor wat je best eerst bekijkt.

Meningen kunnen sterk verschillen, maar de volgende vier categorieen geven grofweg aan wat een goede volgorde zou kunnen zijn.

1. Essentieel (zeker toepassen of tenminste overwegen)
   - (a) GRASP
   - (b) Model-View-Controller (Prof. Jacobs is niet zo een fan, maar je moet zeker rekening houden met de bedoeling achter MVC)
   - (c) Composition-over-inheritance (niet hetzelfde als "Composite")
   - (d) Observer
   - (e) State
2. Nuttig, zeker aanbevolen om te bekijken
   - (a) Decorator
   - (b) Visitor
   - (c) Strategy
   - (d) Bridge
   - (e) Template method
   - (f) Builder
   - (g) Facade
   - (h) Chain-of-responsibility
3. Bestudeer deze eventueel later
   - (a) Mediator
   - (b) Factory method
   - (c) Prototype
   - (d) Singleton
   - (e) Adapter

(f) Composite
(g) Memento
(h) Command
(i) Proxy

4. Normaal gezien niet nodig

5. (a) Flyweight
   (b) Abstract Factory
   (c) Interpreter
   (d) Iterator

Noot: In het bijzonder categorie 2 en 3 lopen wat in elkaar over. Het is sterk afhankelijk van je project welke je het best toepast.

Lees kort de beschrijving op Wikipedia voor elk van de patterns en bestudeer in detail categorie 1 en liefst ook categorie 2 tijdens deel 1 van het project. Idealiter nog meer, natuurlijk. Zorg dat je naar het einde van deel 2 toe alle categorieen behalve de laatste kent.

De design-patterns vertellen je hoe goede ontwerpen er kunnen uitzien. Het is ook zinvol om te kijken naar slechte voorbeelden en een uitleg waarom ze slecht zijn. Die vind je in het boek "Refactoring: Improving the design of existing code". Daarin staan vaak gemaakte fouten en hoe je ze moet verbeteren. Bekijk die als je adviseur negatieve commentaar geeft over je code.

1.1 Corona-jaar 2020

Voor het examen van dit jaar werden volgende theorie documenten als te kennen aangegeven.

- GRASP-principes
- de ontwerppatronen uit "Some design patterns"
- de ontwerppatronen uit het Larman-boek
- de UML-notatie
- de leerstof over softwareontwikkelingsprocessen uit de slides
- de hoog-niveau-informatie uit inleiding-software-ontwerp.pdf

Dit is dan ook de focus van deze samenvatting.

1.2 Overzicht

Een overzichtje van Design Patterns kan HIER gevonden worden.

Contents

1 Nota & wiki **1**
1.1 Corona-jaar 2020 . . . 2
1.2 Overzicht . . . 2

2 GRASP **5**
2.1 Information Expert . . . 5
2.2 Creator . . . 5
2.3 Controller . . . 5
2.4 Low Coupling . . . 6
2.5 High Cohesion . . . 6
2.6 Polymorphism . . . 6
2.7 Pure Fabrication . . . 6
2.8 Indirection . . . 6
2.9 Protected Variations . . . 6

3 Some design patterns **6**
3.1 Observer . . . 6
3.2 Composite . . . 7
3.3 Iterator . . . 7
3.4 Factory Method . . . 7
3.5 Command . . . 7
3.6 Template Method . . . 7
3.7 Builder . . . 7
3.8 Strategy . . . 7
3.9 State . . . 7
3.10 Flyweight . . . 7

4 De ontwerppatronen uit het Larman-boek **8**
4.1 Adapter . . . 8
4.2 Abstract Factory . . . 8
4.3 Singleton . . . 8
4.4 Facade . . . 8
4.5 Prototype . . . 8
4.6 Chain of Responsibility . . . 8
4.7 Interpreter . . . 8
4.8 Mediator . . . 8
4.9 Memento . . . 9
4.10 Visitor . . . 9
4.11 Bridge . . . 9
4.12 Decorator . . . 9
4.13 Proxy . . . 9

5 UML-notificatie **11**
5.1 Sample Unified Process . . . 11
5.2 Class Diagram . . . 12
5.3 Use Case Diagram . . . 13
5.4 Sequence Diagram . . . 13
5.5 System Sequence Diagram . . . 14

