WHAT FITS WHERE? OVERVIEW #1

GOF CLASSIFICATION OF DESIGN PATTERNS

BEHAVIORAL PATTERNS

BRIEF DEFINITION

The focus of Behavioral Patterns focus on three things: **communication**, **separation** of **responsibilities** and **cooperation** between objects.

OVERVIEW OF THE BEHAVIORAL PATTERNS

Chain of responsibility – Passes request through until object is found to execute it.
Command – Encapsulates actions in separate Command objects, each with a specific task and responsibility. Can implement "roll back" or "undo"
Delegate* – Delegates actions to other objects. Can change the objects to delegate to, based on parameters given to the Delegate
Interpreter – Interprets a language and translates that to a relational object structure.
Iterator – Describes ways to iterate through lists.
Manager* – Uses elements from Façade and Mediator Patterns to centralize communication and decision making in the project.
Mediator – Centralizes and mediates communication between objects in your project.
Memento – Describes ways to make snapshots of objects, to be stored somewhere else or to roll back actions and commands.
Observer – Decouples hard dependencies between objects, allows you to observe objects that perform actions and to receive callbacks.
Operator* – Acts as a gatekeeper to subsystems, can make decisions like the Manager and combines elements from Façade and Mediator pattern
State – Is used in situations where a method call might lead to a change in the state of an object, leading to a different way to execute a process.
Strategy – Describes a pattern that allows you to change parts of the implementation of processes used to execute actions in your class.
Template Method – Is a class-based pattern describing how you can use place-holder methods as a template to implement the concrete actions in class-specific code.
Visitor – Is a pattern in which you send an object (the Visitor) into a object structure, to perform actions or collect data.

CREATIONAL PATTERNS

BRIEF DEFINITION

Creational Patterns are used to **create objects**.

OVERVIEW OF THE CREATIONAL PATTERNS

Builder – Creates (complex) object structures using two parts: the Director and the Concrete builder. The Director describes the prcess. The Concrete Builder chooses the materials and creates the objects
Prototype -
Simple Factory* - Is the most used variant on the Factory Pattern. Creates objects on demand, abstracting the creation process.
Singleton – Takes control of and care of the creation of the Singleton object. Creates and provides one single object for all:to retrieve and use.
Multiton* – Takes care and control of creation of the Multiton object. Provides one single object per key provided by the requester.
Factory Object Map* - Creates and maps objects in a multi-tier map, using the object ID and class reference as two main keys for storage and retrieval. Centralizes access to objects that represent an entity.
Factory Object Pool* - Provides recyclable objects from an object pool and creates these objects when the pool is empty. Centralizes re-use of objects.
Extended Identity Map* - Creates and stores objects using their identity. Centralizes creation and access to objects that represent an entity.
Abstract Factory – Describes the abstract classes and cooperation to create interchangeable factories.
Factory Method – Describes the structure to create classes with factory methods, where these (your) classes can be interchangeable. Is also the basis for Abstract Factory.

STRUCTURAL PATTERNS

BRIEF DEFINITION

Structural Patterns focus on three things: **how** to **create (complex) structures**, **how** to **create** and **change** the **behavior** and **structure** of **objects during runtime** and how to **make classes work together**.

OVERVIEW OF STRUCTURAL PATTERNS

Adapter –Adapts objects and classes to the needs of your code, allowing your code to work with that object and class
Bridge –Is a polymorphic Class. Bridge allows you to change behavior of the Context object on runtime. It uses delegation of all actions to one of many possible objects implemented by Bridge to do the real work.
Composite – Describes the class-structure to create nested object structures.
Decorator – Uses a base interface to create several variations on a specific object and Class which can be nested into each other to add specific qualities.
Flyweight – Stores and shares objects for re-use in your application. Used mostly for objects which are very heavy in memory use.
Façade – Simplifies access and use to a subsystem. Combines actions on that system in single methods and these behind the scenes.
Proxy – Acts as a man in the middle between your code and the concrete object. Can represent remote objects and objects not there yet. Can manage references and access to concrete objects and protect them by adding extra (security) layers.

REMARKS

Patterns with a *
Patterns with a * are not part of the 23 GoF Design Patterns.

