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)

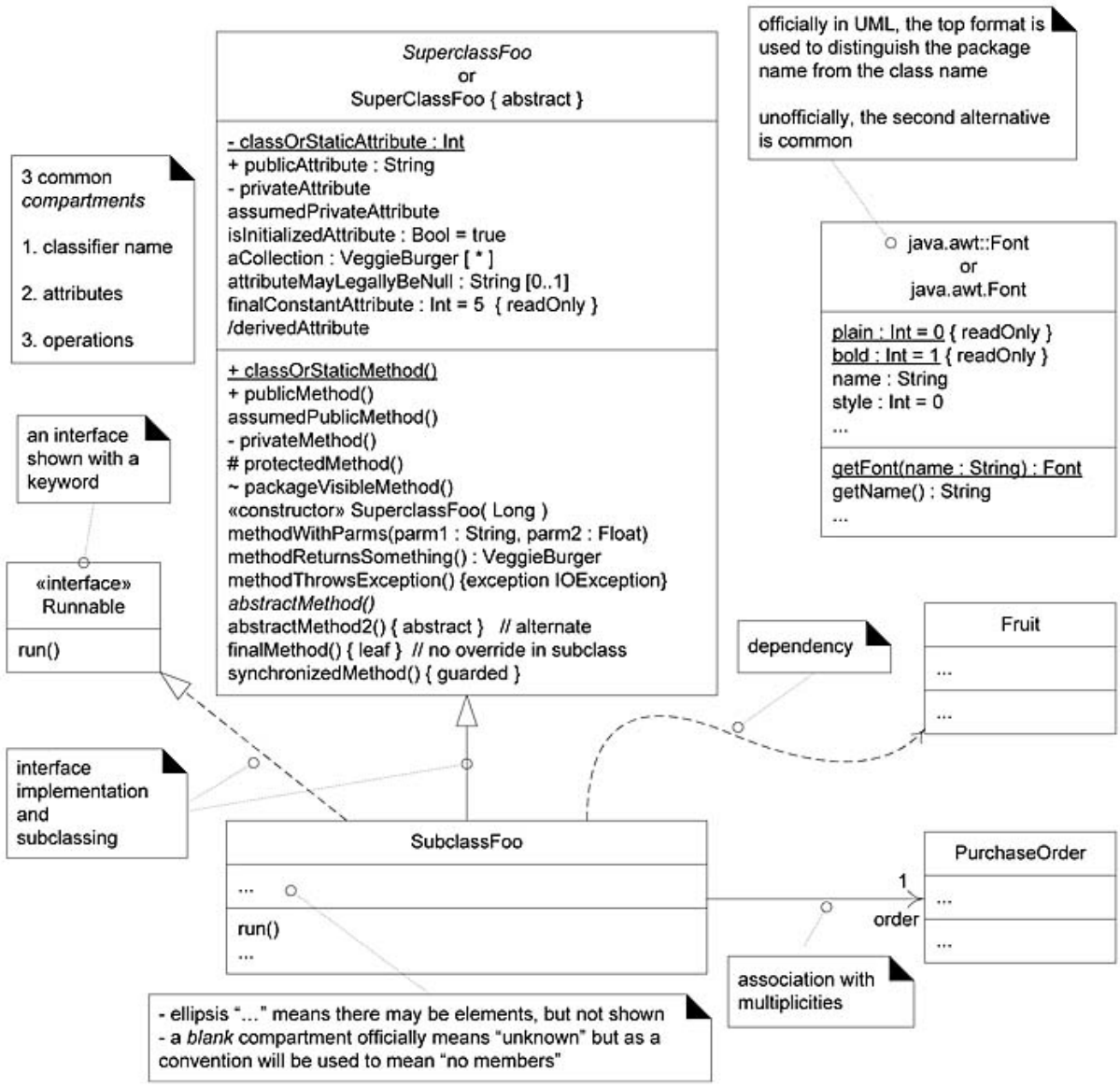
There was only one extra guideline here, and it is that for if = opt squares, for 1 arrow statements you can precede the name of the message with the condition between square brackets. This is not supported anymore in UML 2, so you should not use it, but some people still do, that is why you need to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

2.1.4 Communication VS. Sequence

Type	Strengths	Weaknesses
Communication	• Compact • Flexibility to add new objects	• More difficult to see sequence of messages • Less notation options in UML
Sequence	• Clearly shows sequence or time ordering of messages • Large set of detailed notations in UML because it is more widely used	• Forced to extend to the right when adding new objects • 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.



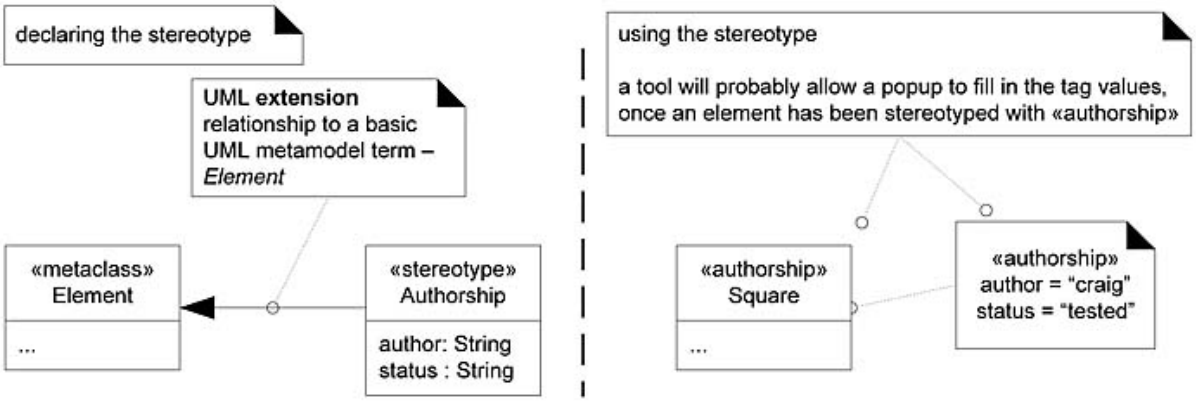
Figuur 1: The summary provided by the book.

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? 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.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B compositely aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? 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 (a use-case or *session controller*)
Low Coupling (evaluative)	How to reduce the impact of change? 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? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
Polymorphism	Who is responsible when behavior varies by type? 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? 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? 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? 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.

5 GoF Design Patterns

Consider consulting wikipedia for good and up to date examples.
In the preface, the authors say the following: "Don't worry if you don't understand this book completely on the first reading. We didn't understand it all on the first writing! Remember that this isn't a book to read once and put on a shelf. We hope you'll find yourself referring to it again and again for design insights and for inspiration." and "We don't consider this collection of design patterns complete and static; it's more a recording of our current thoughts on design." The simplest and most common patters are:

- Abstract Factory
- Adapter
- Composite
- Decorator
- Factory Method
- Observer
- Strategy
- Template Method

A design patter has four essential elements:

1. **Patter name:** the name (handle) of the pattern.
2. **Problem:** describes when to apply the pattern. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. **Solution:** the elements that make up the design, their relationships, responsibilities and collaborations. The solution does not describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.
4. **Consequences:** the results and trade-offs of applying the pattern. The consequences are described in terms of space and time trade-offs and the flexibility, extensibility and portability of the system,

Page 6 explains the sections of the framework in which the patterns are explained. Page 8 contains a summary and the starting page of all patterns.
Patterns are classified by two criteria:

1. **Purpose:** what the pattern does. There are 3 of them:
    - **Creational Patterns:** These patterns are concerned with object creation.

- **Structural Patterns:** These patterns are concerned with the composition of classes or objects.
- **Behavioral Patterns:** These patterns characterize the ways in which classes or objects interact and distribute responsibility.

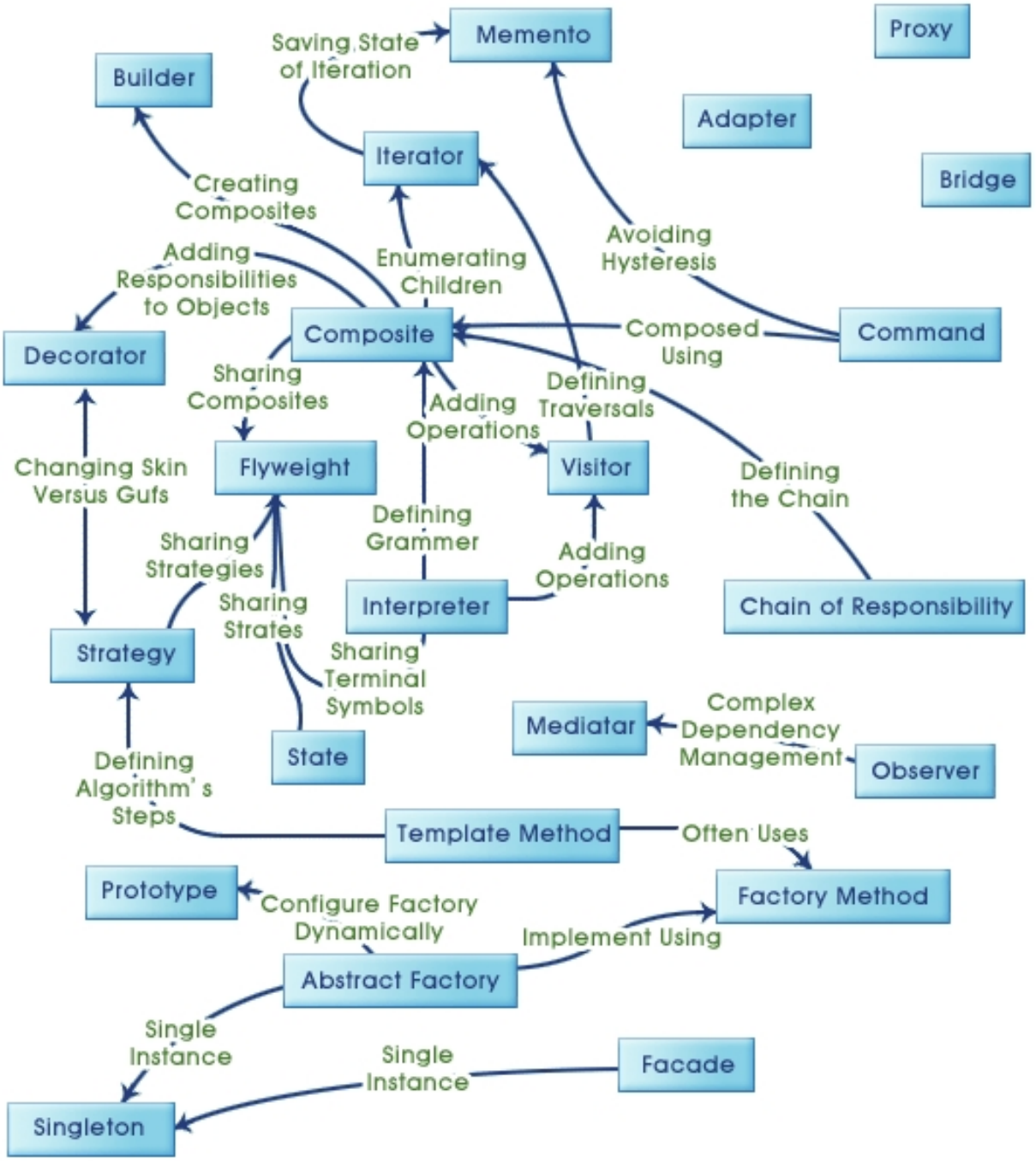
2. **Scope:** specifies whether the pattern applies primarily to classes or to objects. Class patters deal with relationships between classes and their subclasses. Object patterns deal with object relationships.

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figuur 3: The categorisation of the patterns.

Dynamic binding The run-time association of a request to an object and one of its operations.



Figuur 4: The relations of the patterns.

5.1 How to select a design pattern

1. Read through each pattern's intent to find one or more that sound relevant to your problem. You can use the categories of patters to narrow down your search (see 3).
2. Study the relationships between patterns to select a pattern or a group of patterns (see 4).
3. Study the summary of the patterns at the start of the section of their purpose.
4. See page 24 and on for common causes of redesign and see which patterns apply to your problem.

5. Consider what you want to be able to change without redesign.

5.2 How to use a design pattern

1. Read the pattern once through for an overview. Pay particular attention to the applicability and consequences sections to ensure the patters is right for your problem
2. Go back and study the Structure, Participants and Collaboration sections.
3. Look at an example.
4. Choose name for pattern participants that are meaningful in the application context.
5. Define the classes.
6. Define application-specific names for operations in the pattern.
7. Implement the operations to carry out the responsibilities and collaborations in the pattern.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

Figuur 5: The variations in the patterns.

6 Creational Patterns

These patterns are concerned with object creation. They help make a system independent of its objects are created, composed an represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object. Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hardcoding a fixed set of behaviours toward defining a smaller set of fundamental behaviours that can be composed into any number of more complex ones. Thus creating objects with particular behaviours requires more than simply instantiating

a class. There are two recurring themes in these patterns. First, they encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in what get created, who creates it, how it gets created, and when. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be static (compile-time) or dynamic (run-time).

6.1 Abstract Factory

This is an object creational pattern. See p87 and on.
Abstract Factory has the factory object producing objects of several classes.

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also known as

Kit.

Motivation

Consider the problem of a GUI which can have multiple styles but always the same elements. You cannot hardcode the look/style because there are multiple of them. So make an abstract factory WidgetFactory and an abstract class for each type of widget. These serve as interfaces to the user of the classes. Make concrete subclasses implementing the specific styles. For each style you make a subclass of the widgetfactory implementing the abstract methods which return the specific style widgets.