6 Slides **14**
6.1 Software-ontwikkelingsproces . . . 14
6.2 Unified Process . . . 15
6.3 Analyse . . . 16
6.3.1 Use cases . . . 16
6.3.2 Domain model . . . 16
6.3.3 System sequence diagrams & operation contracts . . . 16
6.4 Interaction and class diagrams . . . 16
6.4.1 Interactiediagramma's . . . 16
6.4.2 Ontwerp-klassendiagrammas . . . 16
6.5 GRASP-Principes . . . 17
6.6 Testing . . . 17
6.7 Design patterns en refactoring: zie boek . . . 17

7 Hoog-niveau-informatie inleiding-software-ontwerp **17**
7.1 Iteratieve en incrementele ontwikkeling . . . 18
7.2 Kwaliteit van een software-ontwerp . . . 18
7.3 Rest . . . 18

2 GRASP

Bij het ontwerpen van samenwerkingen tussen objecten moeten we beslissen welke verantwoordelijkheden we toekennen aan welke objecten. Als vuistregels hiervoor zullen we 9 algemene principes of patronen zien voor het toekennen van verantwoordelijkheden in software (Eng.: General Responsibility Assignment Software Principles and Patterns ofwel GRASP), zoals voorgesteld door Larman.

1. Information Expert
2. Creator
3. Controller
4. High Cohesion
5. Low Coupling
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

Video-uitleg: deel 1 en deel 2.

2.1 Information Expert

General principle of object design and responsibility assignment.
Assign a responsibility to the information expert: the class that has the information necessary to fulfill the responsibility.

2.2 Creator

Alternative: Factory
Assign class B the responsibility to create an instance of class A if one of these is true:

1. B contains A
2. B aggregates A
3. B has the initializing data for A
4. B records A
5. B closely uses A

2.3 Controller

The first object beyond the UI layer which receives and coordinates a system operation.

Assign the responsibility to an object representing one of these choices:

1. Represents the overal "system", a "root object", a device that the software is running within, or a major subsystem. (= variations of a **facade controller**)
2. Represents a use case scenario within which the system operation occurs (a use-case or **session controller**)

2.4 Low Coupling

Reduces the impact of change. Evaluative.
Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
Benefits:

- Lower dependency between the classes.
- Change in one class having lower impact on other classes.
- Higher reuse potential.

2.5 High Cohesion

Keeps objects focused, understandable, and manageable, while supporting Low Coupling. Evaluative.
Assign responsibilities so that they are strongly related and highly focused. Use this to evaluate alternatives.
Break programs into classes and subsystems, so every (sub-)element is only focused on its own task.

2.6 Polymorphism

Handling type variety.
When related alternatives or behaviors vary by type (class), assign responsibility for the behavior (using polymorphic operations) to the types for which the behavior varies.

The user of the type should thus use polymorphic operations instead of explicit branching based on type.

2.7 Pure Fabrication

Alternative to Information Expert
A class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesian, and the reuse potential. Called a "service" in domain-driven design.
Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept (something made up).
Most GoF patterns are pure fabrications.

2.8 Indirection

Assigning responsibilities while avoiding direct coupling.
Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.
For example; the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view control pattern. This ensures that coupling between them remains low.

2.9 Protected Variations

Protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

3 Some design patterns

3.1 Observer

Define a one-to-many dependency between objects, when the object changes all its dependents will be notified.

Nonkel Derek

3.2 Composite

Applies when the program manipulates a tree of objects. When applying the pattern, a superclass is introduced that generalizes over both leaf nodes and non-leaf nodes of the tree, allowing both types of nodes to be treated uniformly whenever it makes sense.

Nonkel Derek

3.3 Iterator

Allows us to sequentially access components of an aggregate-object. Without knowing the details of that aggregate-object (independent of representation).

Nonkel Derek

3.4 Factory Method

Allows us to extract common knowledge into a reusable superclass. It allows a superclass to implement functionality that involves the creation of objects whose concrete type and other aspects need to be different for different subclasses. The creation of the objects is delegated to subclasses through the factory method.

Define an interface for the construction of an object, the subclasses specify the type of the object.

Nonkel Derek

3.5 Command

Encapsulate a request into a first-order-object, allows us to parameterise, save, undo, . . . the requests.

Nonkel Derek

3.6 Template Method

Define the skeleton of an algorithm with operations. These will be filled in at the subclass level. The subclasses can then change parts of the algorithm without changing the structure. Promotes reuse.

Nonkel Derek

3.7 Builder

Separate the construction and representation of a complex object, so the same constructionprocess can result in different representations (reusability).