LINKS

GoF Design Patterns overview:
Behavioral Patterns
Creational patterns
Structural Patterns

WHAT FITS WHERE? OVERVIEW #2

DESIGN PATTERNS BY TYPE AND USE

WRAPPING, INTERFACING, ABSTRACTION

TYPE 1: WRAPPERS OF OBJECTS AND CLASSES

Adapter – Wraps objects (Object Adapter) and clases (Class Adapter)
Decorator – Wraps objects and other decorators, adds functionality
Delegate – Wraps the execution process, can delegate internally to any object that is accepted and capable to do the job.
State – Wraps the processes in separate Concrete States
Bridge – Wraps the process of selection, delegation and execution against concrete implementation of the tasks the Context object is supposed to do.
Proxy – Wraps and abstracts the concrete object it represents.

TYPE 2: WRAPPERS OF SUBSYSTEMS

Façade, Mediator, Manager, Operator – Offer a simplified interface to the complexity of the system it "wraps" Centralizes communication in some way. "Wrap" the subsystem by taking the role of point of access.

CREATION OF DYNAMIC CLASSES

TYPE 1: DYNAMIC CLASSES AND OBJECTS

Bridge – Implements one or more delegates to delegate the execution of actions to. Is usually extended by other classes as base-class. By changing the delegate, the behavior of the Bridge changes as if it is a different class. Referred to here as "polymorphic class"

TYPE 2: DYNAMIC EXECUTION OF ACTIONS

Delegate, State, Strategy – How actions are implemented and executed is defined by the delegates used by and in these patterns. These actions can include behaviors and ways to handle a process.

TYPE 3: DYNAMIC EXTENTION

Decorator – The decorator allows you to add functionalities to your class and object by decorating it with these extra functionalities.

DATA AND DATA TRANSFER (1)

TYPE 1: INJECTING VALUES INTO OBJECTS

Parser, Injector, Reflection – All can be used to inject values into an object.

TYPE 2: STORING AND RETRIEVING DATA

Identity Map, Object Map, Multiton – Can be used to store objects and data in a central place, under specific keys for specific retrieval.

SHARING AND RE-USING OBJECTS

TYPE 1: SHARING AND RE-USING OBJECTS

Singleton, Multiton, Flyweight – All take care of creation and sharing of a limited set of objects in your project. Singleton only creates one. Multiton and Flyweight as many as you need, stored under keys
Ideitnty Map, Object Map – All store objects under an identity key, usually referring to that of the entity the object represents.
Object Pool – Focuses on re-use more than sharing. While objects in the Pool are "shared", this is not its purpose.

CREATING OBJECTS AND STRUCTURES

TYPE 1: CREATING AND INSTANTIATING OBJECTS

Simple Factory, Factory Method, Abstract Factory – All deal with the (abstraction of) creation of objects. You call the construction method and it will return the requested object.
Factory Object Map, Factory Object Pool, Extended Identity Map – Each centralize the creation and management of objects. These patterns first check if the requested object is already there.

TYPE 2: CREATING AND READING COMPLEX OBJECTS

Parser, Builder, Iterator – Each are capable of building complex objects using different ways to do so.
Composite – Is the basis for any complex object, defining a structure in which objects can have one or more children.
Iterator, Visitor, Reflection – Are all methods to read and use the complex object structure as defined in (usually) the Composite object.

DELEGATION OF ACTIONS

TYPE 1: DELEGATION OF ALL ACTIONS

Delegate, Adapter, Command, State, Bridge – Delegate all actions to a delegate object. This can be the Adaptee, an injected object (Command) or a context-specific implementation of actions (Delegate, State, Bridge).
Proxy – Acts as a man in the middle all actions to object it represents.

TYPE 2: DELEGATION OF SOME ACTIONS

Strategy – The Strategy pattern assumes your object will do most of the tasks and only delegate those actions you need a variable implementation for.

TYPE 3: DELEGATION TO SUBSYSTEMS

Façade, Mediator, Manager, Operator – While offering a simple interface and centralizing actions, each of these patterns delegate the real work to one or more subsystems.

DATA AND DATA TRANSFER (2)

TYPE 3: WORKING WITH EXTERNAL DATA

Remote Proxy, Data Acces Object – Can be used to approach and work with external data and external systems.

EVENTS, INSTRUCTION, COMMUNICATION

TYPE 1: DIRECT COMMUNICATION

Mediator, Manager, Operator – Each strive to centralize communication within a specific context. This can be by calling (static) methods or via events in the Observer pattern.