Applicability

Use the abstract Factory pattern when:

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure

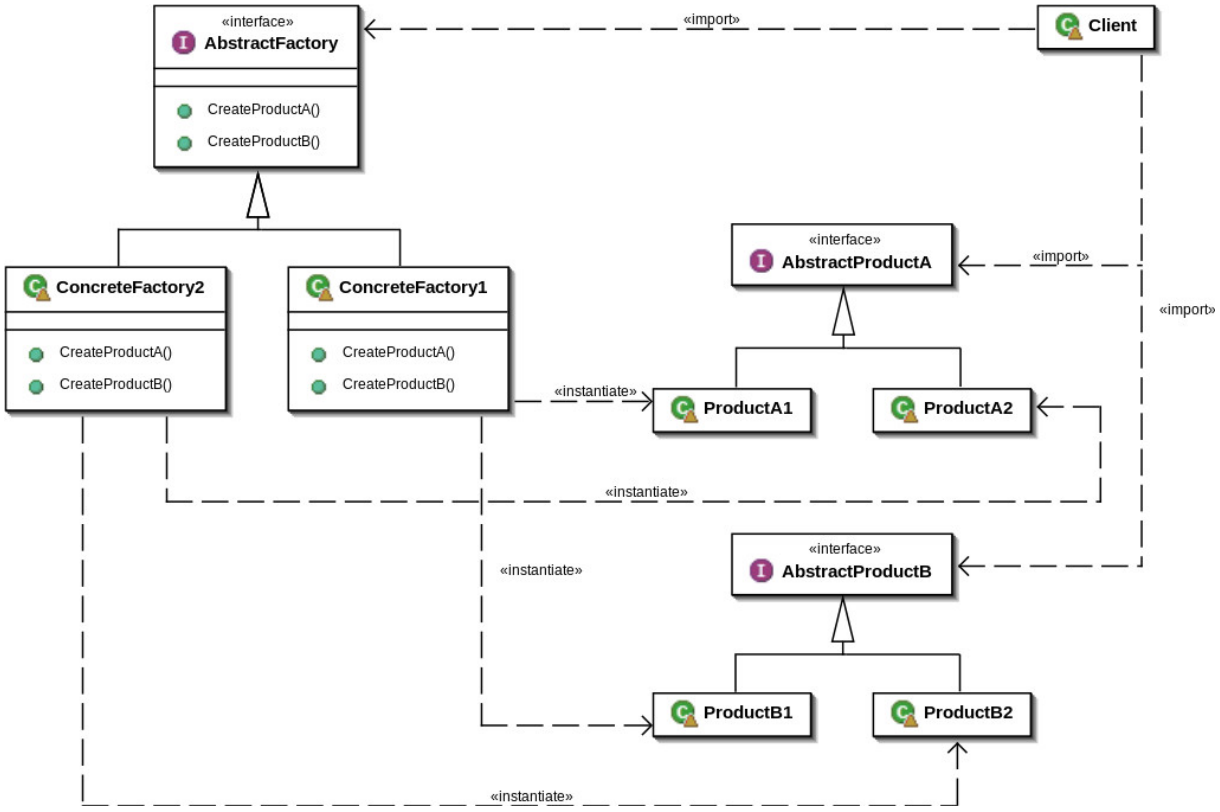
See figure 6.

Participants

- **AbstractFactory:** declares an interface for operations that create abstract product objects. (WidgetFactory)
- **ConcreteFactory:** implements the operations to create concrete product objects. (specific style Factory)
- **AbstractProduct**: declares an interface for a type of product objects. (e.g. window, scrollbar, button)
- **ConcreteProduct**: defines a product object to be created by the corresponding concrete factory and it implements the AbstractProduct.
- **Client**: uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

One concrete factory creates product objects having a particular implementation. AbstractFactory defers creation of product objects to its ConcreteFactory subclass.



Figuur 6: The structure of the pattern.

Consequences

Advantages:

1. It isolates concrete classes.
2. It makes exchanging product families easy, by changing the used concrete factory.
3. It promotes consistency among products. When object families are designed to only use objects of the same family, AbstractFactory makes this easy to enforce.

Disadvantages:

1. Supporting new kinds of products is difficult, because adding new products (e.g. widgets) requires adding them to the abstractfactory and all its subclasses. However, the next section has a solution for this.

Implementation

Here are some useful techniques for implementing the abstract factory pattern:

1. The following is shitty explained, of course you will always only use one concrete factory for a task at a time. A concrete factory is often implemented as a singleton because only one family is used at a given time. However, the professor told us to never use singleton, but I think it does not apply here.
2. The concrete products are only created by their concrete factories using a factory method, described in the factory pattern.
3. If many product families ar possible, the concrete factory can be implemented using the prototype pattern. The concrete factory is initialised with a prototypical instance of each product in the

family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new factory class for each new product family. A variation on this is storing classes in a concrete factory that create various new instances on behalf of the factory. You define a new factory by initialising an instance of a concrete factory with *classes* of products rather than by subclassing.

4. Defining extensible factories. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it. A more flexible *but less safe* design is to add a parameter to operations that create objects. This parameter specifies the kind of object to be created. It could be a class identifier, integer, string, or anything else that identifies the kind of product. In fact with this approach, AbstractFactory only needs a single "Make" operation with a parameter indicating the kind of object to create. This is the technique used in the Prototype- and the class-based abstract factories in the point above.

Sample code

Pages 92-94.

Known uses

The examples given are from the 90's...

Related patterns

AbstractFactory classes are often implemented with factory methods (Factory), but they can also be implemented using Prototype (Prototype). A concrete factory is often a singleton (Singleton).

6.2 Builder

This is an object creational pattern. See p97 and on.
Builder has the factory object building a complex product incrementally using a correspondingly complex protocol.

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Also known as

Kit.

Motivation

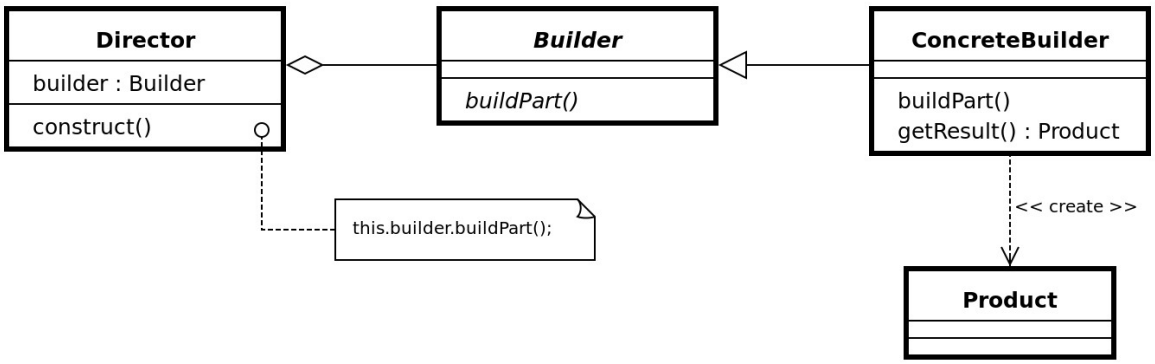
Consider a UTF-8 text document reader which should be able to convert the text into other formats such as ASCII or a text widget that can be edited interactively. Because there will always be new formats, adding a new conversion should be easy without modifying the reader. We can configure the Reader class with a TextConverter object that converts UTF-8 to another representation. When a UTF-8 symbol or control word is read, it issues a request to the TextConverter to convert it. Textconverter objects are responsible for performing the data conversion and for representing the token in a particular format. Subclasses of TextConverter specialize in different conversions and formats. Each converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface. Each converter class is called a **builder** in the pattern, and the reader is called the **director**.

Applicability

Use the Builder pattern when:

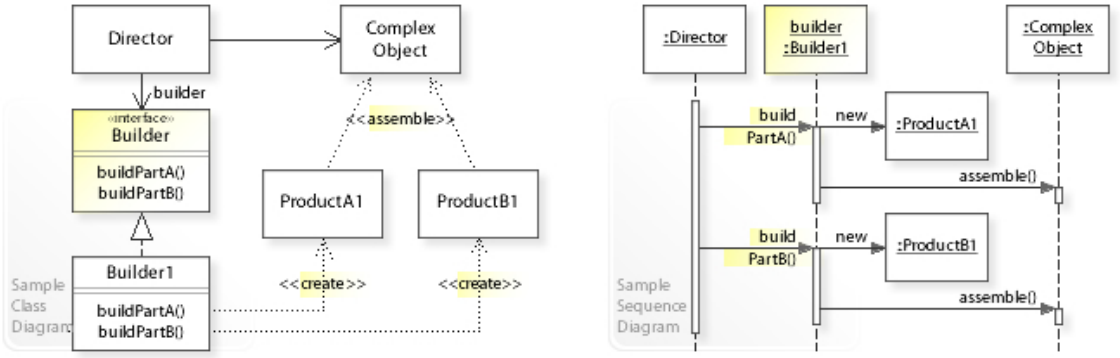
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.

Structure



Figuur 7: The structure of the pattern.

See figure 7 and 8.



Figuur 8: The structure and SSD of the pattern.

Participants

- **Builder**: specifies an abstract interface for crating parts of a Product object. (TextConverter)
- **ConcreteBuilder**: constructs and assembles parts of the product by implementing the Builder. It defines an keeps track of the representation it creates. It provides an interface for retrieving the product. (ASCIIConverter)
- **Director**: constructs an object using the Builder interface. (Reader)
- **Product**: represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it is assembled. It includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations

See figure 8.

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences

Advantages:

1. It lets you vary a product's internal representation and assembly.
2. It isolates/encapsulates code for construction and representation of a complex object (e.g ascii, LaTeXor odf format).
3. It fives you finer control over the construction process, because the builder can construct it in multiple steps and the director only wants the final product of the step.

There are no disadvantages specified.

Implementation

Typically there is an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default. A ConcreteBuilder class overrides operations for components it is interested in creating.
Here are other implementation issues to consider:

1. Because the builder provides abstract methods for building step by step, they must suffice for all kinds of builders. It is key to design the assembly and construction interface well.
2. You should not make an abstract class for products, because they normally vary a lot and do not need one because often the director is configured with the proper concrete builder and thus knows which kind of product to expect and handle.
3. Builder methods should not be abstract and just empty, because like this the concrete builder can only override the functions it really needs. (My note: why not just override the unnecessary abstract methods with empty methods)

Sample code

Pages 101-105 or Wikipedia.

Known uses

Parsers.

Related patterns

A Composite (Composite) is what the builder often builds. The Builder pattern may look a bit like the Abstract Factory pattern, but builder focuses on step by step construction and returns the product in the final step, while AF focuses on families of objects and returns the product immediately.

6.3 Factory Method

This is a ***class*** creational pattern. See p107 and on.

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also known as

Virtual Constructor.

Motivation

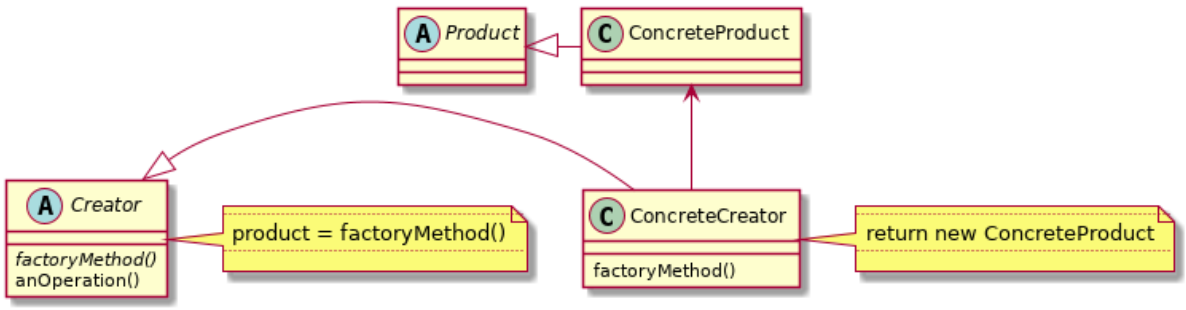
Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass to realize their application-specific implementations. The application class only knows when and not which kind of document should be created. Problem: the framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

Applicability

Use the Factory Method pattern when:

- A class can't anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects its creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Figuur 9: The structure of the pattern.

See figure 9.

Participants

- **Product**: defines the interface of objects the factory method creates. (Document)
- **ConcreteProduct**: implements the Product interface (= this means that it is a non-abstract subclass of Product). (MyDocument)
- **Creator**: declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object. (Application)

- **ConcreteCreator**: overrides the factory method to return an instance of a ConcreteProduct. (MyApplication)

Collaborations

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

Advantages:

1. No need for application-specific classes (for the user) because the code only deals with the product interface.
2. Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook(?) for providing an extended version of an object.
3. Connects parallel class hierarchies.

Disadvantages:

1. Clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution. So basically don't use it if you'll never need it?

Implementation

Consider the following issues when applying the Factory Method pattern:

1. Think well about whether or not to provide a default implementation for the factory method.
2. Whether or not to use *parameterized factory methods*. Therein, the factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface. Base your decision on GRASP.
3. Language specific variants and issues.
4. Whether or not to use templates to avoid subclassing. Base your choice on GRASP, this book has had much critique.
5. As a naming convention, make it clear you are using factory methods.

Sample code

Pages 114-115.

Known uses

Toolkits and frameworks are full of factory methods.

Related patterns

Abstract Factory (Abstract Factory) is often implemented with factory methods. Factory methods are usually called within Template Methods (Template Methods). Prototype (Prototype) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

6.4 Prototype

This is an object creational pattern. See p117 and on.
Prototype has the "factory object" building a product by copying a prototype object.

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Also known as

It is only known as Prototype.

Motivation

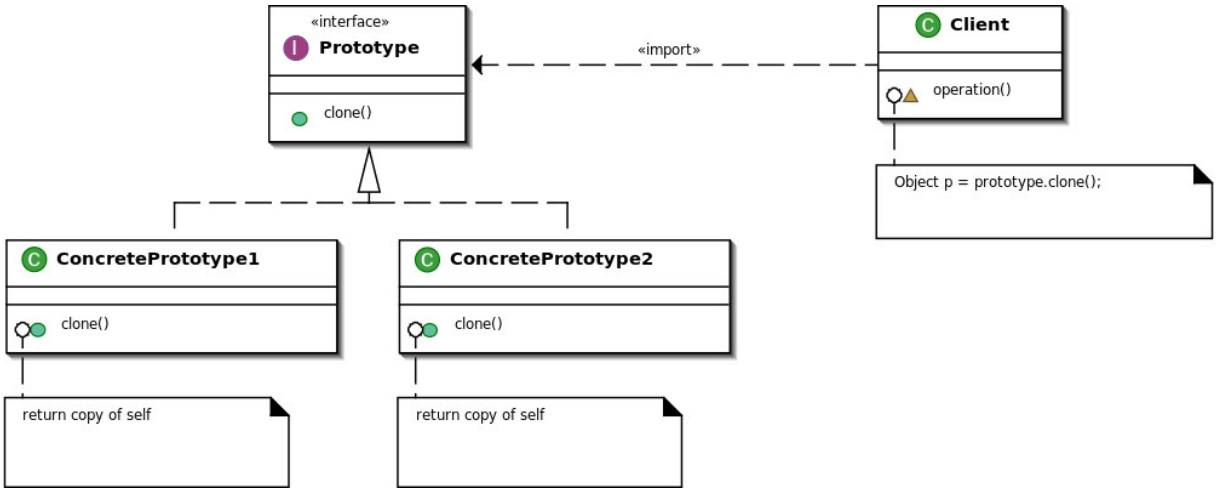
Consider an editor for music "partituren" (like ableton but with old school do-re-...) that you want to implement with an existing graphical framework. Say there is an abstract class for graphical components called Graphic and for tools called Tool. It also has a subclass GraphicTool for tools that create instances of graphical objects and add them to the document. To prevent having to make a subclass for every type of music object e.g. note, stave, ... we can creating a new instance of GraphicTool by copying/cloning an instance of a Graphic subclass. The instance is called a **prototype**. GraphicTool takes as parameter the prototype it should clone and add to the document. So each tool for creating a music objects is an instance of GraphicTool that is initialized with a different prototype. I did not understand this motivation while writing it.

Applicability

Use the Prototype method when a system should be independent of how its products are created, composed and represented AND one of the following:

- when the classes to instantiate are specified at run-time
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products
- when instances of class can have one of only a few different combinations of state.

Structure



Figuur 10: The structure of the pattern.

See figure 20.

