KU LEUVEN kulak

Informatica
software-ontwerp
Tom Holvoet & Bart Jacobs

**Samenvatting Software-Ontwerp**

Academiejaar 2018-2019

Thomas Bamelis

# Inhoudsopgave

# Introduction

Dit is een zeer korte samenvatting van wat ik lees in de boeken van wat we zagen in de lessen.
Ik weet nog altijd niet of het engels of nederlands wordt.
First of two definitions:

- UML = unified modeling language: The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.
- UP = Unified Process: an iterative software development process which describes an approach to building, deploying, and possibly maintaining software. It searches for the middle ground between the waterfall and agile models, adding waterfall principles to agile to make it usable with larger teams. Methods such as Extreme Programming, SCRUM,... are part of UP. The Rational Unified Process = RUP is the most popular implementation. The purpose of modeling is primarily to understand, not to document. UP has 4 main phases, but no, this is not like a waterfall model:

  1. Inception: approximate vision, business case, scope, vague estimates.
  2. Elaboration: refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
  3. Construction: iterative implementation of the remaining lower risk and easier elements, and preparation of deployment.
  4. Transition: beta tests, deployment.

  It is very important to understand these overlap and are iterated and are not like the waterfall model.

  This course focuses on the Elaboration and Construction phase. The other two are handled in different courses in the master.

- UML artifact = the diagrams created with UML, such as class diagrams, sequence diagrams,...

**TODO: the reasons with graphs for software engineering.** Those graphs showing how longer it takes you to discover a problem, the more expensive it is to correct it.

# 1 Requirements/analysis: Use cases and domain modeling

This course does not cover requirements engineering and only covers the functional part of the FURPS+ model.
In this section, three essential models will be explained:

1. **Use cases**: understand/model expected functionality.
2. **Domain modeling**: understand/model the problem domain.
3. **System behaviour (System Sequence Diagram)**: understand/model interaction between actors and system(s).

**System under discussion (SuD)** The system being talked about.

You need to make all three of them when developing software. The last one will be done twice in an iteration but with a different purpose. The second time will be discussed in the next section, where it will be used as a way of portraying interactions. The system sequence diagram can thus be used for more than one purpose.

## 1.1 Use Cases

Use cases are text stories, widely used to discover and record requirements. They are stories of some actor using a system to meet goals. Use cases are not diagrams, they are text.

**Actor** Something with behavior, such as a person (identified by a role, such as cashier), computer system or organisation. They are always capitalised in a use case for clarity.

**Scenario** A specific sequence of actions and interactions between actors and the system. It is one particular story of using a system, or one path through the use case.

**Use case instance** This is the same as a scenario.

**Use case** A collection of relates success and failure scenarios that describe an actor using a system to support a goal. An alternative definition by RUP is: A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor. Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.

**Context Diagram** An optional part to add to the use case, which is a UML use case diagram to show the names of use cases and actors, and their relationships.

Use cases have nothing to do with object oriented analysis. On page 68-72 there is an example of a use case.

Use cases are very useful to let the customer help to describe the system. This is very valuable because as a computer scientist, you will often not (fully) understand the domain you will be working with.

Use case are primarily functional requirements. A use case defines a contract of how a system will behave.

There are 3 types of actors:

**Primary Actor** Has user goals fulfilled through using services of the system.

**Supporting Actor** Provides a service to the system. Often a computer system (for example the cash register), but could be an organization or person.

**Offstage Actor** Has an interest in the behavior of the use case, but is not primary or supporting. For example: a government tax agency.

Use cases can be written in different formats and levels of formality:

**Brief** Short/brief one-paragraph summary, usually of the main success scenario.

**Casual** Informal paragraph format. Multiple paragraphs that cover various scenarios.

**Fully dressed** All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

See slides 13 and 14. I will next describe the different elements of the fully dressed formats except for the system name, which is trivial.

**Scope** The scope bounds the system under design. Typically, a use case describes use of one software (or hardware plus software) system. In this case it is known as a **system use case**. At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level process description is called a **business use case**.

**Level** How deep in the overall system this use case resides.

A **user-goal level** use case ( so a use case at user-goal level) describes the scenarios to fulfil the goals of a primary actor. This is the most used/common level.

A **subfunction-level** use case describes substeps required to support a user goal. It is usually created to factor out duplicate substeps shared by several regular(= user-goal level) use cases, to avoid duplicating common text.

**Primary Actor** The principal actor that calls upon system services to fulfill a goal.

**Stakeholders and Interest List** This is a very important element. This list contains all things that satisfy all the stakeholders' interests. So this holds all stakeholders along with all (and only) the behaviours which satisfies the goal(s) of the stakeholders. This stakeholder interest viewpoint provides a thorough and methodical procedure for discovering an recording all the required behaviors.

**Preconditions** First of all, don't bother with a precondition unless you are stating something non-obvious and noteworthy, to help the user gain insight. Don't add useless noise to requirements documents. Preconditions state what must always be true before a *scenario* is begun in the use case. Preconditions are not tested within the use case, but they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case, such as logging in. Preconditions communicate noteworthy assumptions that the writer thinks readers should be alerted to.

**Success guarantees = postconditions** First of all, don't bother with a postcondition/ success guarantee unless you are stating something non-obvious and noteworthy, to help the user gain insight. Don't add useless noise to requirements documents. Success guarantees state what must be true on successful completion of the use case (can be both the main success scenario or some alternate path). The guarantee should meet the needs of all stakeholders.

**Main Success Scenario** This is also known as **Basic Flow**. It describes a typical success path that satisfies the interests of the stakeholders. It (often) does not include any conditions or branching. Defer all conditional and branching statements to the Extension section. The main success scenario records steps. There are 3 kind of steps:

- An interaction between actors.
- A validation. (usually by the system)
- A state change by the system. (for example, recording or modifying something)

Step one of a use case does not always fall into this classification, but indicates the trigger event that starts the scenario.

**Extensions = Alternate Flows** Extensions indicate all other scenarios of branches, both success and failure. These compromise the majority of the text and are complexer than the main success scenario. The extensions and the main success scenario should together "nearly" satisfy all the interests of the stakeholders. Not all because some interests may be best captures as non-functional requirements. Extension scenarios are branches from the main success scenario and are therefore notated with the number of the step from which it branches along with a letter for identification. If an extension can can occur during any step, you should use a * instead of a number. If they can occur during multiple (but not all) steps, you can use a range (e.g. 3-6) instead of a number. An extension has two parts **the condition** and **the handling**. Write the condition as something that can be detected by the system or an actor. The handling consist of one or more steps. At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system). When a step in an extension is very complex, you can express the extension as a separate use case. Failures within extensions are notated as an extension of an extension (nesting?). An extension can sometimes branch of into an new use case (as stated just 2 sentences ago?).

**Special Requirements** If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case under special requirements. This is later often transferred to the Supplementary Specification because these requirements usually have to be considered as a whole during architectural analysis.

Here are some guidelines for use cases:

1. Write use cases in an essential style, keep the user interface out and focus on actor intent. (not enter password, but identify myself. Find the root goal)
2. Write short use cases. Including compact, not too frivolous sentences.

3. Write **Black-box use cases**: they do not describe the internal workings of the system, its components, or design. The system is described as having responsibilities. Software elements have responsibilities and collaborate with other elements that have responsibilities. Specify "what", not "how".