TYPE 2: DECOUPLING DEPENDENCIES

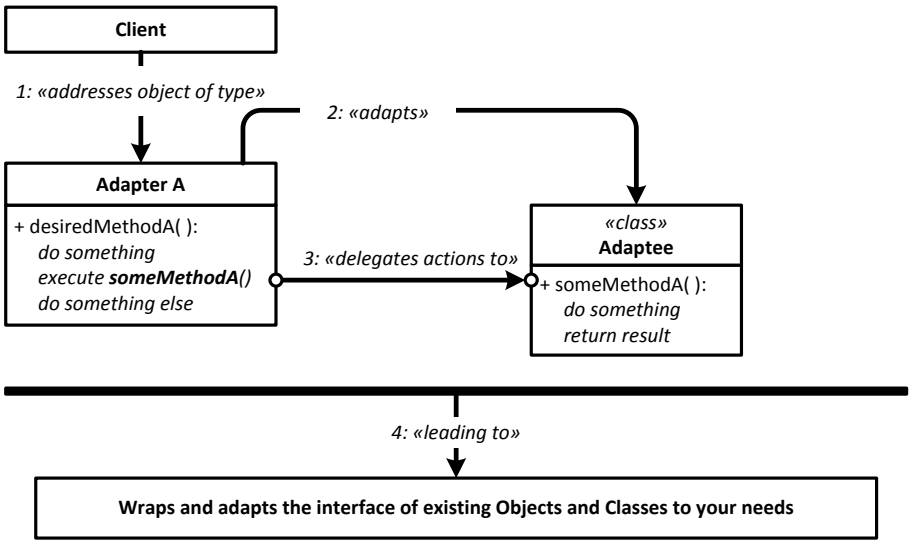
Observer, Command – Decouple dependencies between the user and the classes executing the required actions. Both patterns remove the need for the remote party to know who or what they are addressing.

Link: **Patterns ordered by type and use**

DESIGN PATTERNS #1

ADAPTER (CLASS/OBJECT)

Link: Adapter



Visual summary of the Adapter Pattern

INTENT

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

DEPENDENCIES

Client:
1: Addresses object of type Adapter A
Adapter A:
2 Adapts Adaptee
Desired Method:
3: Delegates actions to Adaptee

RESULT

Wraps and adapts the interface of existing objects and classes to your needs

BASICS

WHEN/WHAT?

When you need to adapt Incompatible Interfaces
You use the Adapter when the Class you want to use has an Interface that is not compatible with the needs of your code.

Two variations
The Adapter Pattern knows two variations:

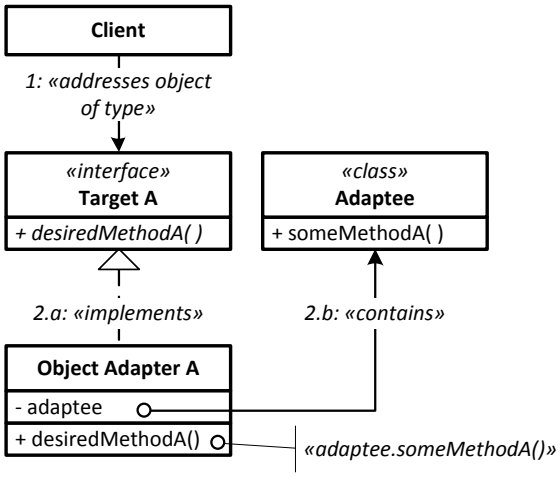
1: The Object Adapter
The Object Adapter adapts an object by wrapping it and delegating the required actions to the Adaptee.

2: The Class Adapter
The Class Adapter extends the Class that needs to be adapted, applies the required Interface your code desires and – like the Object Adapter – delegates the actions to the Adaptee, which is used as a Base Class.

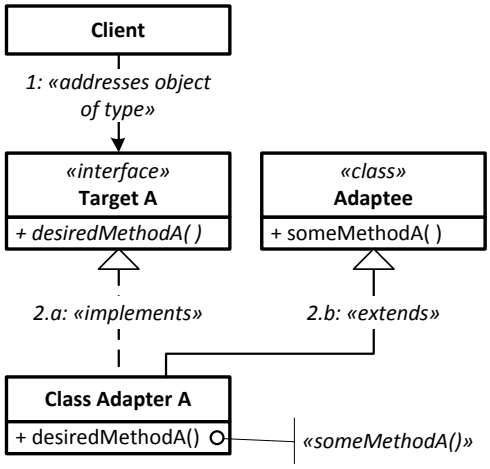
OTHER INFO

Using delegation
The Adapter uses delegation of actions to do its work.