Participants

- **Prototype**: declares an interface for cloning itself. (Graphic)
- **ConcretePrototype**: implements an operation for cloning itself. (WholeNote,HalfNote, Staff)
- **Client**: creates anew object by asking a prototype to clone itself.

Collaborations

A client asks a prototype to clone itself.

Consequences

Advantages:

1. Hides the concrete product classes from the client.
2. The client can work with application-specific classes without modification.
3. Adding or removing products at run-time. Prototype lets you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. This is very flexible.
4. Specifying new objects by varying values. (e.g. circle diameter (it think)) This lets you define new behaviour without defining new classes.
5. Specifying by varying structure. Many applications build objects from parts and subparts. You can define a new prototype by combining subparts.
6. Reduced subclassing.
7. Configuring an application with classes dynamically.

Disadvantages:

- Each subclass of Prototype must implement the Clone operation, which can be difficult if the class already exists and it uses objects that do not support cloning.

Implementation

1. Using a prototype manager: when the number of prototypes is not fixed, keep a registry (= prototype manager) of available prototypes. The prototype manager returns a prototype for a given key.
2. Implementing the cloning operation.
3. Initializing clones with values. The caller can set the values on the clone after cloning or you can provide an initialise method (badly explained, see examples).

Sample code

Pages 122-125.

Known uses

Debuggers and sketchers.

Related patterns

Prototyping and Abstract Factory are competing patterns but can also be used together because an AF can store a set of of prototypes to clone and return. Designs that make heavy use of Composite and Decorator patterns often can benefit from Prototype as well.

6.5 Singleton

NEVER USE THIS.
p127

6.6 Discussion of Creational Patterns

There are two categories of classes of objects a system creates:

- Subclass the class that creates objects: Factory Method. The main drawback of this approach is that it can require creating a new subclass just to change the class of the product.
- Define an object that is responsible for knowing the class of the product objects, and make it a parameter of the system: Abstract Factory, Builder and Prototype. All three involve creating a new "factory object" whose responsibility is to create product objects.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation. People often use Factory Method as the standard way to create objects, but it is not necessary when the class that is instantiated never changes or when instantiation takes place in an operation that subclasses can easily override, such as an initialization operation. Designs that use Abstract Factory, Builder or Prototype are even more flexible than those that use Factory Method, but they are also more complex. Often, designs start out using Factory Method and evolve toward the other creational patterns as the designer discovers where more flexibility is needed. Knowing many design patters gives you more choices when trading off one design criterion against another.

7 Structural Patterns

These patterns are concerned with the composition of classes or objects. These patterns are concerned with how classes and objects are composed to form larger structures.
There are 2 types:

1. **Class**: use inheritance to compose interfaces or implementations. These can not be changed at runtime.
2. **Object**: describe ways to compose (= write) objects to realize new functionality. This compositions can be changed at runtime.

The Composite Pattern describes how to build a class hierarchy made up of classes for two kinds of objects: primitive and composite. The composite objects let you compose primitive and other composite objects into arbitrarily complex structures. In the Proxy pattern, a proxy acts as a convenient substitute or placeholder for another object. A proxy can be used in many ways. It can act as a local representative for an object in a remote address space. It can represent a large object that should be loaded on demand. It might protect access to a sensitive object. Proxies provide a level of indirection to specific properties of objects. Hence they can restrict, enhance, or alter these properties.
The Flyweight patter defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency. Applications that use lots of objects must pay careful attention to the cost of each object. Substantial savings can be had by sharing objects instead of duplicating them. But objects can be shared only if they do not define context-dependent state. Flyweight objects have no such state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, Flyweight objects may be shared Freely.
Whereas Flyweight show how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. A facade is a representative for a set of objects. The facade carries out its responsibilities by forwarding messages to the objects it represents. The Bridge pattern separates an object's abstraction from its implementation so that you can vary them independently.
Decorator describes how to add responsibilities to objects dynamically. Decorator composes objects recursively to allow an open-ended number of additional responsibilities. For example, a Decorator object containing a user interface component can add a decoration like a border or a shadow to the component, or it can add functionality like scrolling or zooming. We can add two decorations simply by nesting one Decorator object within another, and so on for other decoration. To accomplish this, each Decorator object must conform to the interface of its component and must forward messages to it. The Decorator can do its job (such as drawing a border around the component) either before or after forwarding a message.
Many of these patterns are related, which is discussed at the end of this section.

7.1 Adapter

This is a class or object structural pattern. See p139 and on.
This provides a link to use e.g. reused code with another syntax. The adapter has both a *class and object* version.

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Also known as

Wrapper.

Motivation

Sometimes a toolkit class that is designed for reuse is not reusable only because its interface does not match the domain-specific interface an application requires. Consider a drawing editor for graphical elements. The interface for these elements is an abstract class called Shape which has a subclass for every element, e.g. LineShape. A shape for text is complicated however and we would like to use the TextView class of a reusable toolkit to implement TextShape. We do not have the source code for TextView, and even if we did, it would be more economical to still not copy, paste and adapt but to use this pattern. You can do this in two ways:

- **Class** way: inheriting Shape's interface an TextView's implementation.
- **Object** way: composing a TextView instance within TextShape and implementing TextShape in terms of TextView's interface.

TextShape is called an **adapter**.

Applicability

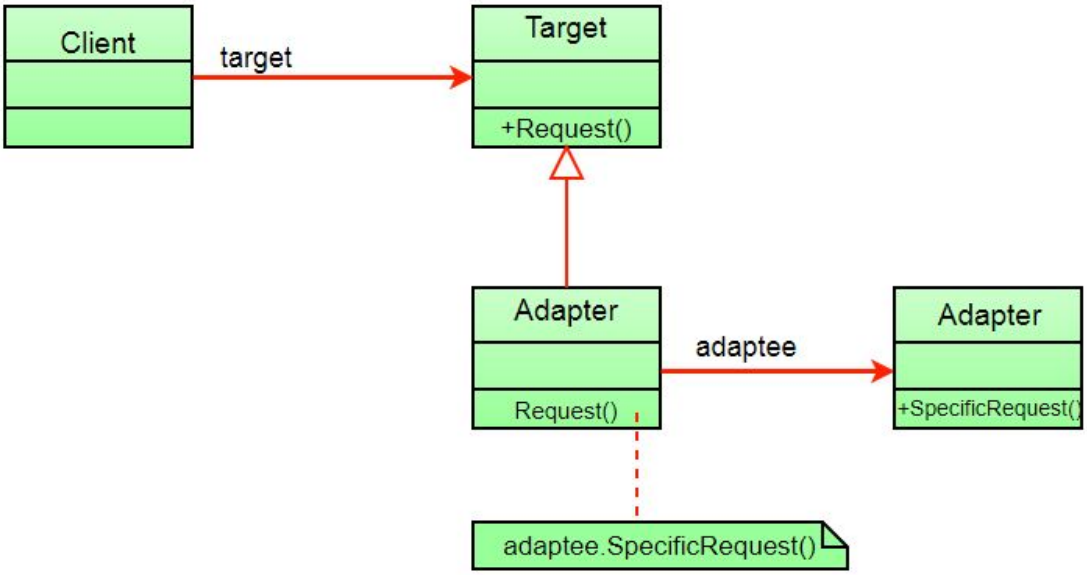
Use the adapter pattern when:

- You want to use an existing class, and its interface does not match the one you need.
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that do not necessarily have compatible interfaces.
- *Object adapter only*: you need to use several existing subclasses, but it is impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

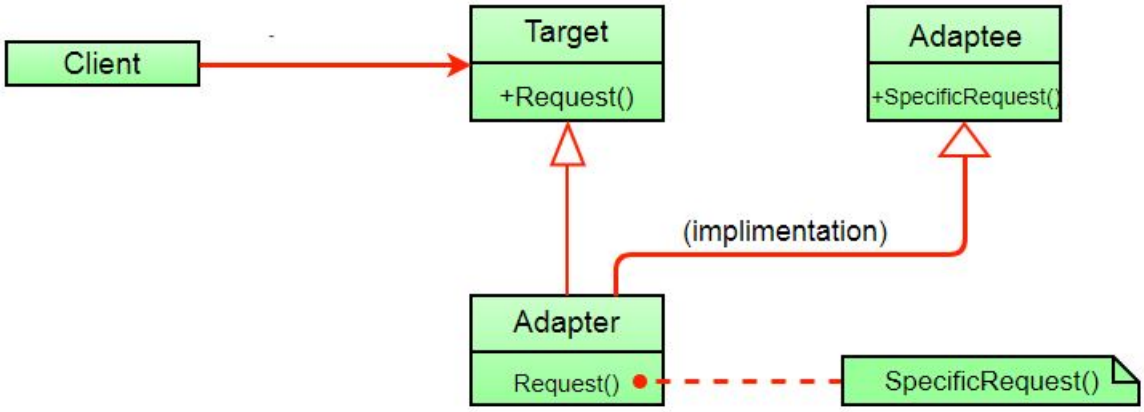
Structure

See figure 11 and 12.

Class version uses inheritance from multiple classes, which we learned in OGP should NEVER be done. ALWAYS use the object version. I will include it just for completeness.
There is also such a thing as a **two way adapter**, an adopter which subclasses from more then one target so you can use even more kinds of interfaces. In my opinion this is messy and of course, it uses the devilish multiple inheritance.



Figuur 11: The structure of the object version of the adapter pattern.



Figuur 12: The structure of the class version of the adapter pattern.

Participants

1. **Target**: defines the domain-specific interface that Client uses. (Shape)
2. **Client**: collaborates with objects conforming to the Target interface. (DrawingEditor)
3. **Adaptee**: defines an existing interface that needs adapting. (TextView)
4. **Adapter**: adapts the interface of Adaptee to the Target interface. (TextShape)

Collaborations

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

- **Object**
  Advantages:

  1. Adapts Adaptee to Target.
  2. Works with Adaptee and all of its subclasses.
  3. Adapter can add functionality to all Adaptees (= it and its subclasses) at once.

  Disadvantages:

  1. You are not able to override Adaptee behaviour. However, it can be overcome by doing the following: it would require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

- **Class**
  Advantages:

  1. Adapts Adaptee to Target.
  2. Adapter can override some of Adaptee's behaviour, because Adapter is a subclass of Adaptee.
  3. Introduces only one object, and no additional pointer indirection is needed to get to Adaptee.

  Disadvantages:

  1. Cannot be used to adapt a class *and* all its subclasses, because it adapts Adaptee to Target by commiting to a concrete Adaptee class.
  2. **MULTIPLE INHERITANCE!!!**

Implementation

Issues to keep in mind:

1. In C++ Adapter would inherit publicly from Target and privately from Adaptee, so Adapter would be a subtype of Target but not of Adaptee.
2. Construct **pluggable adapters**. This is an adapter which has a built-in adapter interface. There are three ways to do this:
   - Using abstract operation: define corresponding abstract operations in the client, which the adapter implements so they can be used with the adaptee.
   - Using Delegate objects (see p145).
   - Parameterised adapters.

Sample code

Pages 146-148.

Known uses

Drawing applications.

Related patterns

Bridge has a structure similar to an object adapter, but bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an *existing* object.
Decorator enhances another object without changing its interface. A decorator is thus more transparant to the application than an adapter is. As a consequence, Decorator supports recursive composition, which is not possible with pure adapters. Proxy defines a representative or substitute for another object and does not change its interface.

7.2 Bridge

This is an object structural pattern. See p151 and on.

Intent

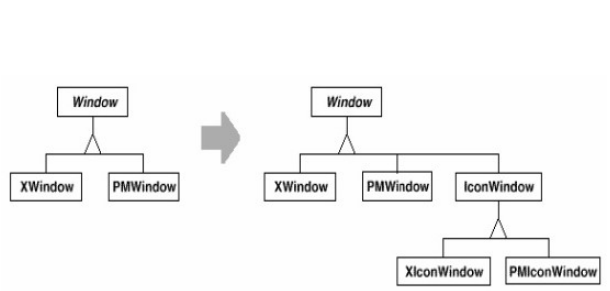
Decouple an abstraction from its implementation so that the two can vary independently.

Also known as

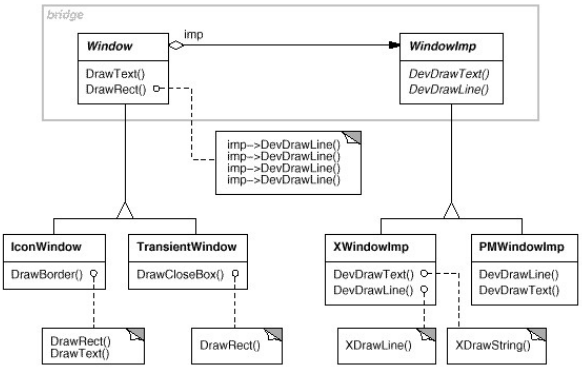
Handle or Body.

Motivation

Sometimes inheritance is not flexible enough when you want an abstraction which can have one of several possible implementation. Consider an abstract window Window which should work in the Xwindow and IBM PM, with respective subclasses XWindow and PMWindow. This has 2 drawbacks:



(a) Problem



(b) Solution

1) when you want a new subclass of window for new functionality, you would have to subclass it again to implement both platforms. See 13a.
2) It makes the code of the person using a window platform depended. This makes the code brittle and a pain to expand.
Instead of subclassing, you can put the window and its implementations in separate class hierarchies. One for window and its functional subclasses (for example: TransientWindow) and one for WindowImp for specific platform implementations subclasses. All operations on Window subclasses are implemented in terms of abstract operations from the WindowImp interface. For example: a drawRectangle function in Window import basic elements like drawLine from WindowImp. See figure 13b.
This relationship is known as a **bridge**.

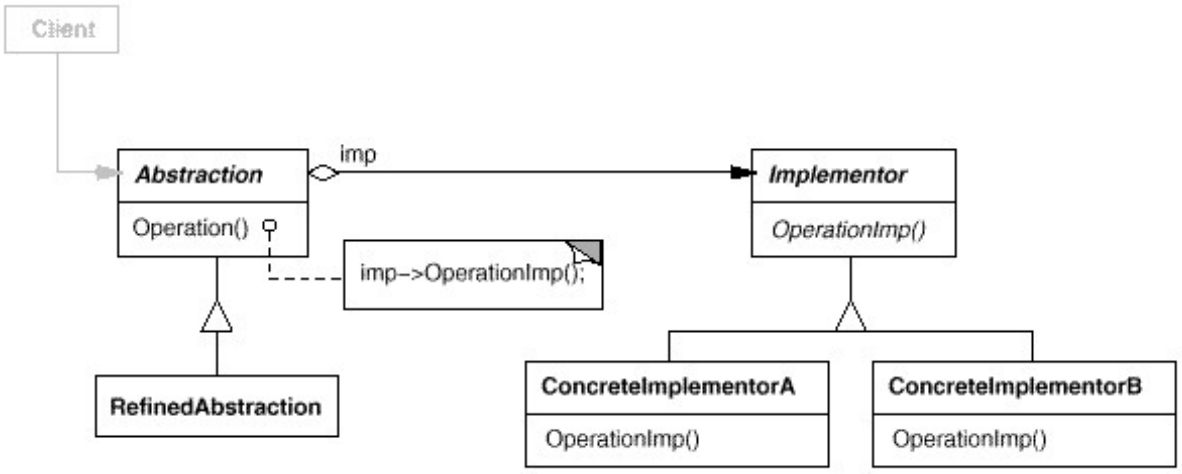
Applicability

Use the bridge pattern when:

- You want to avoid permanent binding between an abstraction and its implementation. (e.g. implementation switch at runtime)
- Both the abstractions and their implementations should be extensible by subclassing.
- Changes in implementation should have no impact on clients, there code should not have to recompile.
- You want to hide the implementation of an abstraction completely from clients.
- You have a growing number of classes as in the motivation (?).