4. Write requirements focusing on the users or actors of a system, asking about their goals and typical situations. Focus on understanding what the actor considers a valuable result. An incredible amount of software projects failed because they did not implement what people really needed (ict autists).

5. See below on how to find use cases.

6. Here are 3 tests to find *useful* use cases at the correct level:

   - The Boss test: ask the user:"Your boss asks: 'what have you been doing all day?'. The user replies:'*the task/goal completed by the use case (e.g. logging in)*'. Would the boss of the user be happy with the answer?" If the answer is "no", the use case is not useful and too specific. This test is not waterproof and may fail important things, such as authentication. But you get the idea.
   - The EBP test: an **Elementary Business Process (EBP)** is a task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state. Focus on use cases that reflect EBPs. Not too small not too big
   - The size test: a use case typically contains many steps, and in the fully dressed format will often require 3-10 pages of text.

7. Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text. If an organization is spending many hours (or worse, days) working on a use case diagram and discussing use case relationships, rather than focusing on writing text, effort has been misplaced. Downplay diagramming, keep it short an simple.

8. Draw a simple use case diagram in conjunction with an actor-goal list. For a use case context diagram, limit the use cases to user-goal level use cases. Show computer system actors with an alternate notation to human actors. Put the primary actor on the left, and supporting actors on the right.

9. See p95 for some comments on use cases in iterative development.

How to find use cases (werkwijze):

1. Choose the System Boundary: If the definition of the boundary of the system under design is not clear, it can be clarified by further definition of what is outside the boundary. The external primary and supporting actors. Once the external actors are identified, the boundary becomes clearer. (e.g. everything outside the cash register is out of bounds, including the cashier and customer)

2. Identify the primary actors: You must identify the actors before their goals. Do this by brainstorming. Page 83-84 contain questions to identify actors. Or identify them by searching for processes/goals they take part in (=actor-based). Or identify external events the system should react to and tie these events to their actors and use cases (event based).

Here are some tips from the slides:

- Plaats jezelf in de rol van de actor. Wat wil die bereiken.
- Taalgebruik nooit passief, maar actief. Zeg wie wat doet.
- Geen GUI details (eerste guideline).
- Zeg wat de actor doet en niet wat hij denkt.
- Zeg wat iedereen doet en niet hoe.

- Maak geen use cases van individuele onderdelen en operaties.
- It is a relatively large end-to-end process description.

Zie slides 31-33 voor belangrijke informatie over rangschikken use cases, conclusies en non-functional requirements.

## 1.2 Domain Models

A domain model describes noteworthy/meaningful concepts in a domain. A critical quality to appreciate about a conceptual model is that it is a representation of real-world things, not of software components. Concepts in this model are not software objects, so no databases, windows, functions,....

**Domain model** a visual representation of conceptual classes or real-situation objects in a domain. Synonyms are: **conceptual, domain object or analysis object model**. It shows (see definitions below):

1. Conceptual classes = domain objects.
2. Attributes of conceptual classes.
3. Associations between conceptual classes.

This is not a data model. There is a completely different meaning for domain model in software engineering for something else, "the domain layer of software objects". In this book they call that the "domain layer" to avoid confusion.

**Conceptual Classes** An idea, thing or object (informal). It is defined in terms of the following:

1. **Symbol**: words or images representing a conceptual class.
2. **Intension**: the definition of a conceptual class.
3. **Extension**: the set of examples to which the conceptual class applies.

Do not exclude a class simply because the requirements don't indicate any obvious need to remember information about it or because the conceptual class has no attributes. You draw a box for every conceptual class and put the name in the top of the box and draw a line under it.

**Attribute** A logical data value of conceptual class that are needed to satisfy the information requirements of the current scenarios under development. They are placed in the second compartment of the conceptual class box. Keep attributes simple, preferably as primitive datatypes and no complex datastructures. Include attributes that the requirements (e.g. use cases) suggest or imply a need to remember information. The full syntax for an attribute in UML is "**visibility name : type multiplicity = default property string**". Private visibility (-) is the default. Multiplicity can be used to indicate the optional presence of a value, or the number of objects that can fill an attribute (e.g. middle name). When we want to communicate that an attribute is noteworthy but derivable, the name is preceded by "/". Most numeric quantities should not be represented as plain numbers, because they have a unit. In the general case, the solution is to represent "Quantity" as a distinct class, with an associated Unit. It is also common to show Quantity specializations.

**Associations** An association is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection. See the common associations list on page 155-156. An association is *not* a statement about data flows. They have a meaning in the real world and will not all be implemented in software, although many will. It is represented as a line between *classes* with a *capitalized* association name. The name should be a verb-phrase and about one moment in time. It can have a reading arrow, but this says nothing about the model and is not a statement about connections between software entities. Each end of an association line is called a role and expresses multiplicity. The multiplicity value communicates how many instances can be validly associated with another, at a articular moment, rather than over a span of time. The multiplicity *furthest* from the class says how many instances of the *other* class can be connected to *this* class. Two classes can have more than one association, this is not uncommon.

**Description classes** A conceptual class which describes something else (another class?). For example a class ProductDescription which describes the price, picture, info about a class item. Add a description class when:

1. There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
2. Deleting instances of things they describe results in a loss of information that needs to be maintained, but was incorrectly associated with the deleted thing.
3. It reduces redundant or duplicated information.

This can be handy for example in a supermarket, if every item-object of the item class represents an actual object in the stores. It is better to have an object which holds the info about an apple than store the info in all the apple objects.

There are three methods to identify Conceptual classes:

1. Reuse or Modify Existing Models. This is the first, best and usually easiest approach. There are published, well-crafted domain models and data models (which can be modified into domain models) for many common domains (e.g. inventory, finance, health,...). Some great books containing great models are "Analysis patterns" by Martin Fowler, "Data Model Patters" by David Hay, and the "Data Model Resource Book" by Len Silverston. This is I think exactly what the second book of this course is about (Design Patters).
2. Use a Category List. Make a list of candidate classes. There are priorities in the list. See page 140 and 141 for common categories worth considering.
3. Noun Phrase Identification (= linguistic analysis). Identify the nouns and noun phrases in textual descriptions (e.g. use cases, minds of experts, other documents...) of a domain, and consider them as candidate conceptual classes or attributes. Care must be applied with this method. A mechanical noun-to-class mapping isn't possible and words in the natural languages are ambiguous. You can combine it with the category list technique.

Consider including the following associations in a domain model:

1. Associations for which knowledge of the relationship needs to be preserved for some duration = **Need-to-Remember associations**.
2. Associations derived from the Common Associations List on page 155-156.

It is mainly preferred for attributes to be a primitive datatype, but you can make "invent" a new one if one of the following conditions is met:

- It is composed of separate sections. (e.g. phone number, name, adres,...)
- There are operations associated with it, such as parsing or validation. (e.g. social security number)
- It has other attributes. (e.g. start and end of a promotion)
- It is a quantity with a unit. (e.g. different currencies)
- It is an abstraction of one or more types with some of these qualities. (e.g. an item identifier which generalises UPC and EAN product numbers)

You can chose whether or not you give the new datatype a class or if it just remains an attribute. The former could be handy if it has many sub-attributes. The choice is yours. See page 164-165.

Here are the guidelines provided by the book:

- Avoid a waterfall-mindset big-modeling effort to make a thorough or "correct" domain model. It won't never be either, and such over-modeling efforts lead to *analysis paralysis*, with little or no return on the investment. Limit domain modeling to no more than a few hours per iteration.