CLASS DIAGRAM



Object Adapter: adapting the object by wrapping it

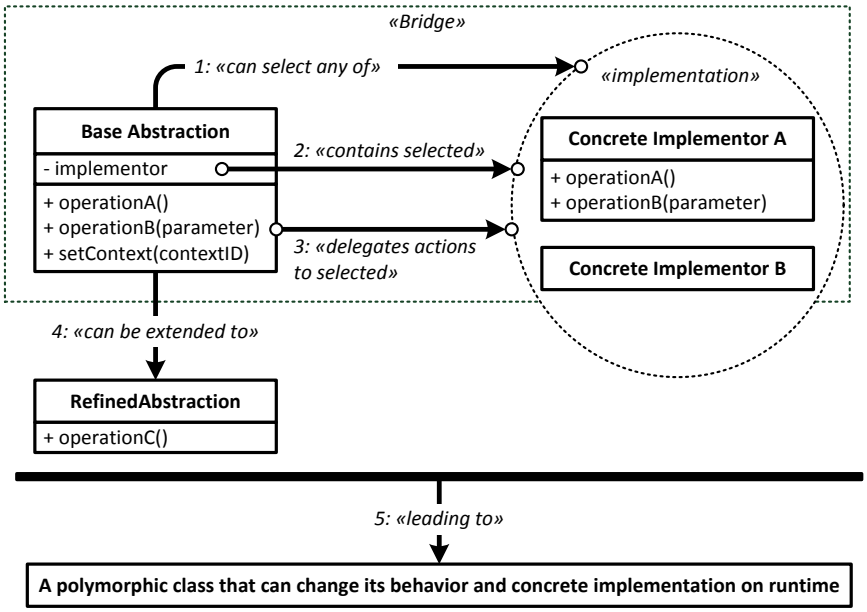


Class Adapter: adapting the class by extending it

DESIGN PATTERNS #2

BRIDGE

Link: Bridge



Visual summary of the Bridge Pattern

INTENT

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

DEPENDENCIES

Base Abstraction:
1: Can select any of the Concrete Implementors exposing the same interface
Implementor variable:
2: Contains selected Concrete Inplementor
Methods:
3: Delegate actions to selected Implementor
Base Abstraction
4: Can be extended to Refined Abstraction

RESULT

A polymorphic class that can change its behavior and concrete implementation on runtime

BASICS

WHEN/WHAT?

When you need a Class that can change its behavior/implementation
You use the Bridge when you need a Class that can change its behavior and concrete implementation when needed.
A Dynamic Base Class
The Bridge is intended to act as a Dynamic Base Class, to be extended by your code. While you only extend one Base Class, that Base Class can internally instantiate any concrete implementation of a specific functionality.
A Pretender
The Bridge can be seen as a Pretender of the Classes it can Instantiate. While Your code thinks it is working with one single object or Class (the Bridge), it is actually interacting with the Instantiated object wrapped by the Bridge.

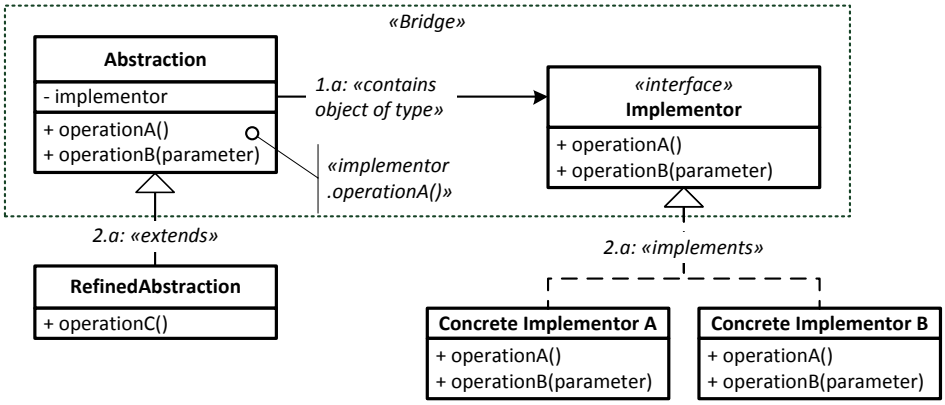
OTHER INFO

Using delegation
The Bridge uses delegation of actions to do its work.

Using a Factory or Object Map to get the delegate
One way to look at the Bridge is as a wrapper around the Factory or Object Map. Using either the Factory or the Object Map you can return any object in a selected group and use it as the delegate for the delegation of actions.