- You want to share an implementation among multiple objects and this should be hidden from the client.

Structure



Figuur 14: The structure of the bridge pattern.

Participants

- **Abstraction**: defines the abstraction interface and maintains a reference to an object of type Implementor. Often contains the higher-level operations. (Window)
- **RefinedAbstraction**: extends the interface defined by Abstraction. (TransientWindow)
- **Implementor**: Defines the interface for the implementation classes. This is often very different from the Abstraction interface and has low-level primitive operations. (WindowImp)
- **ConcreteImplementor**: implements the Implementor interface and defines its concrete implementation. (XWindowImp, PMWindowImp)

Collaborations

Abstraction forwards client requests to its Implementor object.

Consequences

Advantages:

1. Decoupling interface and implementation.
2. You can extend the Abstraction and Implementor hierarchies independently.
3. Hiding implementation details from clients.

Implementation

Consider the following issues when applying the bridge pattern:

1. In situations where there is only one implementation, creating an abstract Implementor class is not necessary. This is still useful when a change in implementation of a class must not affect its existing client.
2. Creating the right implementor object.
3. Sharing implementors.

Sample code

Pages 156-160.

Known uses

Windows, Handle classes, sets/lists, images.

Related patterns

An Abstract Factory can create and configure a particular Bridge. The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they are designed. Bridge is used up-front in a design to let abstractions and implementations vary independently.

7.3 Composite

This is an object structural pattern. See p163 and on.

Intent

Compose objects into tree structures to represent part-whole hierarchies = recursive composition (p36). Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

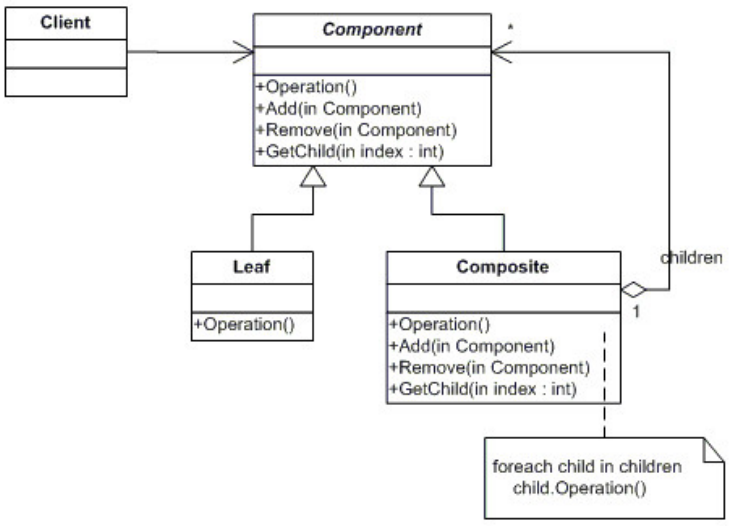
Graphic apps like drawing editors let users build complex diagrams out of simple components. The user can group components to form larger components =(container), which can be grouped to from still larger components. The problem is that these groups and primitive objects must be treated differently, even if most of the time they are treated the same. This makes the application more complex. The composite pattern describes how to use recursive composition so that clients do not have to make this distinction. It contains an abstract class representing both primitives and their containers=groups. In this example, the abstract class is Graphic which declares the common operations (like draw()) on primitives and groups. The subclasses for primitives are Line, Rectangle, Text,... which implement the draw() operation. The group/container is a class Picture which holds its Graphic objects and implements draw() by calling draw on al the Graphics it holds.

Applicability

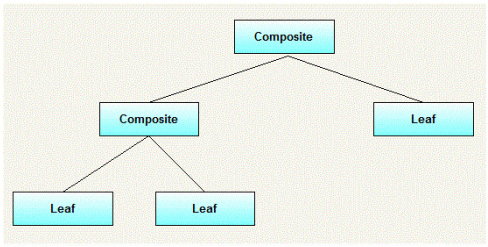
Use the composite pattern when:

- You want to represent recursive composition (see page 36-38).
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composition uniformly.

Structure



(a) The structure of the composite pattern.



(b) A runtime example of objects from the pattern.

Participants

- **Component**: declares the interface for objects in the composition. It implements the default behavior for the interface common to all subclasses (of course...) and declares an interface for accessing and managing its child components (add/remove/...). It is optional to define an interface for accessing a component's parent in the recursive structure (isn't this very dirty?). (Graphic)

- **Leaf**: represents leaf objects in the composition an has no children. It defines behaviour for primitive objects in the composition. (Line, Rectangle, Text,...)
- **Composite**: stores child components and defines behaviour for components having children. It implements child-related operations in the Component interface. (Picture)
- **Client**: manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, the request is handled directly. If it is a Composite, it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

Advantages:

1. You can solve the primitive/group problem.
2. Very simple to use for the client.
3. Easy to add new components. New Leafs work instantly with Composites.

Disadvantages:

1. It is harder to place restrictions on the components of a composite (e.g. only a certain type).

Implementation

There are many issues to consider when implementing the Composite pattern:

1. Explicit parent references (implemented in the Component class). This makes it easy to traverse and support the Chain of Command pattern. It is essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children. You do this by changing the parent when the component is added or removed. These functions are in the Component class.
2. Sharing components (especially Leafe objects) using the Flyweight Pattern, but losing the ability of a parent reference.
3. Defining as many common operations for Composite and Leaf in Component as possible. Solving problems such as getChild() for a Leaf by it never returning any children.
4. Doing one of these two.
   - Defining child management operations (add/remove/...) in Component and making them meaningful for Leaf. This gives the ability to treat all components uniformly. It costs safety because clients may try to do meaningless operations like for example add components to a Leaf.
   - Only declare and define them in Composite. It adds safety because no meaningless operations can be done. However, Leaf and Composite now have a different interface. You would also need extra for setting a Composite to a leave an vice versa.

   The first option is the best. You can make Leaf just throw an exception in the child management operations.
5. Only Composite should have an attribute with the list of child nodes.
6. Implementing a type of order in the children, often using the Iterator pattern.
7. Cache traversal or search info to improve performance.

8. In languages without garbage collection, it is usually best to make Composite responsible for deleting it's children when destroyed.

9. Considering the best data structure to store children. This can be a general data structure such as a list. It can also be attributes of a composite an making subclasses for different types of composites, which can be implemented with the Interpreter interface.

Sample code

Pages 170-172.

Known uses

They are found everywhere. Compilers.

Related patterns

Often the component-parent link is used for a Chain of Responsibility. Decorator is often used with Composite, having a common parent class. So Decorators will have to support the Component interface with operations like Add, Remove and GetChild. Flyweight lets you share components, but they can no longer refer to their parents. Iterator can be used to traverse composites. Visitor localizes operations and behaviour that would otherwise be distributed across Composite and Leaf classes.

7.4 Decorator

This is an object structural pattern. See p175 and on.

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also known as

Wrapper

Motivation

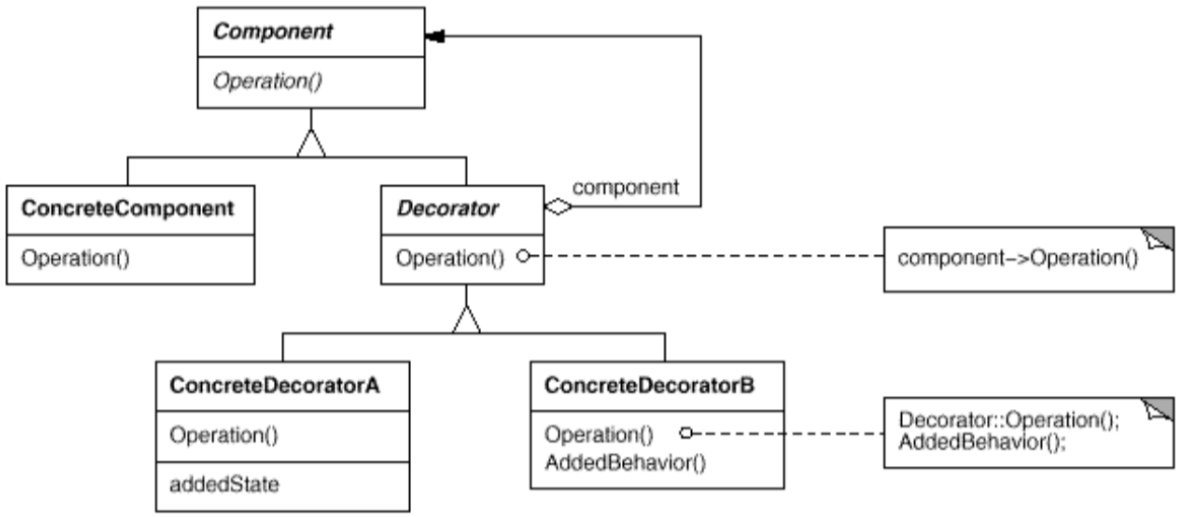
Sometimes you want to add responsibilities to individual objects, not to an entire class. For example borders on graphical components. A more flexible approach then inheritance is to enclose the component in another object that adds the border. This object is called a **decorator**. Consider a TextView object first being decorated by a ScrollDecorator and than a BorderDecorator, with TextView and Decorator being subclasses of VisualComponents and the scroll and border decorator subclasses of Decorator. All decorators then hold a VisualComponent object they paint over.

Applicability

Use the Decorator pattern when:

- To add responsibilities to individual objects dynamically and without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical, for example when many combinations of responsibilities are possible or you do not have access to the superclass.

Structure



Figuur 16: The structure of the Decorator pattern.

Participants

- **Component**: defines the interface for objects that can have responsibilities added to them dynamically. (VisualComponent)

- **ConcreteComponent**: defines an object to which additional responsibilities can be attached. (TextView)
- **Decorator**: maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**: adds responsibilities to the component. (BorderDecorator, ScrollDecorator)

Collaborations

Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

Advantages:

1. More flexibility than static inheritance.
2. Instead of trying to support all foreseeable features and their combinations in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Disadvantages:

- A decorator is not identical to its components, they have a different identity.
- A design that uses Decorator often results in a system composed of lots of little objects that all look alike. This makes learning and debugging the design difficult.

Implementation

Consider the following issues when implementing:

- A decorator object's interface must conform to the interface of the component it decorates.
- There is no need to define an abstract Decorator class when you are absolutely sure that never ever ever more than one responsibility will be added.
- Keep the Component class lightweight, do not let is store much data.
- Changing the skin of an object (Decorator Pattern) versus changing its guts (Strategy Pattern).

Sample code

Pages 180-182.

Known uses

Graphical widgets or I/O streams with different input formats as decorators.

Related patterns

A Decorator is different from an Adapter in that a decorator only changes an object's responsibilities, not its interface. An Adapter will give an object a completely new interface. A Decorator can be viewed as a degenerate Composite with only one component, however, a decorator adds additional responsibilities and is not meant for object aggregation. A Decorator lets you change the skin of an object, while a Strategy lets you change its guts. These are two alternative ways of changing an object.

7.5 Facade

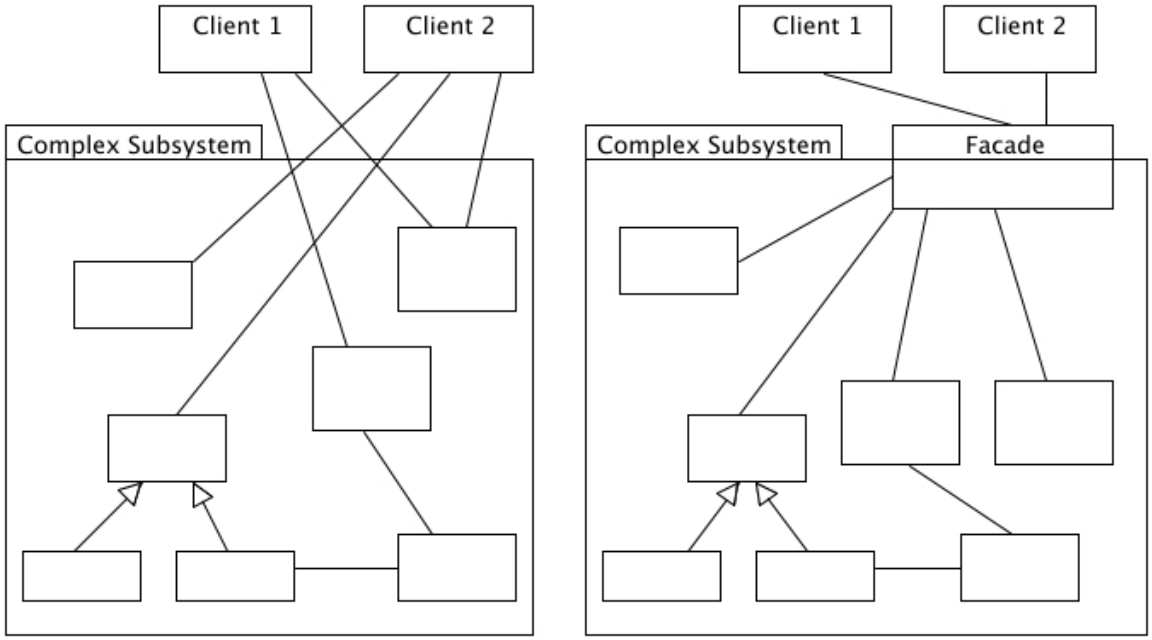
This is an object structural pattern. See p185 and on.

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

Structuring a system into subsystems helps reduce complexity. To minimize communication and dependencies with these subsystems, a **facade** object is introduced which probides a single simplified interface to the more general facilities of a subsystem.



Figuur 17: You want to go from the left to the right.