- leave the right and bottom lines of a class diagram open when drawing by hand to allow it to grow.
- You should not always want to update the domain model diagram. It is just a prototyping concept tool, quick and rough. Often the evolving domain layer of the software hints at most of the noteworthy items.
- Think like a cartographer or mapmaker: use the existing names in the territory, exclude irrelevant or out-of-scope features and do not add things that are not there.
- When modelling something very abstract listen closely to the concepts and vocabulary used by the domain experts.
- If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.
- Avoid adding many associations because it will obscure the diagram with "visual noise". The purpose of the domain model is to make things clear, so it would defeat the purpose. They can rise exponentially in function of the classes (n*(n-1))/2.
- Relate conceptual classes with an association, not with an attribute (so no SQL sheit).
- Attributes should not be used to relate conceptual classes in the domain model. A frequent mistake of this sort are foreign keys.

**Design creep**: solidifying implementation decision in design. As Trump would say: "BAD!".

## 1.3 System Sequence Diagrams

A system sequence diagram (SSD) is a picture that shows, *for one particular scenario of a use case*, the events that external actors generate, their order, and inter-system events. It is thus derived from a use case. All systems are treated as a black box. The emphasis of the diagram is events that cross the system boundary from actors to systems. An SSD show what is done, not how (black box). A scenario implies an SSD. An SSD zooms in on the interaction between users and the system. This is useful to better understand/identify external input events (= **system events**). You should draw an SSD for a main success scenario of each use case, and frequent or complex alternative scenarios. See page 175 for a visual example. In UML an SSD is known as a "sequence diagram". You should keep the names of the events as general and intent-focused as possible (e.g. enterItem is better than scan). You should keep the elements of an SSD basic. The details can be written down in the glossary (see below). Don't create SSDs for all scenarios, unless you are using an estimation technique (such as function point counting) that requires identification of all system operations. Draw them only for the scenarios chosen for the next iteration. You should not spend more than a half hour on an SSD, an preferably a couple of minutes. SSDs are mostly created in the elaboration phase.

In the slides, the following is said which I don't know what it means:
Kenmerken:

- Iteratie: *[ conditie ] + omkadering van systeem events
- Terugkeerwaarde: abstractie die voorstelling en medium negeert
- Parameters: eveneens abstracte voorstelling

## 1.4 Glossary

In its simplest form, the glossary defines noteworthy terms (just a dictionary). It also encompasses the concept of the **data dictionary**. The data dictionary records requirements related to data, such as validation rules, acceptable values,... The glossary can detail any element: an attribute of an object, a parameter of an operation call, a report layout,... Other important terms/artifacts for Requirements are found on p58, but they are either already mentioned or they have not come up in the lessons.

# 2 Design: interactions and class diagrams

Design is about realising software.

## 2.1 Interaction Diagrams

Interaction diagrams illustrate how objects interact *via messages*. These objects resemble classes and actors, not the full system and actors. They are used for **dynamic object modeling**. Dynamic modeling refers to represent the object interactions during runtime. These can be expressed using: SEQUENCE,ACTIVITY, COLLABORATION diagrams and thus also Interaction diagrams. The books says the chapter about interaction diagrams is a reference to skim through. So to quickly look up info about it, see chapter 15 on page 221. However, I do think the slides serve better as quick reference. There are 2 types of interaction diagrams, communication diagrams and sequence diagrams. You can choose which one you use, but in practice most people use sequence diagrams.
It is very important to look at the slides in this section,
because I will not type all the stuff about it and do the visuals,
but I will say things that are not in the slides.

### 2.1.1 Commonalities of interaction diagrams

The participating objects are notated in a box. This box is called a **lifeline box**. Terminology for the explanation is used as if we already know about classes, because they often will be.

- **Class** is the class Class.
- **:Class** is a random instance of the class Class. As of UML 2, this can also be an interface or an abstract class (of course also a superclass, which is a class itself).
- **name:Class** is an instance of the class Class with name as name.
- **«metaclass» instance** an instance instance of the metaclass metaclass (is something you do not know, a class which instances are classes)
- **name:ArrayList<Class>** an array named name which holds instances of the class Class.
- **name[i]:Class** an instance from the class Class which is the i'th element of the array named name, which is an array composed of instances of the class Class.
- **singleton**: if the class is a singleton (ever only one instance of the class), it is noted by a small 1 in the upper right corner.

A message is noted as follows: **return = message(parameter: parameterType) : returnType**
An asynchronous message is noted with a thin curly arrow. The name of the message which creates an object is **create**. The name of the message which explicitly destroys an object is **«destroy»**. You can use a (abstract) superclass in a lifeline box and then draw a separate diagram for each case for the subclasses if necessary.

Guidelines:

1. Spend enough time doing dynamic object modeling with interaction diagrams, not just static object modeling with class diagrams. Too many developers spend too little time on interaction diagrams while they are incredibly valuable.

### 2.1.2 Communication Diagram

Communication diagrams illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram. See page 240 and on. Between 2 lifeline boxes exchanging messages there is **exactly 1** line. This line is NOT directed. All messages between the boxes flow through the line. Messages are written as stated above, BUT they are numbered. All messages, *except the starting message*, have a number before them and then a colomn ":". Messages send at the same time have incremental numbers. A message which is the consequence of another message, an thus is send strictly and instantly

after that message, is notated by the number of that message followed by a dot "." and a new number. All messages (even all the ones on the same line) are followed by an arrow indicating in which direction the arrow flows.
If statements are notated as [*condition*] with this put between the number(s) and the column ":". You can make an else with condition

*not condition*

for a message leaving from the same lifeline box as the one with the if statement. A for statement is notated by *[i=1...n] and if you want to iterate over something, you make the $i$ the index of the element of an array in the lifeline box which is pointed to.

### 2.1.3 Sequence Diagram

Sequence diagrams illustrate object interactions in a kind of fence format, in which each new object is added to the right. Each lifeline box has a dotted line below, the **lifeline**,with boxes where the object is active, the **execution specification bars** (also called activation bar or activation). This is called the **lifeline**. A message is a filled-arrowed solid line. The starting message is called the **found message** and has a small solid ball at the start of the arrow. The return of a message can be notated as specified above, but also as a dotted line with an arrow and as name the variable name of the return value. Messages to itself are messages where the arrows turn back to the lifeline of the object. A create message has a dotted line. When an object is destroyed, the lifeline stops and an **X** is placed at the end.
Blockstatemens can be made by drawing a rectangle around them, and putting one of the next words (and conditions if necessary) in the upper-left corner:

- if-else = Alt
- if = opt
- for = loop (a little box stating i++ or something is placed on the lifeline)
- execute in parallel = par
- critical region where only one thread can run = region
- entire rectangle around a sequence diagram to be able to use it as function in another = sd name (name is the name of the diagram)
- use another diagram as function = ref name (name is the name of the diagram you want to call)