SIMILAR PATTERNS

CLASS DIAGRAM



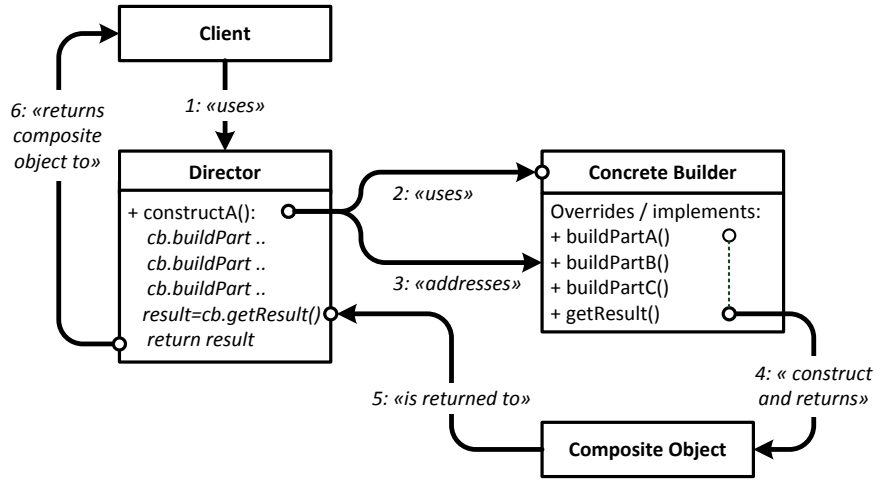
DESIGN PATTERNS #3

BUILDER

Link: Builder

INTENT

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



Visual summary of the Builder Pattern

DEPENDENCIES

Client:
1: Uses the Director
Methods in the Director:
2: Uses the Concrete Builder
Methods:
3: Addresses build-instructions in the Concrete Builder
Build-instructions:
4: Construct and return Composite Object
Composite Object:
5: Is returned to the Director
Director:
6: Returns object to Client

RESULT

A dynamic way to build complex objects using a basic set of building instructions

BASICS

WHEN/WHAT?

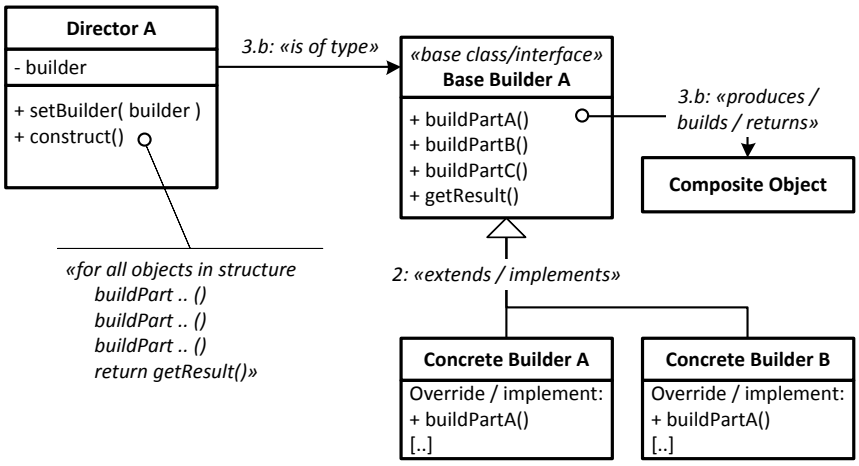
To build dynamic structures
The Builder pattern is used to allow you to build several versions on the same product, using object composition.

Fixed set of build-instructions, variable outcome
While the Builder Pattern can have fixed building instructions inside the Director, the Builder Classes can implement these instructions each in a different way, leading to the same kind of constructions with completely different implementations. See the illustration below.

OTHER INFO

Builder and Composition
In most cases, the Builder will produce a Product using the Composite Pattern.

CLASS DIAGRAM



DESIGN PATTERNS #4

CHAIN OF REPSONSIBILITY *

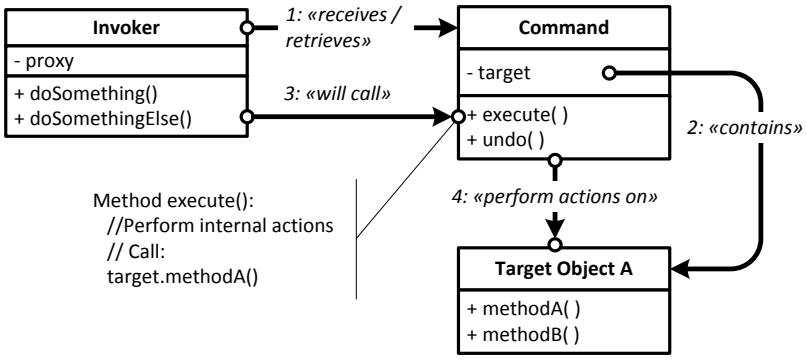
Link: Chain of Responsibility

Visual summary of the Chain of Responisbility Pattern

DESIGN PATTERNS #5

COMMAND

Link: Command



Visual summary of the Command Pattern

INTENT

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

DEPENDENCIES

Invoker:
1: Receives/ retrieves Command
Command:
2: Contains Target Object A
Client methods:
3: Will call "execute" method on Command
Execute method:
4: Will perform actions on Target Object A, can perform internal actions as well.