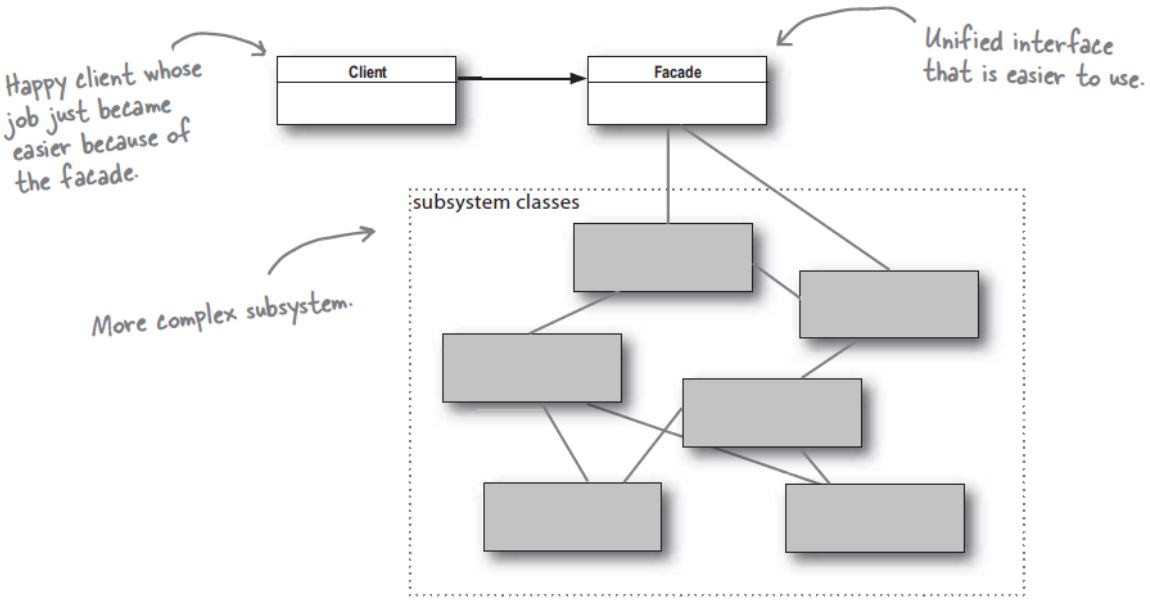
Consider an IDE with Scanner, Parser, ProgramNode, ByteCodeStream, ProgramNodeBuilder,... Some tasks may want to speak to them directly, but almost all other tasks will ask them to compile. So you can make a facade class Compiler.

Applicability

Use the Facade pattern when:

- You want to provide a simple interface to a complex subsystem. Because subsystems often evolve, this solution makes the subsystem more reusable.
- To decouple and lower dependencies.
- You want to layer your subsystem.

Structure



Figuur 18: The structure of the Facade pattern.

Participants

- **Facade**: knows which subsystem classes are responsible for a request and delegates client requests to the appropriate subsystem objects. (Compiler)
- **subsystem classes**: implements subsystem functionality and handles work assigned to it by the Facade object. It has no knowledge or references of the facade.

Collaborations

Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem objects, with the facade possibly doing some work to translate its interface into to subsystem interfaces.

Consequences

Advantages:

1. It shields clients from subsystem components, making the subsystem easier to use.
2. Low coupling.
3. It does not prevent applications from using subsystem classes if they need to.

Implementation

Consider the following issues when implementing:

1. The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Or by configuring a Facade with different subsystem objects. However, the client then cannot (easily) acces subsystem components anymore.
2. Public versus private subsystem classes.

Sample code

Pages 188-191.

Known uses

Compilers, operating systems, frameworks, browsers,...

Related patterns

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. It can also be used as an alternative to Facade for hiding platform-specific classes.
Mediator is similar to Facade in that it abstracts functionality of existing classes, however, Mediator's purpose is to abstract arbitrary communication between colleague objects. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly, while both these are not true for a Facade.

7.6 Flyweight

This is an object structural pattern. See p195 and on.

Intent

Use sharing to support large number of fine-grained objects efficiently.

Motivation

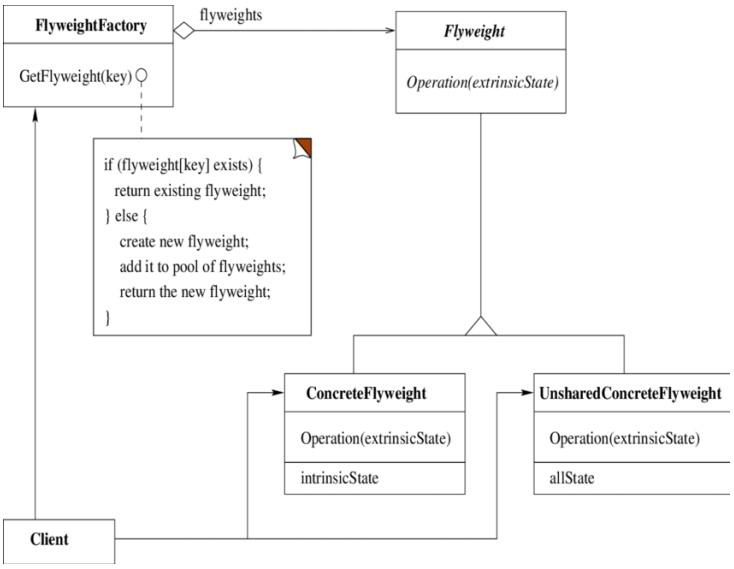
Some applications could benefit from sharing objects, because a naive implementation would be very expensive. Consider a Column or Row holding Characters with specific fonts,... A flyweight is a shared object that can be used in multiple contexts simultaneously. The Character could be implemented as such and the Rows only holding pointers to the flyweights. The key concept here is the distinction between **intrinsic** and **extrinsic** state. Intrinsic state is stored within the flyweight and consists of information that is independent of the flyweights context. Extrinsic state depends on and varies with the flyweight's context and therefore can not be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

Applicability

The flyweight pattern's effectiveness depends heavily on how and where it is used. Apply only when **ALL** the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made intrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application does not depend on object identity.

Structure



Figuur 19: The structure of the Flyweight pattern.

Participants

- **Flyweight**: declares an interface through which flyweights can receive and act on extrinsic state. (Glyph)
- **ConcreteFlyweight**: implements the Flyweight interface and adds storage for intrinsic state. Must be sharable. (Character)
- **UnsharedConcreteFlyweight**: not all Flyweight subclasses have to be shared. Unshared ones often have shared ones as children. (Row,Column)
- **FlyweightFactory**: creates and manages flyweight objects so they are shared properly. When a client requests a flyweight, this supplies an existing one or creates one.
- **Client**: maintains references to flyweight(s) and computes or stores the extrinsic state of flyweight(s).

Collaborations

Clients pass extrinsic state to the flyweight when they invoke its operations. Clients must obtain flyweights only from the FlyweightFactory and never instantiate them themselves.

Consequences

Advantages:

1. Less memory usage.

Disadvantages:

1. Extra run-time cost.

Implementation

Consider the following issues when implementing:

1. How much extrinsic state can be removed.
2. How to manage the shared objects. Often with a hashmap/dictionary.

Sample code

Pages 201-205.

Known uses

GUIs.

Related patterns

The Flyweight patter is often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes, making leaf nodes unable to store a pointer to their parent. The parent pointer is then passed to the flyweight as part of the extrinsic state. This will be a directed, acyclic graph with shared leaf nodes.
It is often best to implement State en Strategy as flyweights.

7.7 Proxy

This is an object structural pattern. See p207 and on.

Intent

Provide a surrogate (=substitute) or placeholder for another object to control access to it.

Also known as

Surrogate.

Motivation

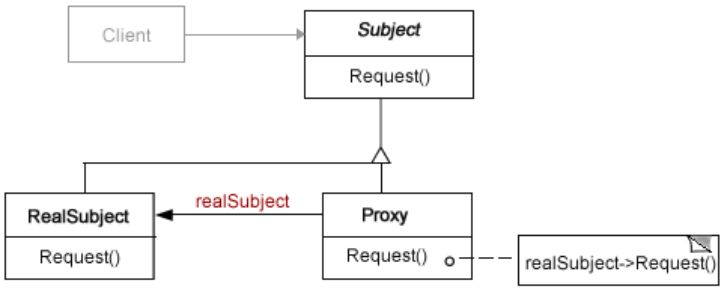
One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor (e.g. ms word) opening a file of multiple pages with a lot of images. Images can be expensive to create/show, but not to whole file will be on the screen at the same time. Mostly a couple of pages. So you could only render pictures when you need them, that is, when you scroll to their page. This is called creating *on demand*. To do this, we use another **proxy** object, that acts as a stand-in for the real image. It acts just like it en loads it when it is needed. It can use the filename to reference the image and stores some info it needs to placehold like the width and height, its **extend**.

Applicability

Proxy is applicable when there is a need for a more versatile or sophisticated reference to an object than a simple pointer:

1. **Remote proxy**: provides a local representative for an object in a different address space.
2. **Virtual proxy**: creates expensive objects on demand.
3. **Protection proxy**: controls access to the original object.
4. **Smart reference**: replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include:
    - counting the number of references to the real object so that it can be freed automatically when there are no more references (= smart pointer).
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



Figuur 20: The structure of the Proxy pattern.

Participants

- **Proxy**: maintains a reference that lets the proxy access the real subject. Provides an interface identical to Subject's so that a proxy can by substituted for the real subject. Controls access to the real subject and may be responsible for creating and deleting it.
  Other responsibilities depend on the kind of proxy:
  - **Remote proxies** are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - **Virtual proxies** may cache additional information about the real subject so that they can postpone accessing it.
  - **Protection proxies** check that the caller has the access permissions required to perform a request.
  - **Copy-on-write**: using a proxy to postpone a very demanding copying process. The subject must be reference counted. Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it and decrements the counter. If the counter is zero, the subjects get deleted.

  (ImageProxy)
- **Subject**: defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected. (Graphic)
- **RealSubject**: defines the real object that the proxy represents. (Image)

Collaborations

Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

Advantages:

1. The Proxy pattern introduces a level of indirection when accessing an object.
2. A remote proxy can hide the fact that an object resides in a different address space.
3. A virtual proxy can perform optimizations such as creating an object on demand.
4. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.
5. Copy-on-write can reduce the cost of copying heavyweight subjects significantly.

Implementation

The Proxy pattern can exploit the following language features:

1. Overloading the member access operator in C++. C++ supports overloading operator->, the member access operator. The proxy behaves just like a pointer.
2. Proxy doesn't always have to know the type of real subject. If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class.

Sample code

Pages 213-216.

Known uses

Kernels, security, graphical, heavy operations.

Related patterns

Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject, and refuse an operation if used for access control.
A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

7.8 Discussion of Structural Patterns

Adapter VS Bridge

Adapter focuses on resolving incompatibilities between two existing interfaces. It doesn't focus on how those interfaces are implemented, nor does it consider how they might evolve independently. An adapter becomes necessary when two incompatible classes work together, it is used after the design. A Bridge bridges an abstraction and its (potentially numerous) implementations. It provides a stable interface to clients even as it lets you vary the classes that implement it. It also accommodates new implementations as the system evolves. The user of a bridge understands upfront that an abstraction must have several implementations, and both may evolve independently. It is used before design

Adapter VS Facade

Facade defines a new interface, whereas an adapter reuses an old interface.

Composite VS Decorator VS Proxy

Decorator is designed to let you add responsibilities to objects without subclassing. Composite focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one. Its focus is not on embellishment but on representation. Proxy pattern is not concerned with attaching or detaching properties dynamically, and it's not designed for recursive composition. Its intent is to provide a stand-in for a subject when it's inconvenient or undesirable to access the subject directly.

8 Behavioral Patters

These patterns characterize the ways in which classes or objects interact and distribute responsibility. Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Class vs Object

Behavioral class patterns use inheritance to distribute behavior between classes. A Template Method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. An Interpreter represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.
Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. The Mediator provides the indirection needed for loose coupling. Chain of Responsibility lets you send requests to an object implicitly through a chain of candidate objects. Any candidate may fulfill the request depending on runtime conditions. You can select which candidates participate in the chain at run-time. The Observer (293) pattern defines and maintains a dependency between objects. Other behavioral object patterns are concerned with encapsulating behavior in an object and delegating requests to it. The Strategy (315) pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses. The Command (233) pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. The State (305) pattern encapsulates the states of an object so that the object can change its behavior when its state object changes. Visitor (331) encapsulates behavior that would otherwise be distributed across classes, and Iterator (257) abstracts the way you access and traverse objects in an aggregate.

8.1 Chain of Responsibility

This is an object behavioral pattern. See p223 and on.

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation

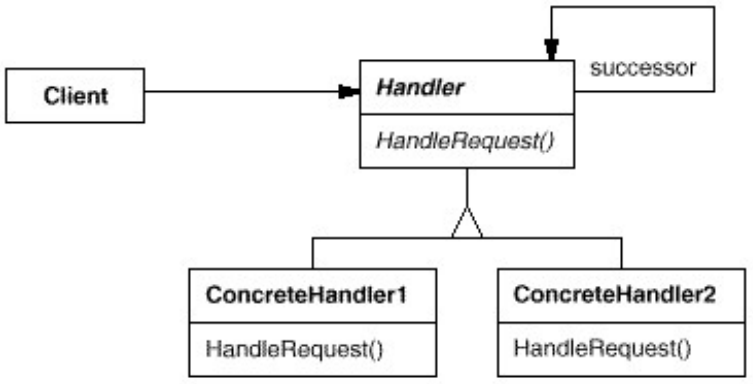
Consider a context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. The help that's provided depends on the part of the interface that's selected and its context. General help is shown when there is no specific help. You can not solve this according to generality because The object that ultimately provides the help isn't known explicitly to the object (e.g., the button) that initiates the help request. The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it. The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain, which does likewise. The object that made the request has no explicit knowledge of who will handle it, we say the request has an **implicit receiver**. The next object in the chain is called the **successor** of the current one. Subclasses can override this operation to provide help under the right circumstances, otherwise they can use the default implementation to forward the request.

Applicability

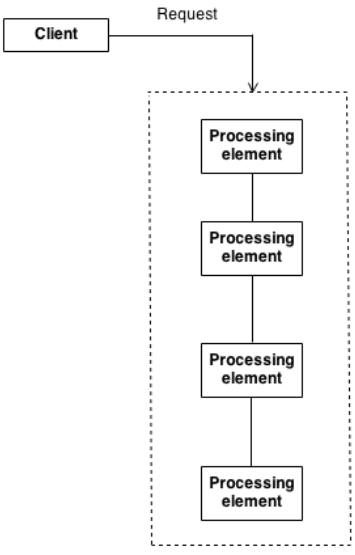
Use Chain of Responsibility when:

- More than one object may handle a request, and the handler isn't known a priori. The handler should be determined automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

Structure



(a) The structure of the composite pattern.



(b) A runtime example of objects from the pattern.

Participants

- **Handler**: defines an interface for handling requests an optionally implements the successor link. (HelpHandler)
- **ConcreteHandler**: handles requests it is responsible for an can access its successor. If the ConcreteHandler can handle the request, it does so. Otherwise it forwards the request to its successor. (PrintButton)
- **Client**: initiates the request to a ConcreteHandler object on the chain.

Collaborations

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

Consequences

Advantages:

1. Reduced coupling. The pattern frees an object from knowing which other object handles a request.
2. Added flexibility in assigning responsibilities to objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at runtime. You can combine this with subclassing to specialize handlers statically.