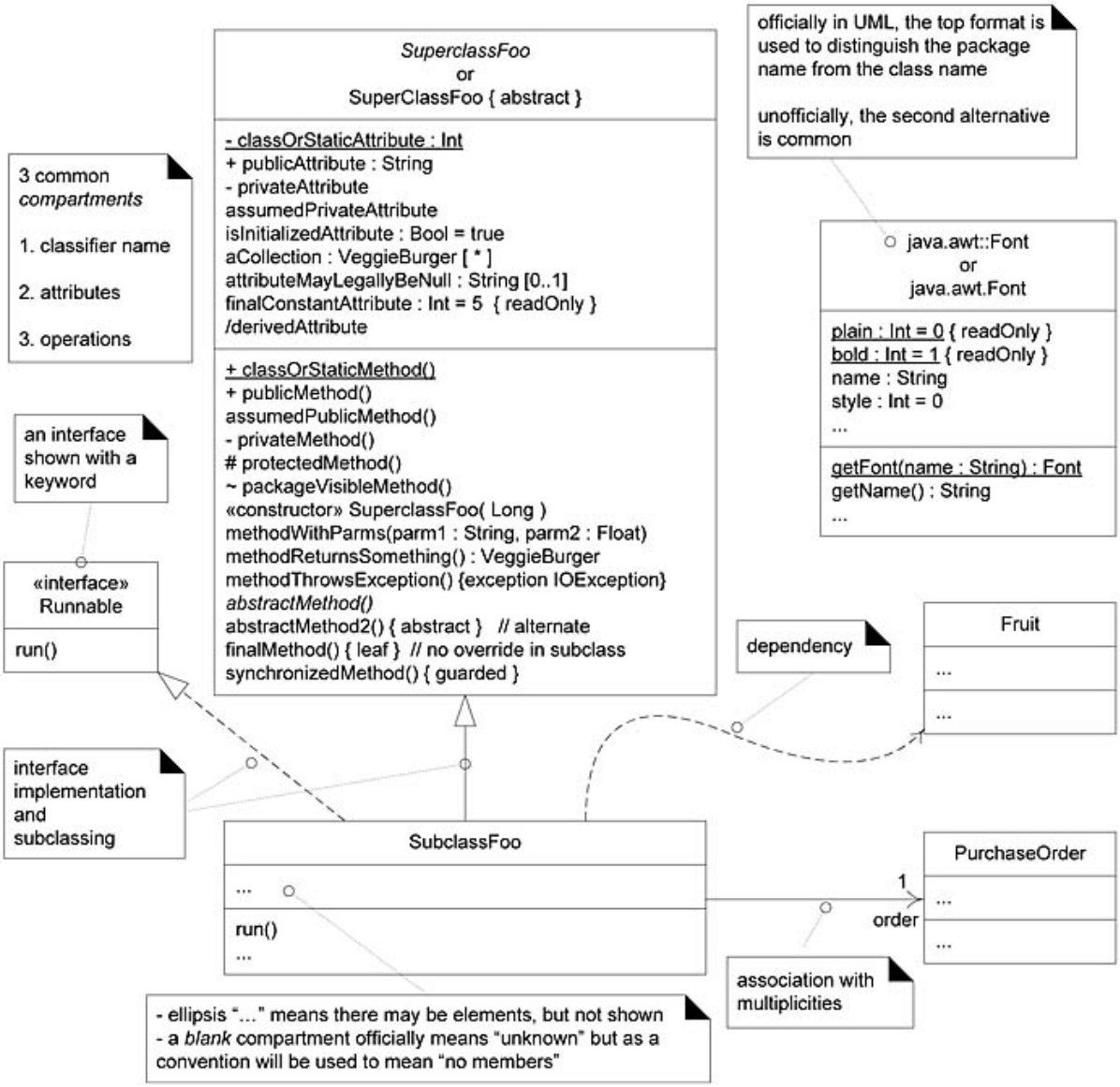
There was only one extra guideline here, and it is that for if = opt squares, for 1 arrow statements you can precede the name of the message with the condition between square brackets. This is not supported anymore in UML 2, so you should not use it, but some people still do, that is why you need to know this.

### 2.1.4 Communication VS. Sequence

| Type | Strengths | Weaknesses |
|---|---|---|
| Communication | • Compact<br>• Flexibility to add new objects | • More difficult to see sequence of messages<br>• Less notation options in UML |
| Sequence | • Clearly shows sequence or time ordering of messages<br>• Large set of detailed notations in UML because it is more widely used | • Forced to extend to the right when adding new objects<br>• Consumes horizontal space. Reading and regular paper is vertical, so you cannot add objects indefinitely without braking the diagram. |

## 2.2 Class Diagrams

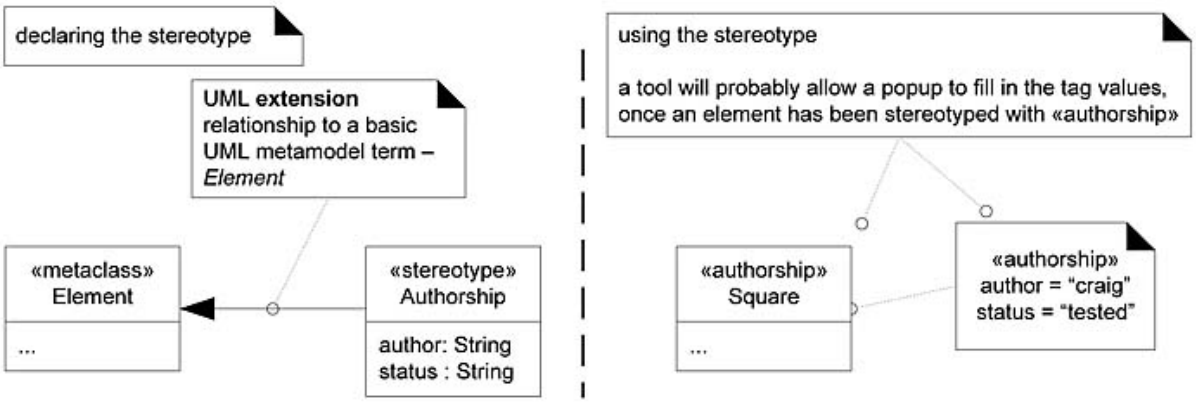
Class diagrams illustrate classes, interfaces and associations. They are used for **static modeling**, which is used to specify the structure of the objects that exist in the problem statement. These can be expressed using: CLASS, OBJECT and USECASE diagrams. You identify software classes by analysing the interaction diagrams and the conceptual model (=domain model). See slide 26, you literally take a subset of the classes from the domain model there. In contrast to the domain model, the class diagram has methods and specifies the visibility of its elements. The common modeling term for designing class diagrams is a **design class diagram (DCD)**. This is because it looks similar to a domain model diagram, but there is a huge difference between the two. Namely, domain model is a conceptual design, while DCD is a software or design perspective. The chapter of the book should also be used as a reference (starting p249). He says there is no need to memorize all these low level details, so I will just include the pages for a lot of stuff cause I am wasting way to much time on this shit.




Figuur 1: The summary provided by the book.

Here are some definitions:

- A **keyword** is a textual adornment (=versiering) to categorize a model element. Keywords are notated between «» such as «interface» or between {} such as abstract.
- A **stereotype** represents a refinement of an existing modeling concept and is defined within a UML profile (see below). It is also notated between «» such as «destroy». It defines a set of tags, so when an element is marked with a stereotype, all the tags apply to the element, and can be assigned values. See figure 2.
- A **tag** is an attribute of a stereotype.
- A UML **profile** is a collection of related stereotypes, tags and constraints to specialize the use of the UML for a specific doamin or platform, such as a UML profile for project management or data modeling.