Nonkel Derek

3.8 Strategy

Define and encapsulate a family of interchangeable algorithms to preserve their independence. A strategy allows us to dynamically adapt the algorithm, depending on the user.

Nonkel Derek

3.9 State

Separate out the logic for each state, moving the variables specific to a certain state to the corresponding state class.

Nonkel Derek

3.10 Flyweight

Reduces the space and time overhead of creating new objects to represent values, by reusing existing objects.

Share objects within a system, so large amounts of objects can be used efficiently and duplicates can be avoided.

Nonkel Derek

4 De ontwerppatronen uit het Larman-boek

Niet allemaal even interessant, maar kan handig zijn in je project.

4.1 Adapter

Lets two classes work together despite incompatible interfaces, by bridging the differences using a composition.
Nonkel Derek

4.2 Abstract Factory

Offer an interface for the creation of a family of related or dependent objects without specifying the concrete classes.
Nonkel Derek

4.3 Singleton

Ensures the existence of a single instance and offers global access to the object.
Nonkel Derek

4.4 Facade

Offer a general interface for a set of interfaces in a subsystem. Eases the use of a subsystem with a higher level interface.
Nonkel Derek

4.5 Prototype

Allows the adding of any subclass instance of a known superclass at runtime. Used when there are numerous potential classes that you want to only use if needed at runtime. Reduces the need for creating potentially unneeded subclasses.
⇒ Create objects by cloning prototypes.
Nonkel Derek

4.6 Chain of Responsibility

Avoid the chaining of the sender of a request and the receiver, by allowing multiple objects to answer the request. The request will move through the chain until an object handles it.
Nonkel Derek

4.7 Interpreter

Specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol, the syntax tree of a sentence is an instance of the composite pattern and is used to interpret the sentence.
Nonkel Derek

4.8 Mediator

Defining an object that abstracts how a set of objects interact. This keeps objects from explicitly pointing to each other, and allows dynamic changes to the interaction. Avoids high coupling between a group of coworking objects.
Nonkel Derek

4.9 Memento

Save the internal state (a snapshot) of an object, with the capability to revert to this state at a later time, all without breaking the encapsulation.
Nonkel Derek

4.10 Visitor

Allows you to add methods to classes of different types without much altering to those classes (independent of the structure). You can make completely different methods depending on the class used with this pattern.

- Adding methods: easy
- Adding structure: hard

Nonkel Derek

4.11 Bridge

Disconnects an abstraction from its implementation, so they can both vary independently.

- Dynamic choice of implementation
- edits to implementation don't require recompiling the abstraction
- independent expansion

Nonkel Derek

4.12 Decorator

Dynamically and transparantly add new responsibilities to an object. Flexibel alternative to subclassing for extended functionality.

- More flexible than inheritance
- "pay as you go"
- decorated object has different identity
- Startegy is preferably for heavy Component-class