Disadvantages:

1. Since a request has no explicit receiver, there's no guarantee it'll be handled.

Implementation

Issues to consider when implementing:

1. There are two possible ways to implement the successor chain:
   (a) Define new links (usually in the Handler, but ConcreteHandlers could define them instead).
   (b) Use existing links. Using existing links works well when the links support the chain you need. But if the structure doesn't reflect the chain of responsibility your application requires, then you'll have to define redundant links.
2. If there are no preexisting references for defining a chain, then you'll have to introduce them yourself. In that case, the Handler not only defines the interface for the requests but usually maintains the successor as well. That lets the handler provide a default implementation of HandleRequest that forwards the request to the successor (if any). If a ConcreteHandler subclass isn't interested in the request, it doesn't have to override the forwarding operation, since its default implementation forwards unconditionally.
3. Representing requests. In the simplest form, the request is a hardcoded operation invocation. An alternative is to use a single handler function that takes a request code (e.g., an integer constant or a string) as parameter. The sender en receiver need to agree what means what. Flexible but requires switch cases. To address the parameter passing problem, we can use separate request objects that bundle request parameters. You can also subclass this.

Sample code

Pages 229-232.

Known uses

Event handlers, graphical updates.

Related patterns

Chain of Responsibility is often applied in conjunction with Composite (163). There, a component's parent can act as its successor.

8.2 Command

This is an object behavioral pattern. See p233 and on.

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Also known as

Action, Transaction.

Motivation

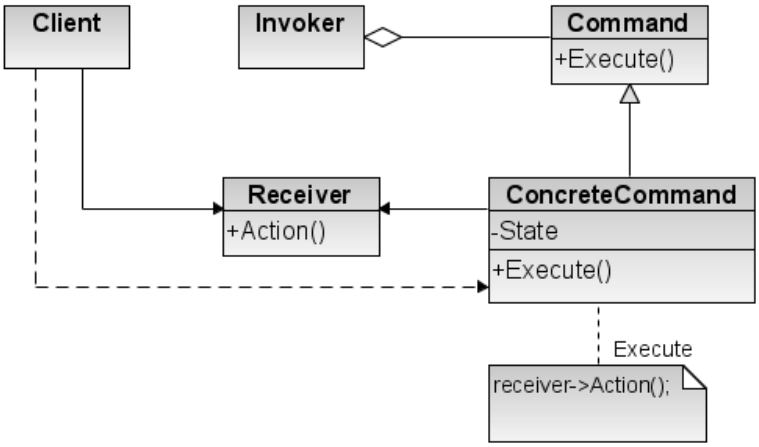
Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. For example, what a button from a toolkit should do cannot be hardcoded because they do different things. The Command pattern turns the request (what a button should do) itself into an object. The key to this pattern is an abstract Command class, which declares an interface for executing operations, which at least declares an abstract Execute operation. Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it. Very flexible. We can replace commands dynamically, which would be useful for implementing context-sensitive menus. We can also support command scripting by composing commands into larger ones.

Applicability

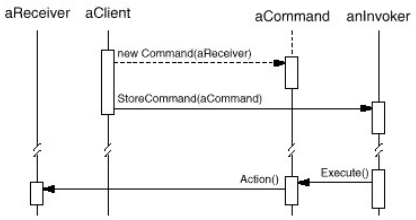
Use the Command pattern when you want to:

- Parameterize objects by an action to perform. In procdural languages this is done with a callback function. This is a function that's registered somewhere to be called at a later point. Commands are an objectoriented replacement for callbacks.
- Specify, queue, and execute requests at different times.
- Support undo, by declaring an unexexute() function. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.
- Support logging changes so that they can be reapplied in case of a system crash, by declaring load() and store() operations.
- Structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A **transaction** encapsulates a set of changes to data.

Structure



(a) The structure of the Command pattern.



(b) A runtime example of objects from the pattern.

Participants

- **Command**: declares an interface for executing an operation.
- **ConcreteCommand**: Defines a binding between a Receiver object and an action. It also implements Execute by invoking the corresponding operation(s) on Receiver. (PasteCommand, OpenCommand)
- **Client**: creates a ConcreteCommand object and sets its receiver. (Application)
- **Invoker**: asks the command to carry out the request. (MenuItem)
- **Receiver**: knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver. (Document, Application)

Collaborations

The client creates a ConcreteCommand object and specifies its receiver. An Invoker object stores the ConcreteCommand object. The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute. The ConcreteCommand object invokes operations on its receiver to carry out the request.

Consequences

Advantages:

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands can be manipulated and extended like any other object.
- You can assemble commands into a composite command.
- It's easy to add new Commands, because you don't have to change existing classes.

Implementation

Consider the following issues when implementing:

- How intelligent should a command be?
- Supporting undo and redo. The application needs a history list of commands that have been executed, where the maximum length of the list determines the number of undo/redo levels. The history list stores sequences of commands that have been executed. Traversing backward through the list and reverse-executing commands cancels their effect; traversing forward and executing commands re-executes them.
- Avoiding error accumulation in the undo process. It may be necessary therefore to store more information in the command to ensure that objects are restored to their original state. The Memento (283) pattern can be applied to give the command access to this information without exposing the internals of other objects.
- Using C++ templates. For commands that (1) aren't undoable and (2) don't require arguments, we can use C++ templates to avoid creating a Command subclass for every kind of action and receiver.

Sample code

Pages 239-242.

Known uses

Operating systems, functors.

Related patterns

A Composite (163) can be used to implement MacroCommands (a command consisting of commands). A Memento (283) can keep state the command requires to undo its effect. A command that must be copied before being placed on the history list acts as a Prototype (117).

8.3 Interpreter

This is a class behavioral pattern. See p243 and on.

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Motivation

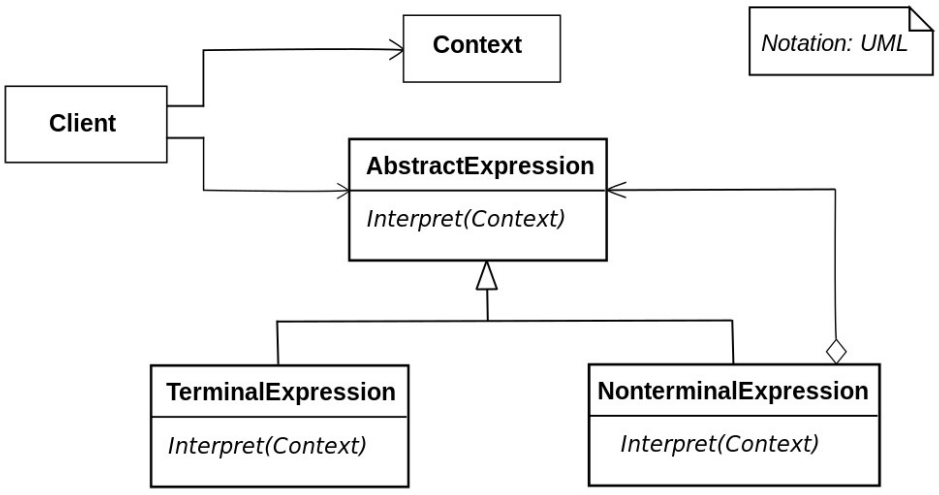
If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences.
Consider searching for strings to match a pattern. You could make a regular expressions for the set of strings to match. The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences. The Interpreter language here is a regular expression. We can create an interpreter for these regular expressions by defining the Interpret operation on each subclass of RegularExpression. Interpret takes as an argument the context in which to interpret the expression. The context contains the input string and information on how much of it has been matched so far. Each subclass of RegularExpression implements Interpret to match the next part of the input string based on the current context. You have literals, alternation (=or) and repetition.

Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. It works best when:

- The grammar is simple.
- Efficiency is not a critical concern.

Structure



Figuur 23: The structure of the Interpreter pattern.

Participants

- **AbstractExpression**: declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree. (RegularExpression)

- **TerminalExpression**: –implements an Interpret operation associated with terminal symbols in the grammar. An instance is required for every terminal symbol in a sentence. (LiteralExpression)
- **NonterminalExpression**: One such class is required for every rule $R ::= R_1, R_2, ..., R_n$ in the grammar. It maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n. It implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n. (Alternation-, Repitition- and SequenceExpression)
- **Context**: contains information that's global to the interpreter.
- **Client**: builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes. Invokes the Interpret operation on it.

Collaborations

The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation. Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion. The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

Advantages:

1. It's easy to change and extend the grammar, by changing or adding subclasses.
2. Classes (grammar) is easy to write.
3. The Interpreter pattern makes it easier to evaluate an expression in a new way.

Disadvantages:

- Complex grammars are hard to maintain.

Implementation

The Interpreter and Composite (163) patterns share many implementation issues. Consider these issues when implementing:

1. Creating the abstract syntax tree. The Interpreter pattern doesn't explain how to create an abstract syntax tree. In other words, it doesn't address parsing.
2. Defining the Interpret operation. You don't have to define the Interpret operation in the expression classes. If it's common to create a new interpreter, then it's better to use the Visitor (331) pattern to put Interpret in a separate "visitor" object.
3. Sharing terminal symbols with the Flyweight pattern.

Sample code

Pages 248-255.

Known uses

Widely used in compilers. Nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of the class hierarchy as defining a language.

Related patterns

Composite (163): The abstract syntax tree is an instance of the Composite pattern.
Flyweight (195) shows how to share terminal symbols within the abstract syntax tree.
Iterator (257): The interpreter can use an Iterator to traverse the structure.
Visitor (331) can be used to maintain the behavior in each node in the abstract syntax tree in one class.

8.4 Iterator

This is an object behavioral pattern. See p257 and on.

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also known as

Cursor.

Motivation

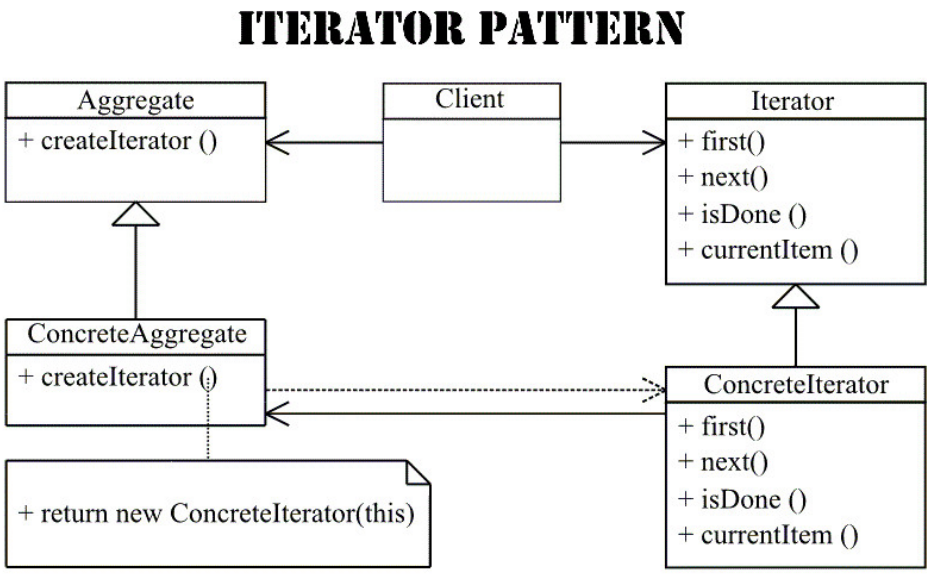
You might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an **iterator object**. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

Applicability

Use the Iterator pattern to:

- access an aggregate object's contents without exposing its internal representation.
- support multiple traversals of aggregate objects.
- provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure



Figuur 24: The structure of the Iterator pattern.

Participants

- **Iterator**: defines an interface for accessing and traversing elements.
- **ConcreteIterator**: implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.
- **Aggregate**: defines an interface for creating an Iterator object.
- **ConcreteAggregate**: implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

Advantages:

1. It supports variations in the traversal of an aggregate. Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.
2. Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation

Iterator has many implementation variants and alternatives. Some important ones follow. The trade-offs often depend on the control structures your language provides.

1. A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator. When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**.2 Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. In contrast, the client hands an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate. External iterators are more flexible than internal iterators. Internal iterators are especially weak in a language like C++ that does not provide anonymous functions. But on the other hand, internal iterators are easier to use, because they define the iteration logic for you.
2. Who defines the traversal algorithm? The iterator is not the only place where the traversal algorithm can be defined. The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a **cursor**, since it merely points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor. If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates. On the other hand, the traversal algorithm might need to access the private variables of the aggregate. If so, putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.
3. It can be dangerous to modify an aggregate while you're traversing it. If elements are added or deleted from the aggregate, you might end up accessing an element twice or missing it completely. A **robust** iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate. Most rely on registering the iterator with the aggregate. On insertion or removal, the aggregate either adjusts the internal state of iterators it has produced, or it maintains information internally to ensure proper traversal.

4. Additional Iterator operations. E.G. previous() or skipTo().

5. Using polymorphic iterators in C++. Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack. Polymorphic iterators have another drawback: the client is responsible for deleting them. This is error-prone. The Proxy (207) pattern provides a remedy. We can use a stack-allocated proxy as a stand-in for the real iterator.

6. Iterators may have privileged access. An iterator can be viewed as an extension of the aggregate that created it. However, such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend. To avoid this problem, the Iterator class can include protected operations for accessing important but publicly unavailable members of the aggregate. Iterator subclasses (and only Iterator subclasses) may use these protected operations to gain privileged access to the aggregate.

7. Iterators for composites. Externaliterators can be difficult to implement over recursive aggregate structures like those in the Composite (163) pattern, because a position in the structure may span many levels of nested aggregates. Therefore an externaliterator has to store a path through the Composite to keep track of the current object. Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack. If the nodes in a Composite have an interface for moving from a node to its siblings, parents, and children, then a cursorbased iterator may offer a better alternative. The cursor only needs to keep track of the current node; it can rely on the node interface to traverse the Composite.

8. Null iterators. A NullIterator is a degenerate iterator that's helpful for handling boundary conditions. By definition, a NullIterator is always done with traversal; that is, its IsDone operation always evaluates to true. NullIterator can make traversing treestructured aggregates (like Composites) easier. At each point in the traversal, we ask the current element for an iterator for its children. Aggregate elements return a concrete iterator as usual. But leaf elements return an instance of NullIterator. That lets us implement traversal over the entire structure in a uniform way.

Sample code

Pages 263-270.

Known uses

Common in OOP.

Related patterns

Composite (163): Iterators are often applied to recursive structures such as Composites.
Factory Method (107): Polymorphic iterators rely on factory methods to instantiate the appropriate Iterator subclass.
Memento (283) is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.

8.5 Mediator

This is an object behavioral pattern. See p273 and on.

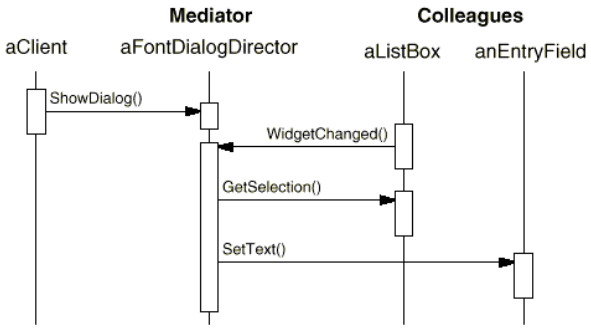
Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Motivation

Consider an object structure with many connections between objects. Consider the implementation of dialog boxes in a graphical user interface. A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields. Often there are dependencies between the widgets in the dialog. For example, a button gets disabled when a certain entry field is empty. Different dialog boxes will have different dependencies between widgets. You can avoid these problems by encapsulating collective behavior in a separate **mediator object**. A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections. FontDialogDirector can be the mediator between the widgets in a dialog box. A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction.