Figuur 2: Example of a stereotype

### 2.2.1 Classifier (classes, etc)

**Classifier** A model element that describes behavioral and structure features. Classifiers can also be specialized. They are a generalization of many elements of the UML, including classes, interfaces, use cases, and actors. In class diagrams, the two most common classifiers are regular classes and interfaces.

A classifier always has 3 compartments:

1. Classifier name
2. Attributes
3. Operations (=methods)

You can define your own compartments such as an "exceptions thrown" or "responsibility" compartment.

### 2.2.2 Associations

You should only notate associations by an association line, not by including an attribute or doing both (slide 29). Except for data type objects such as string, self created data-types such as address, and so on ... See page 245 for more examples. Such a line has one or two **navigability arrow(s)** pointing from the class who will hold a reference to the class it will reference. It uses the same multiplicity notation as the domain model. An association line has one or two **rolename(s)** under the navigability arrow(s) to show what the attribute name will be of the reference when implemented. Under this name, you can put properties of the attribute between {}. Properties can be things such as list, ordered,unique... An association does not have a central association name, unlike in the domain model. A list of objects is then denoted by a 0...* or 1...* association with property list, and no entry of it in the attributes just like said above.

### 2.2.3 Attributes

The notation for attributes is:
**visibility name : type multiplicity = default {property string}** See in methods what the property string is. Attributes are usually assumed private if non visibility is given.

### 2.2.4 Operations and Methods

A method is the implementation of an operation. That is what the book says, but we know them as the same. Operation is the line for the method in the class diagram. A method is notated as:
**visibility name (parameter-list) : return-type property-string**
with property string being some arbitrary additional information, such as an exceptions that may be raised, if the operation is abstract, or even self-created properties with an assigned value such as *property=value,...* A method/operation is assumed public if no visibility is shown. The constructor method is always notated by the *create* method, independent of what the name is in the code because different languages have different constructor name rules. You can also precede create by «constructor» for clarity. Getters and setters are often excluded (or filtered) from the class diagram because of the high noise-to-value ratio they generate (they don not add much value, but make the diagram much more cluttered).

### 2.2.5 Notes

Every element of the class diagram can have a note, making a comment about it. It will be put close to the element with a dashed line with a dot at the end pointing to the element. A note can also show how a method is implemented, when the first line is «method». When you include an implementation, you are mixing static and dynamic views.

### 2.2.6 Constraint

A note but enclosed by braces {}, which represent a constraint such as an @invar, postcondition and precondition etc... UML has its own language to specify constraints, the Object Constraint language (OCL). See slide 39 or p 265.

### 2.2.7 Generalization

Generalization is the same as inheritance for the class diagram, but a subset relation in a domain model. It is notated with a solid line and fat triangular arrow from the subclass to superclass. The official definition is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier indirectly has features of the more general classifier. Abstract classes are written in italic or followed by {abstract} and final classes and methods are shown with a {leaf} tag.

### 2.2.8 Dependency

A general **dependency relationship** that indicates a **client** element ( of any kind, including classes, packages, parameter, use cases, interfaces, subclasses, associations, object creation and use in a method,... ) has knowledge of another **supplier** element and that a change in the supplier could affect the client. This is a broad relationship. The client and the supplier (=server) are said to be **coupled**. Dependency is any line in the diagram. When in the code of an element (e.g. class) there is anywhere a mention of another elements such as class/interface/... there is a dependency between them. BUT, there can also be dependencies when this is not the case.

In class diagrams, use the dependency line (dashed with thin arrow) to depict global, parameter variable, local variable and static-method (when a call is made to a static method of another class) dependency between objects. The dependency line can be labeled with keywords or stereotypes (e.g. «call», «create»,... ).

### 2.2.9 Interfaces

When a class uses another class through an interface, this is notated by a **socket** line. See page 263. It shows several important notations for interfaces, but I could not find a picture of it and it can't be scanned.

### 2.2.10 Aggregation and Composition

Aggregation is a regular association. Don't use aggregation, but the following when necessary. **Composition** is a strong kind of aggregation which suggests an instance of the part belong to only one composite instance at a time, and that the part must always belong to a composite and the composite is responsible for the creation and deletion of its parts (either by creating or deleting the parts, or by collaborating with other objects. For example, a hand can be the composite and a finger the part. This is notated with a solid line with a thick part by the composite. A composite relationship does not need a name. See page 264.

### 2.2.11 Qualified association

This has a qualifier that is used to select an object from a larger set of related objects, based upon the qualifier key. Something like a hashmap. This has an impact on the multiplicity. See page 265-266.

### 2.2.12 Association Class

An association class allows you to treat an association itself as a class, and model it with attributes, operation,... This can be used to simplify many-to-many multiplicity.

### 2.2.13 Singleton Classes

This is a design pattern and will be talked about later, but it means only 1 instance of the class can exist at any time.

### 2.2.14 Template Classes and Interfaces

See page 267-268.

### 2.2.15 Active Class

An active object runs on and controls its own thread of execution. It is notated with double vertical lines on the left and right sides of the class box.

## 2.3 Relationship Between Interaction and Class Diagrams

In practice, you should make interaction and class diagrams in parallel. They need to be consistent with each other. All the classes/messages in the interaction diagram need to be in the class diagram as classes/operations. A good UML tool should automatically support changes in one diagram being reflected in the other. If wall sketching, use one wall for interaction diagrams, and an adjacent wall for class diagrams.

# 3 Design: GRASP Patterns

The first 5 are covered in chapter 17 (p271) and the other 5 in chapter 25 (p413).

## 3.1 Responsibility-driven design

**Responsibility-driven design (RDD)** In RDD we think of software objects as having responsibilities, with responsibilities being an abstraction of what they do. A responsibility is a contract or obligation of a classifier. Responsibilities are related to the obligations or behavior of an object in terms of its role. RDD is a general metaphor for thinking about Object Oriented software design. Think of software objects as similar to people with responsibilities who collaborate with other people to get work done. RDD leads to viewing an OO design as a *community of collaborating responsible objects.* Responsibilities are assigned to classes during object design, and are reflected in methods. A responsibility is not the same thing as a method, it is an abstraction, but methods fulfill responsibilities.
RDD also includes the idea of **collaboration**. Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects. Responsibilities starts from the design phase, thus from the interaction and class diagrams.

There are 2 types of responsibilities:

1. **Doing**
   - doing something itself, such as creating an object or doing a calculation
   - initiating action in other objects
   - controlling and coordinating activities in other objects

   Example: a Sale is responsible for creating SalesLineItems.
2. **Knowing**
   - knowing about private encapsulated data
   - knowing about related objects
   - knowing about thing it can derive or calculate

   Example: a Sale is responsible for knowing its total.
   Knowing responsibilities are often derivable from the Domain Model. For software domain objects, the domain model, because of the attributes and associations it illustrates, often inspires the relevant responsibilities related to "knowing".

**Low Representational Gap (LRG)** is when the internal structure of the software closely reflects the structure of the domain that the software is designed to fit in.
The **granularity of a responsibility** is how many classes and methods it takes to fulfil the responsibility. Big responsibilities can require hundreds of classes and methods.

## 3.2 GRASP

**General Responsibility Assignment Software Patterns (GRASP)** names and describes some basic principles to assign responsibilities to support RDD. GRASP is a learning aid for OO design with responsibilities. This approach to understand and using design principles is based on *patterns of assigning responsibilities.* The whole of chapter 18 (p321) is dedicated to examples for the first 5 GRASP principles.