Nonkel Derek

4.13 Proxy

A surrogate-object to control the access to another object.
Nonkel Derek

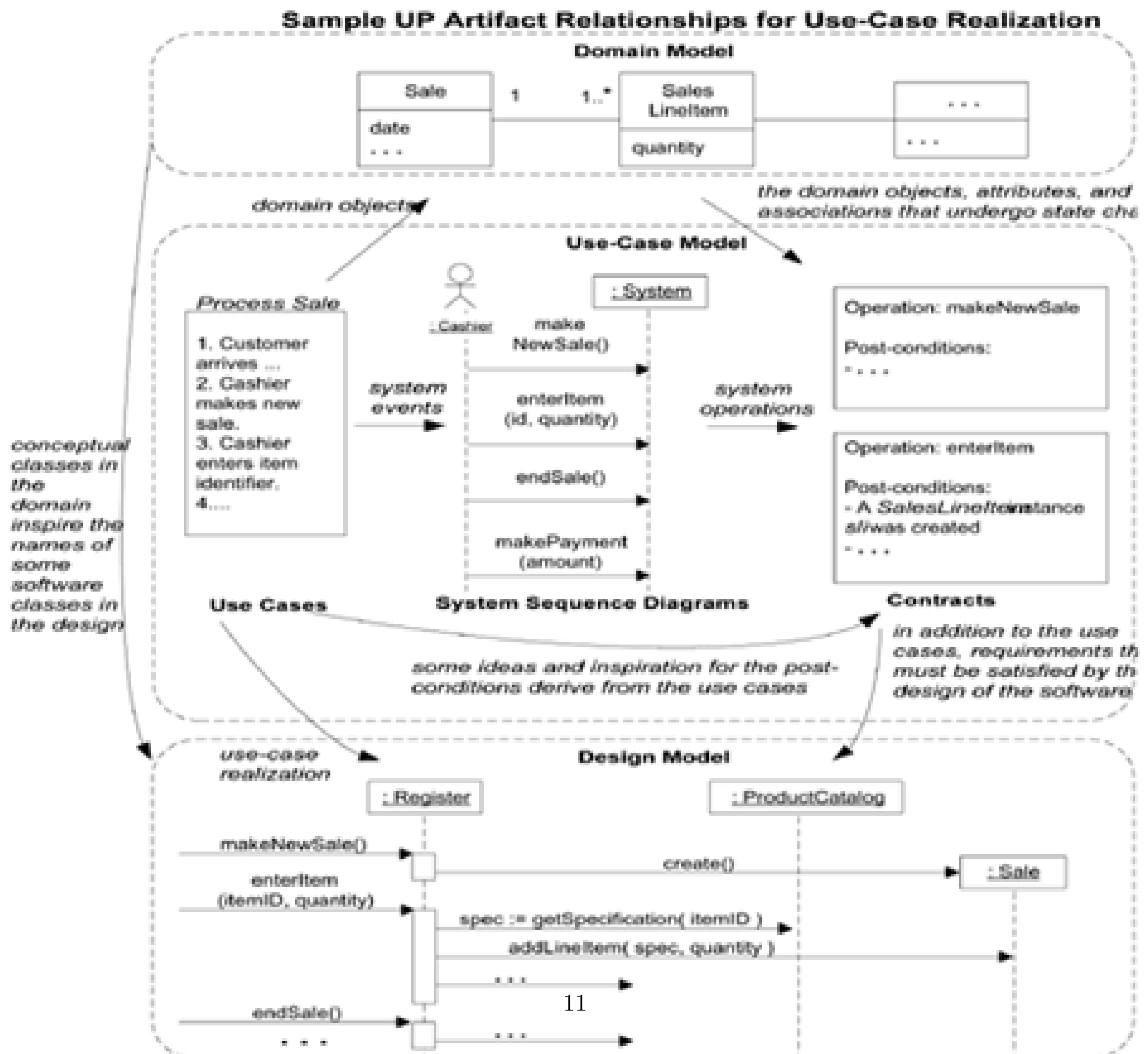
5 UML-notificatie

5.1 Sample Unified Process

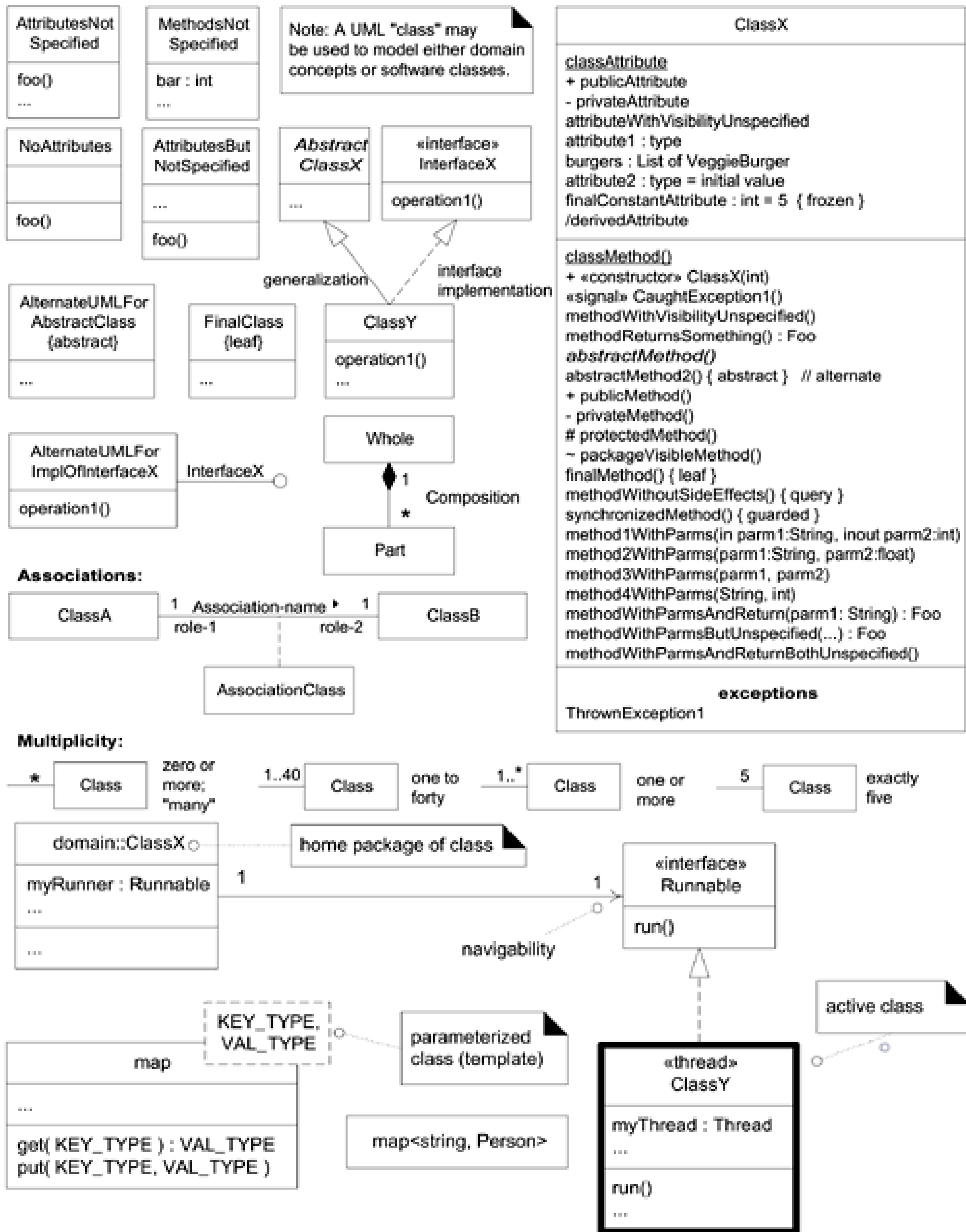