Figuur 25: The interaction of the example.

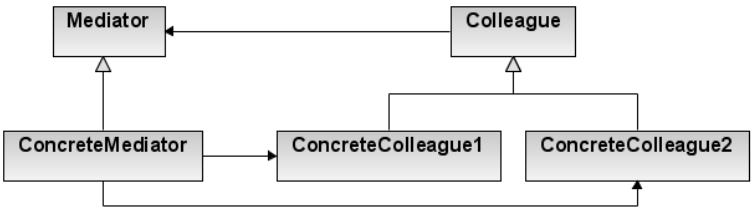


Applicability

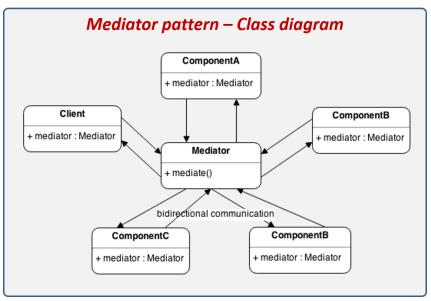
Use when:

- A set of objects communicate in welldefined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure



(a) The structure of the Mediator pattern.



(b) A class diagram example of the pattern.

Participants

- **Mediator**: defines an interface for communicating with Colleague objects. (DialogDirector)
- **ConcreteMediator**: Implements cooperative behavior by coordinating Colleague objects and knows and maintains its colleagues. (FontDiaglogDirector)
- **Colleague classes**: each Colleague class knows its Mediator object and communicates with its mediator whenever it would have otherwise communicated with another colleague. (ListBox, EntryField)

Collaborations

Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences

Advantages:

1. A mediator promotes loose coupling between colleagues. You can vary and reuse Colleague and Mediator classes independently.
2. It simplifies object protocols. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues, which are easier to work with.
3. It abstracts how objects cooperate. Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior. That can help clarify how objects interact in a system.

Disadvantages:

1. It limits subclassing. A mediator localizes behavior that otherwise would be distributed among several objects. Changing this behavior requires subclassing Mediator only; Colleague classes can be reused as is.
2. It centralizes control. This can make the mediator itself a monolith that's hard to maintain.

Implementation

Consider the following issues:

1. *This is my note: How do objects reference each other? By string IDs such as in HTML or something? And this only used in mediator as a dict? Or they are variables and know which they have to work with?*
2. Omitting the abstract Mediator class. There's no need to define an abstract Mediator class when colleagues work with only one mediator.
3. Colleague-Mediator communication. Colleagues have to communicate with their mediator when an event of interest occurs. One approach is to implement the Mediator as an Observer using the Observer (293) pattern. Colleague classes act as Subjects, sending notifications to the mediator whenever they change state. The mediator responds by propagating the effects of the change to other colleagues. Another approach defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication.

Sample code

Pages 278-281.

Known uses

Update managers, GUIs

Related patterns

Facade (185) differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.
Colleagues can communicate with the mediator using the Observer (293) pattern.

8.6 Memento

This is an object behavioral pattern. See p283 and on.

Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
My note: this seams to be great to work with databases or to save/load object to hdd!

Also known as

Token.

Motivation

Sometimes it's necessary to record the internal state of an object. But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally. Exposing this state would violate encapsulation.
Consider for example a graphical editor that supports connectivity between objects. A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them. The editor ensures that the line stretches to maintain the connection. We can encapsulate this functionality in a **ConstraintSolver** object. ConstraintSolver records connections as they are made and generates mathematical equations that describe them. It solves these equations whenever the user makes a connection or otherwise modifies the diagram. Supporting undo in this application isn't as easy as it may seem. A **memento** is an object that stores a snapshot of the internal state of another object. This other object is the **originator**. The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. The originator initializes the memento with information that characterizes its current state. Only the originator can store and retrieve information from the memento. The ConstraintSolver can act as an originator.
The sequence is as follows when implementing an undo operation this way:

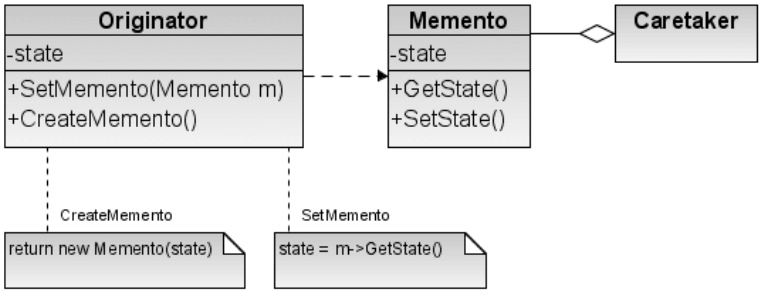
1. The editor requests a memento from the ConstraintSolver as a side-effect of the move operation.
2. The ConstraintSolver creates and returns a memento, an instance of a class SolverState in this case. A SolverState memento contains data structures that describe the current state of the ConstraintSolver's internal equations and variables.
3. Later when the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver.
4. Based on the information in the SolverState, the ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state.

Applicability

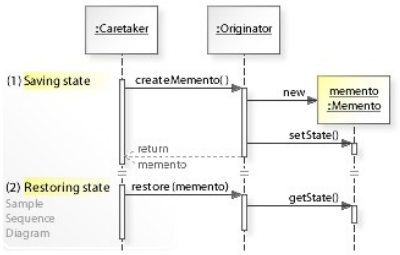
Use when **BOTH** are true:

1. a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.
2. a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure



(a) The structure of the Memento pattern.



(b) Interaction diagram of the pattern.

Participants

- **Memento**: stores internal state of the Originator object and protects against access by objects other than the originator. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state. (SolverState)
- **Originator**: creates a memento containing a snapshot of its current internal state and uses the memento to restore its internal state. Originator sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. (ConstraintSolver)
- **Caretaker**: is responsible for the memento's safekeeping. It never operates on or examines the contents of a memento. Caretaker sees a narrow interface to the Memento—it can only pass the memento to other objects. (undo mechanism)

Collaborations

A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator. Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.
Mementos are passive. Only the originator that created a memento will assign or retrieve its state. See figure 27b

Consequences

Advantages:

1. Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator. The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.
2. It simplifies Originator.

Disadvantages:

1. Using mementos might be expensive. Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.
2. It may be difficult in some languages to ensure that only the originator can access the memento's state.
3. Hidden costs in caring for mementos. A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento.

Implementation

Consider the following issues when implementing:

1. Language support. Mementos have two interfaces: a wide one for originators and a narrow one for other objects. Ideally the implementation language will support two levels of static protection. C++ lets you do this by making the Originator a friend of Memento and making Memento's wide interface private. Only the narrow interface should be declared public.

2. Storing incremental changes. When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just the incremental change to the originator's internal state. For example, undoable commands in a history list can use mementos to ensure that commands are restored to their exact state when they're undone. That means mementos can store just the incremental change that a command makes rather than the full state of every object they affect.

Sample code

Pages 288-289.

Known uses

Solvers.

Related patterns

Command (233): Commands can use mementos to maintain state for undoable operations.
Iterator (257): Mementos can be used for iteration as described earlier.

8.7 Observer

This is an object behavioral pattern. See p293 and on.

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. *wow!*

Also known as

Dependents, Publish-Subscribe.

Motivation

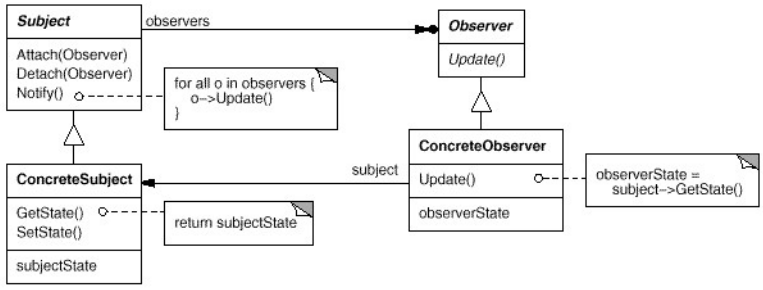
Often you want to maintain consistency between related objects, and you want to do it while maintaining low coupling. Consider a GUI toolkit that separates the representation from the application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa. This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. The key objects in this pattern are subject and observer. A **subject** may have any number of dependent observers. All **observers** are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Applicability

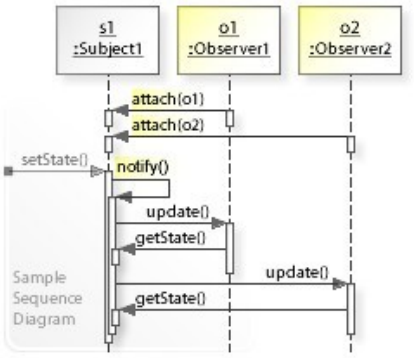
Use when:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure



(a) The structure of the Observer pattern.



(b) Interaction diagram of the pattern.

Participants

- **Subject**: provides an interface for attaching and detaching Observer objects, and knows its observers. Any number of Observer objects may observe a subject.
- **Observer**: defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**: stores state of interest to ConcreteObserver objects and sends a notification to its observers when its state changes.
- **ConcreteObserver**: maintains a reference to a ConcreteSubject object and stores state that should stay consistent with the subject's. It implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject. See figure 28b. Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject. Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely.

Consequences

Advantages:

1. The Observer pattern lets you vary subjects and observers independently.
2. You can reuse subjects without reusing their observers, and vice versa.
3. It lets you add observers without modifying the subject or other observers.
4. Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. Thus reduced coupling.
5. Support for broadcasting.

Disadvantages:

1. Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

Implementation

Consider the following issues:

1. Mapping subjects to their observers. By stored references, hash-map,...
2. Observing more than one subject. It might make sense in some situations for an observer to depend on more than one subject. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer know which subject to examine.
3. Who triggers the update? Have state-setting operations on Subject call Notify after they change the subject's state. Or make clients responsible for calling Notify at the right time.
4. Dangling references to deleted subjects. Deleting a subject should not produce dangling references in its observers.
5. Making sure Subject state is self-consistent before notification. You can avoid this pitfall by sending notifications from template methods (Template Method (325)) in abstract Subject classes. Define a primitive operation for subclasses to override, and make Notify the last operation in the template method, which will ensure that the object is selfconsistent when subclasses override Subject operations. By the way, it's always a good idea to document which Subject operations trigger notifications.
6. Avoiding observerspecific update protocols: the push and pull models. Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely. At one extreme, which we call the **push** model, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the **pull** model; the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter. The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.
7. Specifying modifications of interest explicitly.
8. Encapsulating complex update semantics. When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a **Change-Manager**. Its purpose is to minimize the work required to make observers reflect a change in their subject. The change manager should do 3 things: 1) It maps a subject to its observers and provides an interface to maintain this mapping. 2) It defines a particular update strategy. 3) It updates all dependent observers at the request of a subject. ChangeManager is an instance of the Mediator (273) pattern.
9. Combining the Subject and Observer classes.

Sample code

Pages 300-303.

Known uses

GUIs.

Related patterns

Mediator (273): By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

8.8 State

This is an object behavioral pattern. See p305 and on.

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Also known as

Objects for states.

Motivation

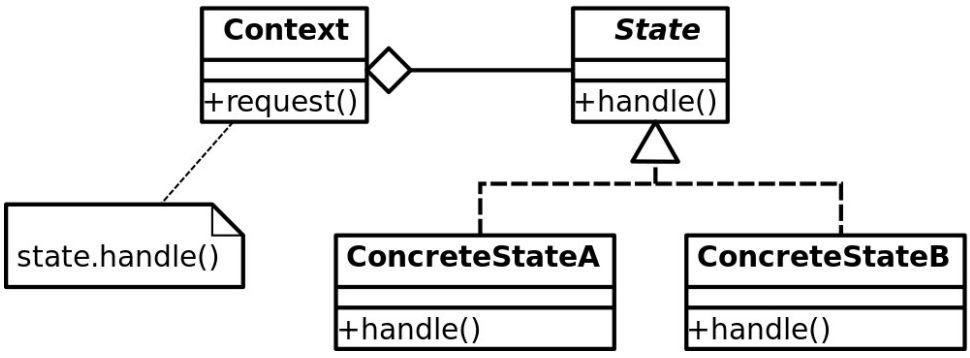
Consider a class TCPConnection that represents a network connection. A TCP-Connection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state. The key idea in this pattern is to introduce an abstract class called TCPState to represent the states of the network connection. The TCPState class declares an interface common to all classes that represent different operational states. Subclasses of TCPState implement statespecific behavior. The class TCPConnection maintains a state object (an instance of a subclass of TCPState) that represents the current state of the TCP connection. The class TCP Connection delegates all statespecific requests to this state object. TCPConnection uses its TCPState subclass instance to perform operations particular to the state of the connection. Whenever the connection changes state, the TCPConnection object changes the state object it uses.

Applicability

Use when:

- An object's behavior depends on its state, and it must change its behavior at runtime depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.

Structure



Figuur 29: The structure of the State pattern.

Participants

- **Context** defines the interface of interest to clients and maintains an instance of a ConcreteState subclass that defines the current state. (TCPConnection)
- **State** defines an interface for encapsulating the behavior associated with a particular state of the Context. (TCPState)

- **ConcreteState subclasses** each subclass implements a behavior associated with a state of the Context. (TCPEstablish, TCPListen, TCPClosed)

Collaborations

Context delegates state-specific requests to the current ConcreteState object. A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary. Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

Advantages:

1. It localizes state-specific behavior and partitions behavior for different states.
2. Easy to add new states.
3. It makes state transitions explicit.
4. State objects can protect the Context from inconsistent internal states.
5. State objects can be shared. When states are shared in this way, they are essentially flyweights (see Flyweight (195)) with no intrinsic state, only behavior.

Implementation

Consider the following issues:

1. Who defines the state transitions? The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context. It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make the transition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
2. A table-based alternative. He uses tables to map inputs to state transitions. For each state, a table maps every possible input to a succeeding state. In effect, this approach converts conditional code (and virtual functions, in the case of the State pattern) into a table look-up.
3. Creating and destroying State objects. A common implementation tradeoff worth considering is whether (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them ahead of time and never destroying them.
4. Using dynamic inheritance. Changing the behavior for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most object-oriented programming languages.

Sample code

Pages 309-312.

Known uses

Network protocols.

Related patterns

The Flyweight (195) pattern explains when and how State objects can be shared.

8.9 Strategy

This is an object behavioral pattern. See p315 and on.

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Also known as

Policy.