**Understanding how to apply GRASP for object design is a key goal of the book.**

Once you grasp the fundamentals, the specific GRASP terms aren't important. You begin to use GRASP when you draw the interaction and class diagram. A **pattern** is a *named* and *well-known* problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussions of its trade-offs, implementations, variations and so forth. The **Gang of Four (GoF)** are the 4 writers of the design pattern book for this course. The book is apparently insanely important and popular.

**It is a key goal to learn both GRASP and essential GoF patterns.**

See pages 281-290 for an example of the first 5 GRASP patterns/principles.

### 3.2.1 Creator

**Problem:** who should be responsible for creating a new instance of some class?

**Solution:** Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

1. B contains A (B is a **Container**)
2. B compositely aggregates A (B is a **Composite**)
3. B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.
4. B records A (B houd instanties van A bij) (B is a **Recorder**)
5. B closely uses A

B is a **creator** of A objects. If more than one option applies, usually prefer a class B which aggregates or contains class A. Example on page 292. When creation of an object is very complex, it is better to use a helper class which creates the object. These classes will be talked about later (p440) and are called *Concrete Factory* or an *Abstract Factory*.
**Benefits:** the basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event. Choosing it as the creator supports low coupling.

### 3.2.2 Information Expert (or Expert)

**Problem:** What is a general principle of object design and responsibility assignment?
A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. If we have chosen well, systems tend to be easier to understand, maintain, and extend, and our choices afford more opportunity to reuse components in future applications.
**Solution:** assign a responsibility to the information expert. This is the class that has the information necessary to fulfill the responsibility. Start assigning responsibilities by clearly stating the responsibility. Example on page 294. When trying to find the information expert, if there are relevant classes in the Design Model, look there first. Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes. The expert principle expresses the common intuition that objects do things related to the information they have. In the real world we commonly give responsibility to individuals who have the information necessary to fulfill the task. Notice that fulfillment of a responsibility often requires information that is spread across different classes of object. This implies that many partial information experts will collaborate in the task. The example on page 295-296 shows this very well with getTotal() and getSubTotal().
In some situations a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.
**Benefits**: information encapsulation is maintained since objects use their own information to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems. Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. This supports high cohesion.

### 3.2.3 Low Coupling

**UNBELIEVABLY IMPORTANT!!!**

**Problem:** how to support low dependency, low change impact, and increased reuse?

**Coupling** is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low/weak coupling is not dependent on too many other elements. "Too

many" is context dependent, but we examine it anyway. These elements include classes, subsystems, systems,... A class with high/strong coupling relies on many other classes. Such classes may be undesirable because they suffer from the following problems:

- Forced local changes because of changes in related classes.
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Example of coupling are:

- A has an attribute that refers to a B instance or B itself.
- An A object calls on services of a B object.
- A has a method that references an instance of B, or B itself, by any means. These typically include a parameter or local variable of type B or the object returned from a message being an instance of B.
- A is a direct or indirect subclass of B. Inheritance is a very strong form of coupling, see page 301 about halfway to page.
- B is an interface, and A implements that interface.

In general, if anywhere in the code the name of a classifier is written, there is always coupling between them. However, there can also be coupling without that happening, zie voorbeeld op laatste witte pagina van de slides over class diagrams. Ik begrijp die notas wel niet, maar misschien later wel.
*Classes need to be implemented from least-coupled (and ideally, fully unit tested) to most-coupled. See chapter 21 (p385) for test-driven development.*

**Solution:** assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives. Example on p299-300.
In practice, the level of coupling alone cannot be considered in isolation from other principles such as Expert and High Cohesion. Nevertheless, it is one factor to consider in improving a design. Low coupling is a principle to keep in mind during all design decisions. It is an underlying goal to continually consider. It is an **Evaluative Principle** that you apply while evaluating all design decisions. Low Coupling supports the design of classes that are more independent, which reduces the impact of change. It is important to note that it is not high coupling per se that is the problem in the moment, it are the problems it can provide later when changes have to be made. It is fundamental for future-proofing your application, but if there is no realistic motivation for future-proofing, for example if nobody will ever change the software again, you should not waste time on it.
**Benefits**

- Not affected by changes in other components.
- Simple to understand in isolation.
- Convenient to reuse.

### 3.2.4 Controller

**Problem:** what first object beyond the UI layer receives and coordinates ("controls") a system operation?

**System operations** are the major input events upon the system. Pressing a button for example. These have been explored during the analysis of SSD.

**Controller** A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

**Solution:** assign the responsibility to an object representing one of the following:

1. Represents the overall "system", a "root object", a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*).

2. Represents a use case scenario within which the system operation occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session (*use case* or *session controller*).

    - Use the same controller class for all system events in the same use case scenario.
    - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).

**Important:** Note that "window", "view", and "document" classes are not on this list. Such classes should *not* fulfill the tasks associated with system events. They typically receive these events and delegate them to a controller. The GRASP controller is essentially unaware of what UI technology is being used. UI objects and the UI layer should not have responsibility for handling system events (p312). It is NOT the controller in the term (which I don't know, but it's programmers jargon) Model-View-Controller (MVC). To understand this and how it interacts with windows/... you must have a look at the theoretical example on page 303-305 and surely at the practical (java code) example on page 309-312 which also contains commentary on bloated controllers.
*Guideline*: normally, a controller should *delegate* to other objects the work that needs to be done. The responsibility should be contained to control and coordination. It coordinates or controls the activity. It does not do much work itself.
Types of controllers:

**Facade controller** represents the overall system, device or a subsystem. The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application and that provides the main point of service calls from the UI layer down to other layers. The facade could be an abstraction of the overall physical unit, such as phone or robot. The facade could also be any other concept or subsystem, such as ChessGame. Facade controllers can only be used when there are a limited ("not too many") amount of system events (Cohesion!), or when the UI cannot redirect system event messages to alternative controllers, such as in a message-processing system.

**Use Case Controller** has a different controller for every use case. These controllers will be an artificial construct to support the system (Pure fabrication, something completely made up that does not exist in real life) and is not a domain object. You should use these when facade has to many system events which results in high coupling or low cohesion. It allows for control of the sequence of events and can hold information about the state of a use case (this sentence is found exclusively in the slides and they are vague).

An issue that often occurs is the following:

**Bloated Controllers** is a poorly designed controller class which is unfocused and handling too many areas of responsibility which results in low cohesion. Signs of bloating are:

- There is only a *single* controller class receiving *all* system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and it maintains significant information about the system or domain, which should have been distributed to other objects, or it duplicates information found elsewhere.

Bloated controllers can be cured with, among others, the following actions:

1. Add more controllers. A system does not have to need only one. Instead of facade controllers, employ use case controllers. For example, in an airline reservation system there can be a controller for making reservations, for managing flight schedules and for managing pricing.

2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

**Benefits:**

- Increased potential for reuse and pluggable interfaces. Because the controller is not bound to a particular UI implementation, it is able to be reused with other implementation, and thus "plug" another user interface. This means not just another visual UI, but could also be changed to something completely different like only buttons.
- Opportunity to reason about the state of the use case. When system operations have to occur in a legal sequence or the current state of activity and operations within the use case have to be known, you can capture this in the controller because the same controller always has to be used throughout the same use case.