Sample Unified Process Artifacts and Timing (s-start; r-refine)

Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

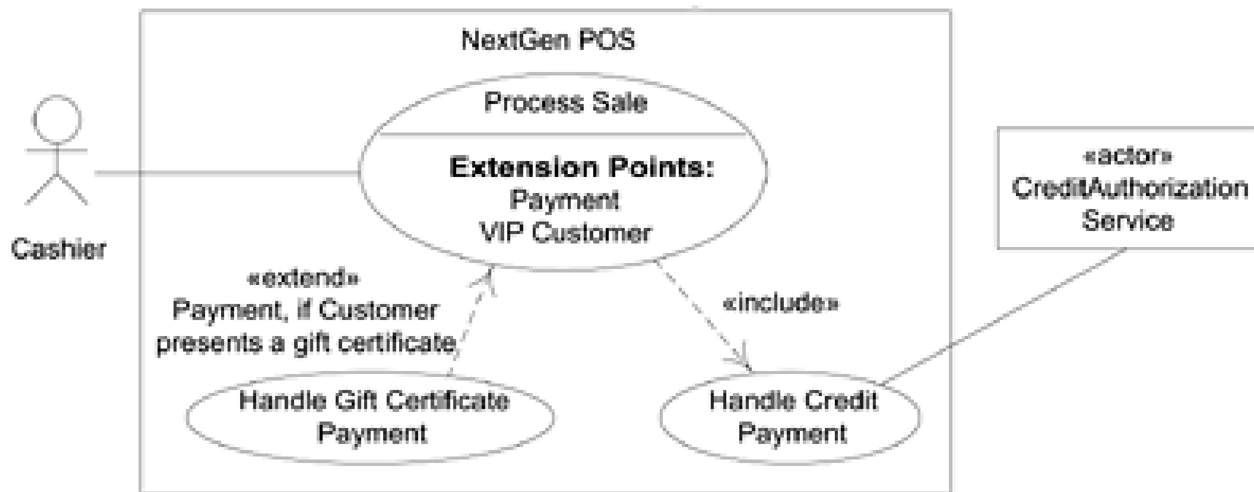
Sample Unified Process Artifact Relationships