RESULT

A way to decouple, abstract and package any action from the objects on which these actions are performed

BASICS

WHEN/WHAT?

Decouple, abstract and package any action
The Command offers a way to decouple, abstract and package any action from the objects on which these actions are performed.

Sending command objects into your project, have them executed elsewhere
The Command Pattern allows you to send objects into your project that contain specific callbacks to specific parts of your code. Combined with an Object Map you can even create a system that has no dependencies between the caller and callee of a specific Command.

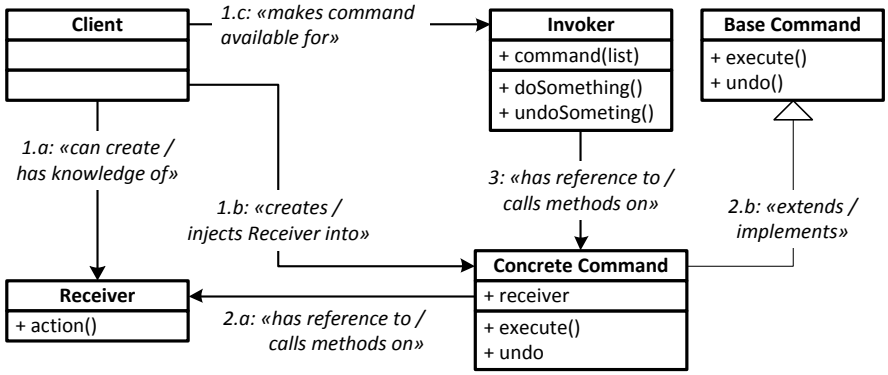
Inversion of control
Commands allow for Inversion of Control by separating the definition of the Command from the actual execution. Each and any Command can be executed to perform a specific action and by sending a different Command for the same action, you can change the location where and way in which that command is executed.

OTHER INFO

Who creates and passes the Command to Some System?
The Command can be passed and created by the "Client". (See class diagram below). This can be any other object in your project.

Using an unified Command Interface
Each Command implements exactly the same Interface as any other Command in your collection. While each Command can be handling completely different actions on completely different objects, you can use each and all Commands in exactly the same way. Depending on the type of Command, this Interface shares the methods "execute()" and, if there is an Undo: "undo()"

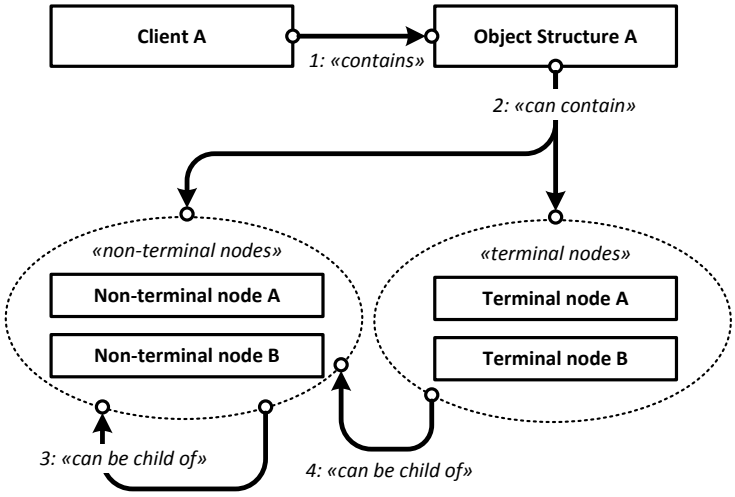
CLASS DIAGRAM



DESIGN PATTERNS #6

COMPOSITE

Link: Composite



Visual summary of the Composite Pattern

INTENT

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

DEPENDENCIES

Client A:
1: Contains Object Structure A
Object Structure A:
2: Can contain terminal and non-temrinal nodes
Non-terminal nodes:
3: Can be children of non-terminal nodes
Terminal nodes:
4: Can be children of non-terminal nodes.

RESULT

A way to build an object tree with nodes and children, that can be of any depth

BASICS

WHEN/WHAT?

Parent/child relationships
The Composite pattern describes objects that can have parent/child relationships.