### 3.2.5 High Cohesion

**UNBELIEVABLY IMPORTANT!!!**

**Problem:** how to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

**Cohesion** (functional cohesion) is a measure of how strongly related and focused responsibilities of an element are. An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems,... A component (class) has high functional cohesion when its elements all work together to provide some well-bounded behavior.

A class with low cohesion does many unrelated thing or does too much work, and suffers from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate, constantly affected by change

Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.
**Solution:** assign responsibility so that cohesion remains high. Use this to evaluate alternatives. See the example on p315. In practice, the level of cohesion alone cannot be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.
Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions. It is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.
Here are some rules of thumb to indicate the degree of cohesion of a class (from low/bad to high/good):

- Very low cohesion: a class is solely responsible for many things in very different functional areas.
- Low cohesion: a class has sole responsibility for a complex task in one functional area.
- Moderate cohesion: a class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.
- High cohesion: a class has moderate responsibilities in one functional area (e.g. interacting with a database) and collaborates with other classes to fulfill tasks.

**High cohesion** indications: a class has moderate responsibilities in one functional area (e.g. interacting with a database) and collaborates with other classes to fulfill tasks. It has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.
The real world analogy for high cohesion is a common observation that if a person takes on too many unrelated responsibilities (especially ones that should properly be delegated to others), then a person is not effective. This is observed by managers who have not learned how to delegate.
Another classic principle related to coupling and cohesion should at the same time be promoted:

**Modular design** Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. A modular design is promoted by creating methods and classes with high cohesion. Each method should be designed with a clear, single purpose and by grouping a related set of concerns into a class.

In a few cases, accepting lower cohesion is justified (don't do this bullshit, they literally contradict themselves, wtf):

- Grouping of responsibilities or code into one class or component to simplify maintenance by one person, although this does worsen maintenance. For example when a very good SQL but very bad OO programmer groups all SQL operations in one location.
- Because of overhead and performance implications associated with remote objects and remote communication with distributed server objects, it is sometimes associated with fewer and larger, less cohesive server objects that provide an interface for many operations. This is related to a design pattern called **Coarse-Grained Remote Interface**.

**Benefits:**

- Clarity ans ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

### 3.2.6 Polymorphism

**Problem:** who is responsible when behavior varies by type? How to create pluggable software components?

**Alternatives bases on type** If a program is designed using if-then-else statements, then if a new variation arises, it requires modification of the case logic (often in many places). This makes it expensive and times consuming to add new variations.

**Pluggable software components** How can you replace one component of the software with another, without affecting pieces of the software outside that component.

**Solution:** 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. What I think they mean by this, that it is better to make for all types of vehicles a new class and overload the method, instead of having one method that does something else for every type with an if-then-else statement. You assign the responsibilities to the subclasses. See example on page 414-420.
*Guideline:* unless there is a default behavior in the superclass declare a polymorphic operationi (=methode) in the superclass to be abstract.
*When to use interfaces:* the general answer is to introduce one when you want to support polymorphism without being committed to a particular class hierarchy. If an abstract superclass AC is used without an interface, any new polymorphic solution must be a subclass of AC, which is very limiting in single-inheritance languages such as Java and C#. As a rule of thumb, if there is a class hierarchy with an

abstract superclass C1, consider making an interface I1 that corresponds to the public method signatures of C1, and then declare C1 to implement the I1 interface. Then, even if there is no immediate motivation to avoid subclassing under C1 for a new polymorphic solution, there is a flexible evolution point for unknown future cases.
Considering all the above, you should be realistic about not doing too much interfaces and polymorphism for future-proofing, because it increases coupling. It is unnecessary to use polymorphism for variations points that in fact are improbable and will never actually arise.
**Benefits:**

- Extensions required for new variations are easy to add.
- New implementations can be introduced without affecting clients.

### 3.2.7 Pure Fabrication

**Problem:** who is responsible when you are desperate, and do not want to violate high cohesion and low coupling, or other goals, but solutions offered by Information Expert (for example) are not appropriate? Sometimes OO design with a very tight relational gap leads to problems in terms of poor cohesion or coupling, or low reuse potential. They reiterated once more: sometimes a solution offered by Information Expert is not desirable. Even though the object is a candidate for the responsibility by virtue of having much of the information related to the responsibility, in other ways, its choice leads to a poor design, usually due to problems in cohesion or coupling.
**Solution:** assign a highly cohesive set of responsibilities to an artificial of convenience "behavior" class that does not represent a problem domain concept. Something made up, in order to support high cohesion, low coupling, and reuse. Such a class is a *fabrication* of the imagination. Ideally , the responsibilities assigned to this fabrication support high cohesion and low coupling, so that design of the fabrication is very clean, which is then know as a *pure fabrication*. Remember, the software is not designed to simulate the domain, but operate in it. But also remember, you need to balance these concerns and keep a low representational gap. Be aware, because pure fabrication is sometimes overused by those new to object design and more familiar with decomposing or organizing software in terms of functions. In the extreme, functions just become objects, which is very bad coupling. A symptom of this is that most of the data inside the objects being passed to other objects to reason with.
See example on page 421-423.
Object design can be broadly divided into two groups.

1. **Representational decomposition**: the software class is related to or represents a thing in a domain. It supports low representational gap.
2. **Behavioral decomposition**: assign responsibilities by grouping behaviors or by algorithm, without any concern for creating class with a name or purpose that is related to a real-world domain concept. Some classes exist as a convenience class conceived by the developer to group together some related behavior or methods.

**Benefits:**

- High Cohesion is supported because responsibilities are factored into a fine-grained class that only focuses on a very specific set of related tasks.
- Reuse potential may increase because of the presence of fine-grained Pure Fabrication classes whose responsibilities have applicability in other applications.

### 3.2.8 Indirection

**Problem:** where to assign a responsibility, to avoid direct coupling between two or more things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?
**Solution:** assign the responsibilities to an intermediate object to mediate between other components or services, so that they are not directly coupled. The intermediary creates an *indirection* between the other components. Example on p426.

Indirection has been one an old and very much used concept throughout computer science and programming history, in OOP an outside it. An old saying goes: "Most problems in computer science can be solved by another level of indirection" (David Wheeler).
**Benefits:**

- Lower coupling between components

### 3.2.9 Protected Variations

**Problem:** how to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
**Solution:** identify points of predicted variation or instability and assign responsibilities to create a stable "interface" around them. The term "interface" is used in the broadest sense of an access view. It does not literally only mean something like a Java Interface. Example on page 427-428 or slide 44 is good as well. This is a **very important**, fundamental principle of software design! Almost every software or architectural design trick in the book (data encapsulation, polymorphism, data-driven design, reflective or meta-level designs, Interpreter-driven designs, Uniform Access, standard languages, The Liskov Substitution Principle (LSP), interfaces, virtual machines, configuration files, operating systems,...) is a specialization of Protected Variations. See page 428-429 for more information on why these are specializations of Protected Variations.
Two points of change are worth defining where Protected Variation needs to be thought about:

**Variation point** Variations in the existing, current system or requirements.

**Evolution point** Speculative points of variation that may arise in the future but which are not present in the existing requirements. Beware: the cost of engineering protection at evolution points can be higher than reworking a simple (brittle) design. At evolution points, restraint and critical thinking is needed when applying Protected Variations.