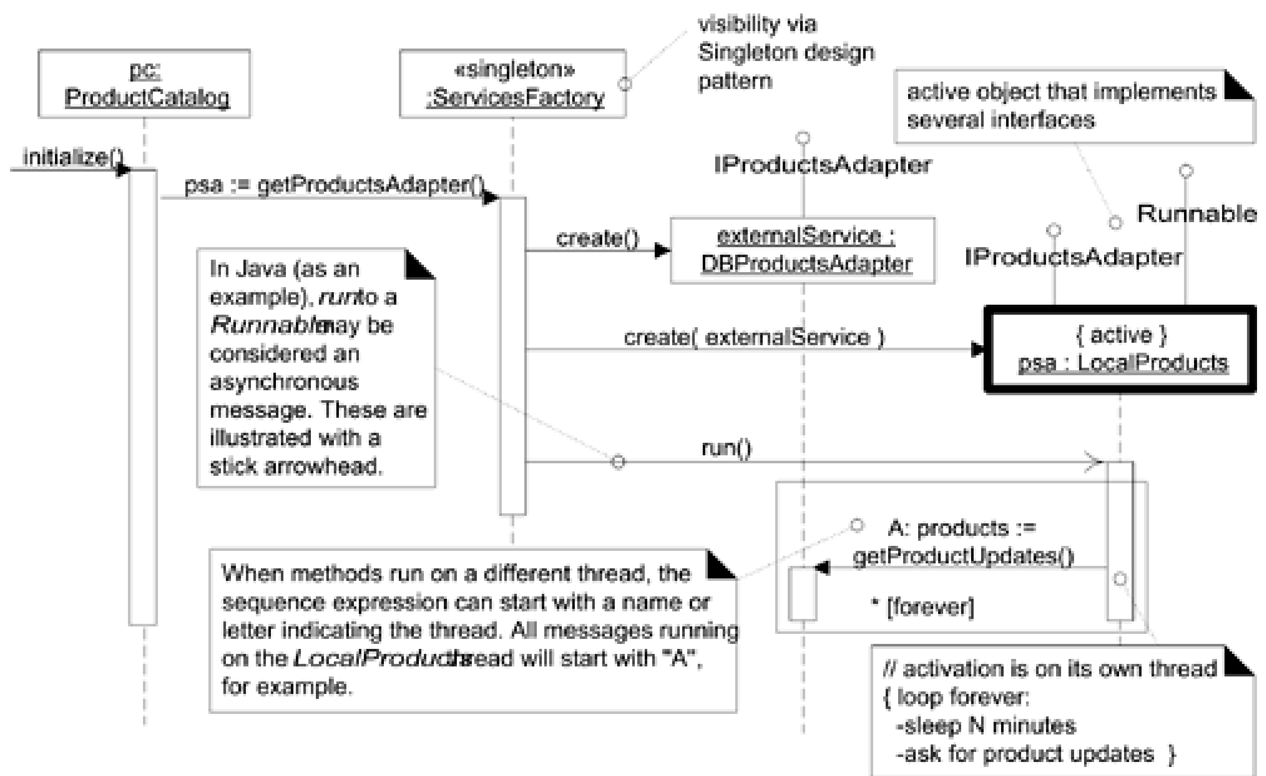
5.2 Class Diagram



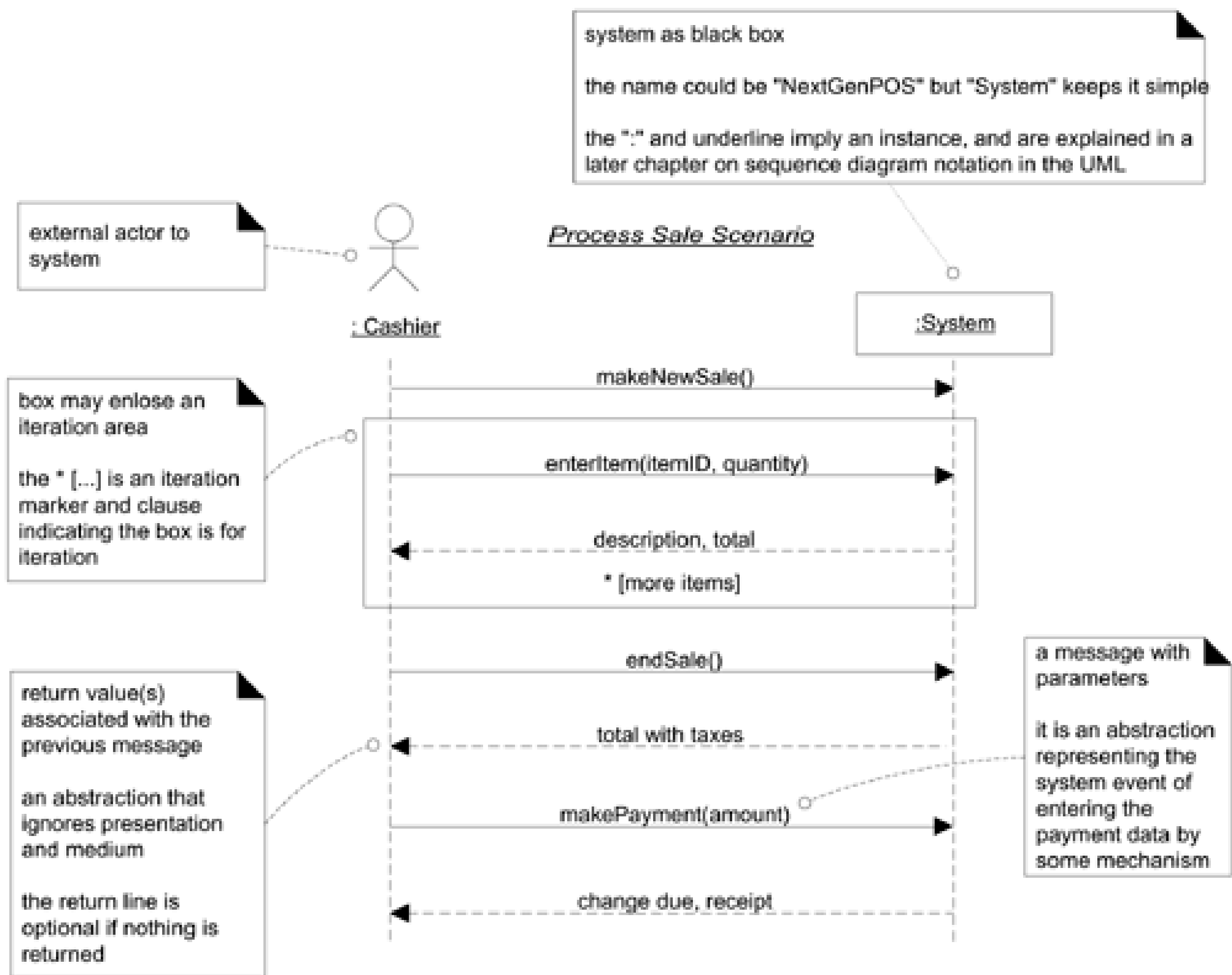
5.3 Use Case Diagram



5.4 Sequence Diagram



5.5 System Sequence Diagram



6 Slides

6.1 Software-ontwikkelingsproces

= een verzameling van partieel geordende activiteiten om software te bouwen, deployen en onderhouden.
Activiteiten:

- Business Modeling
- Requirements Analysis
- Design
- Implementation
- Testing
- ...

Afhankelijkheden tussen activiteiten:
Vereistenanalyse → Vereisten → Ontwerpen → Ontwerpbeslissingen → Implementatie → Code

Waterval-model: 1-na-1

Lightweight:

- Focus on working code rather than documentation
- Focus on direct communication
- Low management overhead

Heavyweight:

- Document driven
- Elaborate workflow definitions
- Many different roles
- Many checkpoints
- High management overhead
- Highly bureaucratic

Incremental model: go through process multiple times for small subprojects.
Evolutionary model: incremental using feedback of provious iteration.

6.2 Unified Process

Iterative and incremental development, Use-Case-Driven
Belangerijkste fasen:

1. Inception
    - Approximate vision
    - Business case
    - Scope
    - Vague estimates
    - Continue or stop?
2. Elaboration
    - Identification of most requirements
    - Iterative implementation of the core architecture
    - Resolution of high risks
3. Construction
    - Iterative implementation of lower risk elements
    - Preparation for deployment
4. Transition
    - Beta tests
    - Deployment

6.3 Analyse

Types of requirements:

- Functional: features, capabilities ⇒ Use Cases
- Usability: human factors, help, documentation
- Reliability: frequency of failure, recoverability
- Performance: response times, throughput, accuracy, resource usage
- Supportability: adaptability, maintainability, configurability
- +: implementation, interfacen operations, packaging, legal

6.3.1 Use cases

geschreven verhalen over het gebruik van het systeem om een bepaald doel te verwezelijken.
Black box aanpak: geen interne werking systeem
Werkwijze:

1. Kies de systeemgrens
2. Identificeer de primaire actors
3. Identificeer hun goals
4. Bepaal de use cases overeenkomstig die goals

Zegt wat er gebeurt, niet hoe.