Terminal and non-terminal nodes
The Composite Pattern makes explicit mention of terminal and non-terminal nodes.

Used in almost any situation where you build dynamic structures
Each and all software you use implements at least one Composite. For instance: the web-pages you see in your web browser are rendered from composite objects derived from the HTML your browser received. Applications like Word, Excel and Powerpoint do similar things with the content they load and present and store again.

OTHER INFO

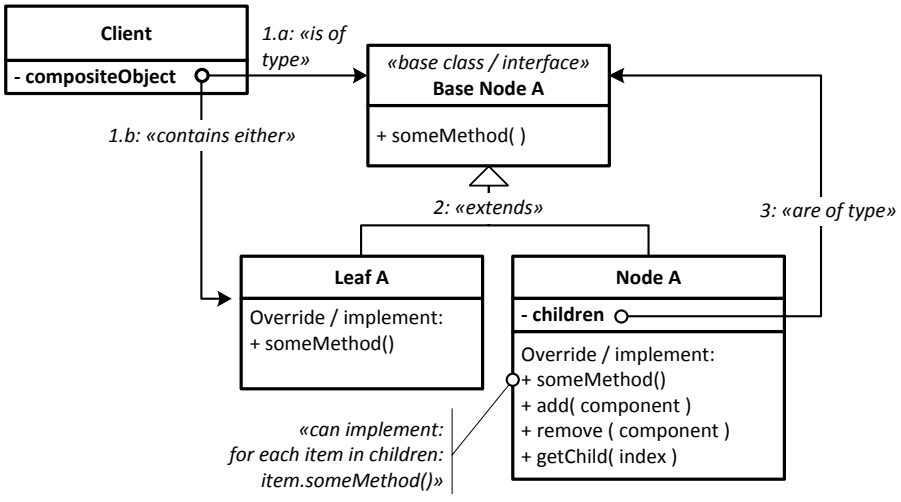
Iterating the Composite object
The Composite can be iterated (run through) to perform actions or get data using a Parser, a Visitor or an Iterator. Each of these patterns have their advantages and drawbacks.

Using as the basis for another structure
When you parse data, one Composite object can be the basis for another, for instance when you read a representation of a real-world object and you want to extract only specific aspects or parts of it or save it as an abstract definition to your hard drive.

Building a composite
The Builder, Parser and Interpreter patternsn use the Composite pattern as the basis for the objects they use to build composite object structures.

SIMILAR PATTERNS

CLASS DIAGRAM



DESIGN PATTERNS #7

DATA ACCESS OBJECT *

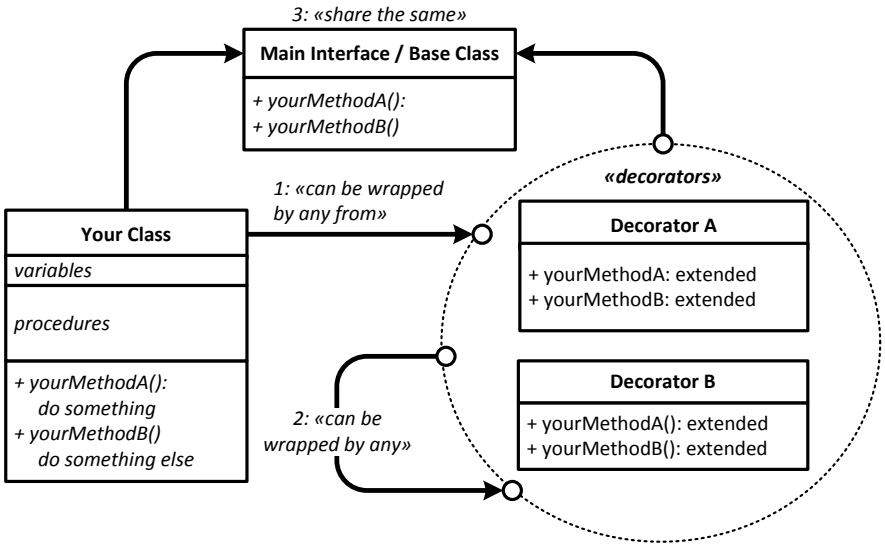
Link: Data Access Object

T

DESIGN PATTERNS #8

DECORATOR

Link: Decorator



Visual summary of the Decorator Pattern

INTENT

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

DEPENDENCIES

Your Class:
1: Can be wrapped buy any
Any Decorator:
2: Can can be decorated by any other Decorator form the same collection
Decorators and Your Class:
3: Share the same Main Interface / Base Class

RESULT

Extended functionalities on existing objects

BASICS

WHEN/WHAT?