Motivation

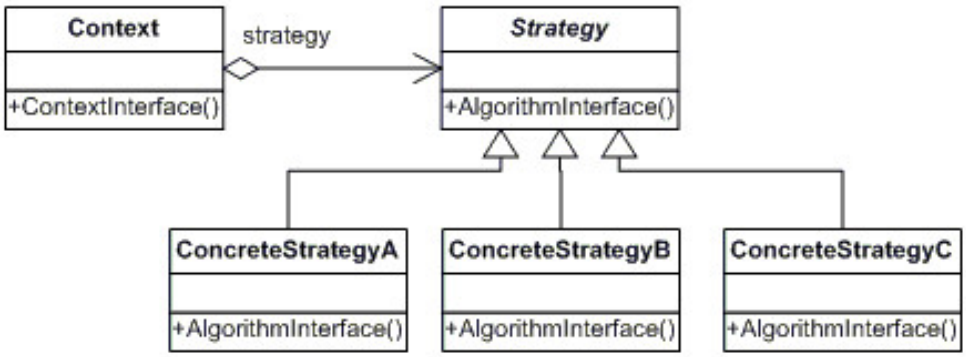
Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons: clients get more complex and bloated, different kinds algorithms will be used and difficult to add and change algorithms. We can avoid these problems by defining classes that encapsulate different line-breaking algorithms. An algorithm that's encapsulated in this way is called a **strategy**. Linebreaking strategies are implemented separately by subclasses of the abstract Compositor class. A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.

Applicability

Use when:

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Figuur 30: The structure of the Strategy pattern.

Participants

- **Strategy**: declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy. (Compositor)
- **ConcreteStrategy**: implements the algorithm using the Strategy interface. (SimpleCompositor, TeXCompositor, ArrayCompositor)
- **Context**: is configured with a ConcreteStrategy object and maintains a reference to a Strategy object. May define an interface that lets Strategy access its data. (Composition)

Collaborations

Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required. A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

Advantages:

1. Families of related algorithms. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse.
2. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. Strategies eliminate conditional statements. Code containing many conditional statements often indicates the need to apply the Strategy pattern.
4. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space tradeoffs.

Disadvantages:

1. Clients must be aware of different Strategies.
2. Communication overhead between Strategy and Context.
3. Increased number of objects.

Implementation

Consider the following implementation issues:

1. Defining the Strategy and Context interfaces. The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa. One approach is to have Context pass data in parameters to Strategy operations. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need. Another technique has a context pass itself as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. Either way, the strategy can request exactly what it needs. But now Context must define a more elaborate interface to its data, which couples Strategy and Context more closely. The needs of the particular algorithm and its data requirements will determine the best technique.

2. Strategies as template parameters. In C++ templates can be used to configure a class with a strategy. This technique is only applicable if (1) the Strategy can be selected at compiletime, and (2) it does not have to be changed at run time. In this case, the class to be configured (e.g., Context) is defined as a template class that has a Strategy class as a parameter. With templates, there's no need to define an abstract class that defines the interface to the Strategy. Using Strategy as a template parameter also lets you bind a Strategy to its Context statically, which can increase efficiency.

3. Making Strategy objects optional. The Context class may be simplified if it's meaningful not to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If there is one, then Context uses it normally. If there isn't a strategy, then Context carries out default behavior. The benefit of this approach is that clients don't have to deal with Strategy objects at all unless they don't like the default behavior.

Sample code

Pages 320-322.

Known uses

Financial instruments, memory allocation, GUIs.

Related patterns

Flyweight (195): Strategy objects often make good flyweights.

8.10 Template Method

This is a class behavioral pattern. See p325 and on.

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Motivation

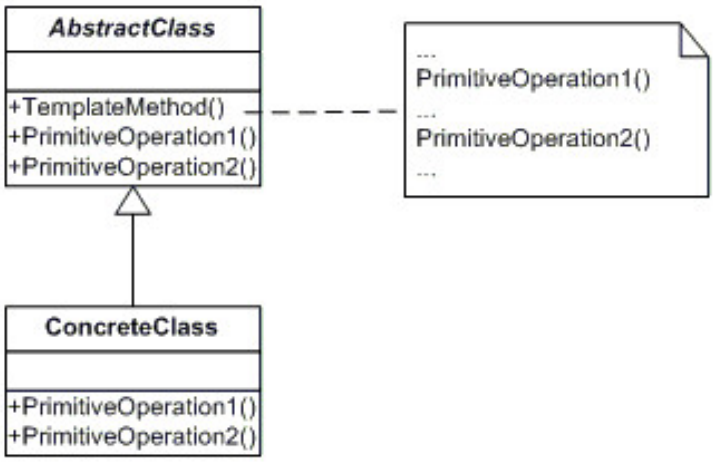
Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file. Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines Draw Application and DrawDocument subclasses; a spreadsheet application defines Spreadsheet Application and SpreadsheetDocument subclasses. The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation. OpenDocument defines each step for opening a document. We call OpenDocument a template method. A **template method** defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. Application subclasses define the steps of the algorithm that check if the document can be opened (CanOpenDocument) and that create the Document (DoCreateDocument). Document classes define the step that reads the document (DoRead). The template method also defines an operation that lets Application subclasses know when the document is about to be opened (AboutToOpenDocument), in case they care. By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but it lets Application and Document subclasses vary those steps to suit their needs.

Applicability

Use when:

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

Structure



Figuur 31: The structure of the Template Method pattern.

Participants

- **AbstractClass** Defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm. Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects. (Application)
- **ConcreteClass** Implements the primitive operations to carry out subclass-specific steps of the algorithm. (MyApplication)
- Types of operations:
  - Concrete operations (either on the ConcreteClass or on client classes).
  - Concrete AbstractClass operations (i.e., operations that are generally useful to subclasses).
  - Primitive operations (i.e., abstract operations).
  - Factory methods (see Factory Method (107)).
  - **Hook operations**, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.

  It's important for template methods to specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden). A subclass can extend a parent class operation's behavior by overriding the operation and calling the parent operation explicitly. Unfortunately, it's easy to forget to call the inherited operation. We can transform such an operation into a template method to give the parent control over how subclasses extend it. The idea is to call a hook operation from a template method in the parent class. Then subclasses can then override this hook operation. See page 328.

Collaborations

ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Advantages:

1. Fundamental technique for code reuse.

Implementation

1. Using C++ access control. In C++, the primitive operations that a template method calls can be declared protected members. This ensures that they are only called by the template method. Primitive operations that must be overridden are declared pure virtual. The template method itself should not be overridden; therefore you can make the template method a nonvirtual member function.

2. Minimizing primitive operations. An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.

3. Naming conventions. You can identify the operations that should be overridden by adding a prefix to their names.

Sample code

Pages 329.

Known uses

Class libraries. So fundamental they are found everywhere.

Related patterns

Factory Methods (107) are often called by template methods.
Strategy (315): Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

8.11 Visitor

This is an object behavioral pattern. See p331 and on.

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

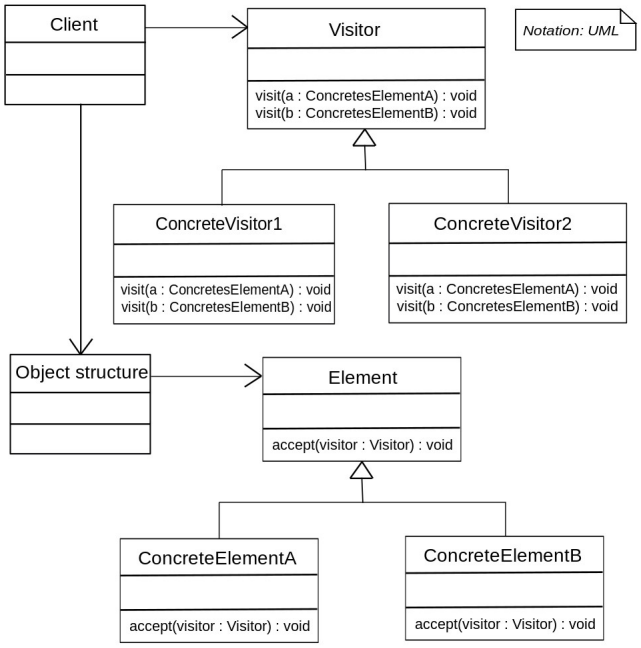
Consider a compiler that represents programs as abstract syntax trees. It might define operations for typechecking, code optimization, flow analysis,... Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions. The problem here is that distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It would be better if each new operation could be added separately, and the node classes were independent of the operations that apply to them. We can have both by packaging related operations from each class in a separate object, called a visitor, and passing it to elements of the abstract syntax tree as it's traversed. When an element "accepts" the visitor, it sends a request to the visitor that encodes the element's class. It also includes the element as an argument. The visitor will then execute the operation for that element. To make visitors work for more than just typechecking, we need an abstract parent class NodeVisitor for all visitors of an abstract syntax tree. NodeVisitor must declare an operation for each node class. With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. We can add new functionality simply by defining new NodeVisitor subclasses.

Applicability

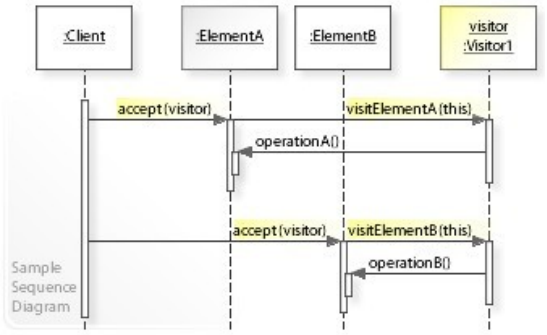
Use when:

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure.

Structure



(a) The structure of the Visitor pattern.



(b) Interaction diagram of the pattern.

Participants

- **Visitor**: declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. (NodeVisitor)
- **ConcreteVisitor**: implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure. (TypeCheckingVisitor)
- **Element**: defines an Accept operation that takes a visitor as an argument. (Node)
- **ConcreteElement**: implements an Accept operation that takes a visitor as an argument. (AssignmentNode, VariableRefNode)
- **ObjectStructure**: can enumerate its elements. May provide a high-level interface to allow the visitor to visit its elements. May either be a composite (see Composite (163)) or a collection such as a list or a set. (Program)

Collaborations

A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor. When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary. See figure 32b.

Consequences

Advantages:

1. Visitor makes adding new operations easy. You can define a new operation over an object structure simply by adding a new visitor.

2. A visitor gathers related operations and separates unrelated ones. Unrelated sets of behavior are partitioned in their own visitor subclasses. Any algorithm-specific data structures can be hidden in the visitor.

3. It can visit objects that don't have a common parent class.

4. Visitors can accumulate state as they visit each element in the object structure.

Disadvantages:

1. Adding new ConcreteElement classes is hard. Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.

2. Breaking encapsulation. Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.

Implementation

Consider the following issues:

1. See page 337, I did not understand the first thing. The only thing that I see they could make a point about is: the operation that ends up getting called depends on both the class of the element and the class of the visitor.

2. Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called **double-dispatch**. In **single-dispatch** languages (like C++), two criteria determine which operation will fulfill a request: the name of the request and the type of receiver. "Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of two receivers. Accept is a doubledispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Doubledispatching lets visitors request different operations on each class of element. You can consolidate the operations in a Visitor and use Accept to do the binding at runtime. Extending the Element interface amounts to defining one new Visitor subclass rather than many new Element subclasses.

3. Who is responsible for traversing the object structure? We can put responsibility for traversal in any of three places: in the object structure, in the visitor, or in a separate iterator object. You could even put the traversal algorithm in the visitor, although you'll end up duplicating the traversal code in each ConcreteVisitor for each aggregate ConcreteElement. The main reason to put the traversal strategy in the visitor is to implement a particularly complex traversal, one that depends on the results of the operations on the object structure.

Sample code

Pages 339-344.

Known uses

Compilers, graphics, toolkits.

Related patterns

Composite (163): Visitors can be used to apply an operation over an object structure defined by the Composite pattern.
Interpreter (243): Visitor may be applied to do the interpretation.

8.12 Discussion of Behavioral Patterns

Encapsulating Variation

When an aspect of a program changes frequently, these patterns define an object that encapsulates that aspect. Then other parts of the program can collaborate with the object whenever they depend on that aspect. The patterns usually define an abstract class that describes the encapsulating object, and the pattern derives its name from that object.

- a Strategy object encapsulates an algorithm (Strategy (315)).
- a State object encapsulates a statedependent behavior (State (305)).
- a Mediator object encapsulates the protocol between objects (Mediator (273)).
- an Iterator object encapsulates the way you access and traverse the components of an aggregate object (Iterator (257)).

These patterns describe aspects of a program that are likely to change. Most patterns have two kinds of objects: the new object(s) that encapsulate the aspect, and the existing object(s) that use the new ones. Usually the functionality of new objects would be an integral part of the existing objects were it not for the pattern.

Objects as Arguments

Several patterns introduce an object that's always used as an argument, i.e. Visitor (331). Other patterns define objects that act as magic tokens to be passed around and invoked at a later time. Both Command (233) and Memento (283) fall into this category.

Should Communication be Encapsulated or Distributed

Mediator (273) and Observer (293) are competing patterns. The difference between them is that Observer distributes communication by introducing Observer and Subject objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators. On the other hand, it's easier to understand the flow of communication in Mediator than in Observer.

Decoupling Senders and Receivers

When collaborating objects refer to each other directly, they become dependent on each other, and that can have an adverse impact on the layering and reusability of a system. Command, Observer, Mediator, and Chain of Responsibility address how you can decouple senders and receivers, but with different trade-offs.
Command keeps the sender decoupled from the receivers, making senders easy to reuse. Moreover, you can reuse the Command object to parameterize a receiver with different senders.
Observer defines a looser sender-receiver binding than Command, since a subject may have multiple observers, and their number can vary at run-time. Therefore the Observer pattern is best for decoupling objects when there are data dependencies between them.
A Mediator object routes requests between Colleague objects and centralizes their communication. Consequently, colleagues can only talk to each other through the Mediator interface. Because this interface is fixed, the Mediator might have to implement its own dispatching scheme for added flexibility. Requests can be encoded and arguments packed in such a way that colleagues can request an open-ended set of operations. The Mediator pattern can reduce subclassing in a system, because it centralizes communication behaviour in one class instead of distributing it among subclasses. However, ad hoc dispatching schemes often decrease type safety.
Chain of Responsibility may also require a custom dispatching scheme. It has the same type-safety drawbacks as Mediator. Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request. Moreover, the pattern offers added flexibility in that the chain can be changed or extended easily.

Summary

With few exceptions, behavioral design patterns complement and reinforce each other. A class in a chain of responsibility, for example, will probably include at least one application of Template Method (325). The template method can use primitive operations to determine whether the object should handle the request and to choose the object to forward to. The chain can use the Command pattern to represent requests as objects. Interpreter (243) can use the State pattern to define parsing contexts. An iterator can traverse an aggregate, and a visitor can apply an operation to each element in the aggregate. Behavioral patterns work well with other patterns, too. For example, a system that uses the Composite (163) pattern might use a visitor to perform operations on components of the composition. It could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator (175) to override these properties on parts of the composition. It could use the Observer pattern to tie one object structure to another and the State pattern to let a component change its behavior as its state changes. The composition itself might be created using the approach in Builder (97), and it might be treated as a Prototype (117) by some other part of the system. Welldesigned objectoriented systems are just like this—they have multiple patterns embedded in them—but not because their designers necessarily thought in these terms. Composition at the pattern level rather than the class or object levels lets us achieve the same synergy with greater ease.

Conclusion

So, was this summary worth your time?