Novice developers tend toward brittle designs.
Intermediate developers tend towards overly fancy and flexible, generalised designs (in ways that never get used).
Expert designers choose with insight, perhaps a simple and brittle design whose cost of change is balanced against its likelihood.
**Benefits:**

- Extensions required for new variations are easy to add.
- New implementations can be introduced without affecting clients.
- Coupling is lowered.
- The impact or cost of changes can be lowered.

Protected Variations is essentially the same as these older principles (and can help you understand it better):

- **Information Hiding**: hiding information about the design from other modules, at the points of difficulty or likely change. He once said about information hiding: "We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others." It is NOT data-encapsulation, which it is commonly misunderstood as.
- **Open-Closed Principle (OCP)**: modules should be both open (for extension; adaptable) and closed (the module is closed to modification in ways that affect clients). Modules include all discrete software elements, including methods, classes, subsystems, applications,..."closed in respect to X" means that clients are not affected if X changes. See page 434 for examples.

### 3.2.10 Don't Talk to Strangers

**Solution:** do not couple objects who have no obvious need to communicate. Avoid creating designs that traverse long object structure paths and send messages to distant, indirect (stranger) objects. Such design are fragile with respect to changes in the object structures, which is a common point of instability. Model relationships only if they are necessary to complete a use case. Example on page 430-431 and slide 48.
This was not included from the second edition of the book, because it is a special case of Protected Variations.
Messages should only be send to the following objects:

- The *this* object (or *self*) (=the object itself).
- A parameter of a method.
- An attribute of *this* (=of the object itself).
- An element of a collection which is an attribute of *this* (=of the object itself).
- An object created within the method.

The intend is to avoid coupling a client to knowledge of indirect objects and the object connections between objects. Direct objects are a client's "**familiars**", indirect objects are "**strangers**". A client should talk to familiars and avoid talking to strangers. The farther among a path the program traverses, the more fragile it is. This is because the object structure (the connections) may change. This is especially true in young applications or early iterations. In standard libraries, the structural connections between classes of objects are relatively stable. In mature systems, the structure is more stable. In new systems in early iteration, it isn't stable. Strictly obeying this law requires adding new public operations to the familiars of an object. These operations provide the ultimately desired information, and hide how it was obtained.
**Benefits:** the same as Protected Variations, because is a special case of it.

### 3.2.11 Cohesion and Coupling: Yin and Yang

Bad cohesion usually causes bad coupling and vice versa. They are the yin and yang of software engineering. When some class does many non-related things, it will often have many associations and such, and vice versa.

## 3.3 Summary

| Principle | Description |
|---|---|
| **Information Expert** | What is a general principle of object design and responsibility assignment?<br>Assign a responsibility to the information expert. This is the class that has the information necessary to fulfill the responsibility. |
| **Creator** | Who creates?(Note that Factory is a common alternate solution.)<br>Assign class B the responsibility to create an instance of class A if one of these is true:<br>1. B contains A<br>2. B compositely aggregates A<br>3. B has the initializing data for A<br>4. B records A<br>5. B closely uses A |
| **Controller** | What first object beyond the UI layer receives and coordinates ("controls") a system operation?<br>Assign the responsibility to an object representing one of the following:<br>1. Represents the overall "system", a "root object", a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*).<br>2. Represents a use case scenario within which the system operation occurs (a use-case or *session controller*) |
| **Low Coupling (evaluative)** | How to reduce the impact of change?<br>Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives. |
| **High Cohesion (evaluative)** | How to keep objects focused, understandable, and manageable, and as side-effect support Low Coupling?<br>Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives. |
| **Polymorphism** | Who is responsible when behavior varies by type?<br>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. |
| **Pure Fabrication** | Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?<br>Assign a highly cohesive set of responsibilities to an artificial of convenience "behavior" class that does not represent a problem domain concept. Something made up, in order to support high cohesion, low coupling, and reuse. |
| **Indirection** | How to assign responsibilities to avoid direct coupling?<br>Assign the responsibilities to an intermediate object to mediate between other components or services, so that they are not directly coupled. |
| **Protected Variations** | How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?<br>Identify points of predicted variation or instability and assign responsibilities to create a stable "interface" around them. |
| **Don't Talk to Strangers** | Do not couple objects who have no obvious need to communicate. Avoid creating designs that traverse long object structure paths and send messages to distant, indirect (stranger) objects. |

## 3.4 Remarks on this section from the slides

### 3.4.1 Start Up use case

"What to do when the system starts before any interaction". Make this diagram last. Create an initial design object which is at initialisation responsible for creation and initialisation for all design objects. This will be a facade controller. It is described by the create in interaction diagrams (what does this mean? Is said on slide 51) and in the Factory design. See the example on slide 52.

### 3.4.2 The connection between the presentation layer and the domain layer

The presentation-layer (GUI) must be able to be replaced by another! No domain-related responsibilities in the presentation layer. Must be completely decoupled. (All these thing have been said in the Controller principle?)

### 3.4.3 Visibility of objects and translation to the implementation

You want to translate visibility to the implementation (?). When an object X sends a message to an object Y, Y should be visible from X.
This can be done by:

- Y attribute of X.
  Most often used. (slide 55)
- Y is a parameter of a method of X.
  Is only temporary and passed along to an attribute. (slide 56)
- Y is a local object in a method of X.
- Y is a globally accessible object.

# 4 Design: Test-Driven Development

This section is not talked about in the course, but I think it is important to at least have a very short summary of the basics of it. It is talked about briefly in chapter 21.
Extreme Programming (XP) promotes, among other things, writing tests *first*. It also promotes continuously refactoring code to improve its quality by having less duplication, increased clarity,... Many OO developers swear by their value.
**Test-Driven Development (TDD)** covers more then just unit testing (= testing individual components). In OO unit testing TDD-style, test code is written *before* the class to be tested, and the developer writes unit testing code for nearly *all* production code. The basic rhythm is to write a little test code (e.g. for a method), then write a little production code, make it pass the test, then write some more test code, and so forth. The test is written first, imagining the code to be tested is written. It is important to note that we do NOT write all unit test for a class first. Rather, we write only one test method, implement the solution in the class to make it pass and then repeat.
Benefits:

- *The unit tests are actually written.*
- *Programmer satisfaction leading to more consistent test writing.* It is, due to human psychology, satisfying to write your test (like some kind of goal) and then realizing an implementation that satisfies it (meeting the goal). When writing everything and then implementing the tests has to opposite effects, it breaks something you just did. And for that to happen, you first of all need to fight a lot to not skip writing the tests (see the first point).
- *Clarification of detailed interface and behavior.* It makes the goals, behavior, interface much clearer and provides great reflection upon it before actually writing the code. When writing the test code, you need to imagine that the object code exists.

- *Provable, repeatable, automated verification.* Hundred or thousands of unit tests that build up over the weeks provides some meaningful verification of correctness. And because they run automatically, it is easy. Over time, as the test base build from 10 to 50 to 500 tests, the early, more painful investment in writing tests starts to really feel like it is paying off as the size of the application grows.
- *The confidence to change thing.* When a developer needs to change existing code, there is a unit test suite that can be run, providing immediate feedback if the change caused an error.

The most popular unit testing framework is the xUnit family (for many languages). JUnit for java, NUnit for .NET,... See the example JUnit on page 387-388.

## Conclusion

So, was this summary worth your time?