6.3.2 Domain model

beschrijft betekenisvolle concepten in het probleemdomein, hun associaties en attributen.

6.3.3 System sequence diagrams & operation contracts

beschrijft wat het systeem doet, UML versie van UC

6.4 Interaction and class diagrams

6.4.1 Interactiediagramma's

voor elke niet-triviale functionalitiet, met als doel de identificatie/ontwerp van een groep samenwerkende objecten om die functionaliteit te verwezelijken.
2 Soorten:

- Communication Diagrams: nadruk op objectnavigatie
- Sequence Diagrams: nadruk op tijdsverloop (favoriet van de prof)

6.4.2 Ontwerp-klassendiagrammas

shows methods and visibility, associations and attributes, generalization, inheritance and dependencies, ...

6.5 GRASP-Principes

Responsibility driven:

- Doing:
  - DIY
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects
- Knowing:
  - about private encapsulated data
  - about related objects
  - about things it can derive or calculate

Extra principle: Don't talk to strangers:
Do not couple two objects who have no obvious need to communicate.

6.6 Testing

- Unit testing: test individual components
- Module testing: test a collection of related components
- Sub-System testing: test sub-system interface mismatches
- System testing: test interactions between sub-systems and that complete system fulfils requirements
- Acceptance testing: test system with real rather than simulated data

Good Test-Coverage: minimum 70%, 95%+ without UI
Good Unit test:

- repeatable and deterministic
- requires no human intervention ⇒ executed in batch
- "self-describing": serves as documentation
- changes less often than the system: encode stable functionality

6.7 Design patterns en refactoring: zie boek

7 Hoog-niveau-informatie inleiding-software-ontwerp

- Vereistenanalyse: voor welke omgeving moet het systeem gebouwt worden, en welk gedrag moet het vertonen, opdat zijn waarde optimaal zou zijn? Resulteert in een **vereistenmodel**.
- Domeinmodellering: een analyse en een model worden gemaakt van de meest relevante concepten in het probleemdomein, en hun verbanden, teneinde een goed begrip op te bouwen en een precieze terminologie op te bouwen.
- Verdeel en heers: Ontbindt via de modellen het systeem in deelsystemen en verdeel de verantwoordelijkheden. Kan je recursief toepassen tot rechtstreeks programmacode geschreven kan worden.
- Software-ontwerp: het oordeelkundig ontbinden van een systeem en toekennen van verantwoordelijkheden aan de deelsystemen.
- Kwaliteitscontrole: op alle niveaus:

- de implementatie van het ontwerp: komen de realisaties overeen met de verantwoordelijkheden in het ontwerp?
- de implementatie en het ontwerp tegen het vereistenmodel: zijn de vereisten voldaan?
- validatie van het systeem en het vereistenmodel: is de klant blij?

Verificatie via:

- Testing: unit tests (deelsystemen) en integration testing (geheel systeem)
- Code review

7.1 Iteratieve en incrementele ontwikkeling

1. Vereistenanalyse
2. Ontwerp
3. Implementatie
4. ...

Bij een bug of een tekort kan het zijn dat het hele proces opnieuw uitgevoerd moet worden: Iteratief.

Hoe sneller een bug gevonden wordt hoe goedkoper.

Incrementele ontwikkeling: om verloren werk te verminderen enkel zoveel designen als nodig voor de volgende iteratie en dan implementeren en testen.

7.2 Kwaliteit van een software-ontwerp

Veel manieren om doel te bereiken en veel versies die enkel in structuur verschillen.

Goed ontwerpbeslissingen besparen kost, een eenduidig ontwerp schept duidelijkheid voor de teams die het implementeren.

- cohesie: de eenvoud van de verantwoordelijkheden van het deelsysteem. Hoge cohesie = simpeler te realiseren en wijzigen.
- koppeling: de complexiteit van de afhankelijkheden tussen de deelsystemen. Hoge koppeling = minder effectiviteit: moeilijker te realiseren en wijzigen.

7.3 Rest

Hou het simpel.

Use cases, use case model, domeinmodel, UML diagram.

Gebruik zo veel mogelijk de GRASP principes en ontwerppatronen, daarna refactoren we.