To dynamically add and change functionalities to/on an object
The Decorator allows you to dynamically add functionalities to an existing object. This can either be extra components and parts when it is a visual component, or extra and different actions in methods it exposes to the world.

Wrap-and-add
The Decorator wraps the object you Inject and then adds new functionalities to it.

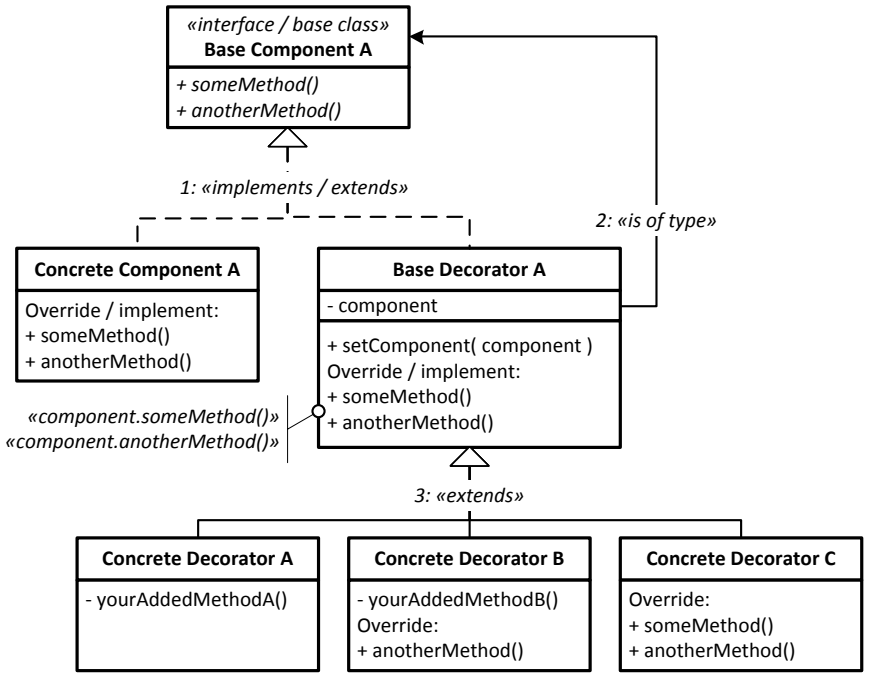
OTHER INFO

Injection instead of internal creation
The Decorator differs from patterns in the way it obtains the object it wraps. Where the Adapter usually creates the Adaptee, the Decorator gets that object via Injection by your code.

Decorators should not exceed the basic interface
The Decorators should not expose additional public methods as these Decorators will be decorated themselves, making these additional "rogue" methods unreachable from other objects.

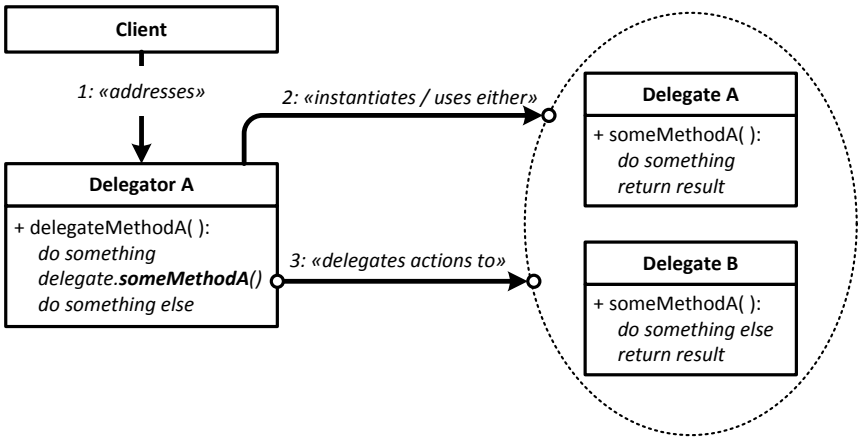
SIMILAR PATTERNS

CLASS DIAGRAM



DESIGN PATTERNS #9

DELEGATE



Visual summary of the Delegate Pattern

Link: Delegate

INTENT

To delegate and abstract the actual execution of an Action to another object. To offer one single Interface for the execution of these Actions, independent of the concrete implementation of these Actions.

DEPENDENCIES

Client:
1: Addresses Delegator A
Delegator A:
2: Instantiates uses either Delegate A or B
Delegate Method:
3: Delegates actions to either Delegate A or B

RESULT

Plug & play code: a cleaner way to define who executes what, how, when

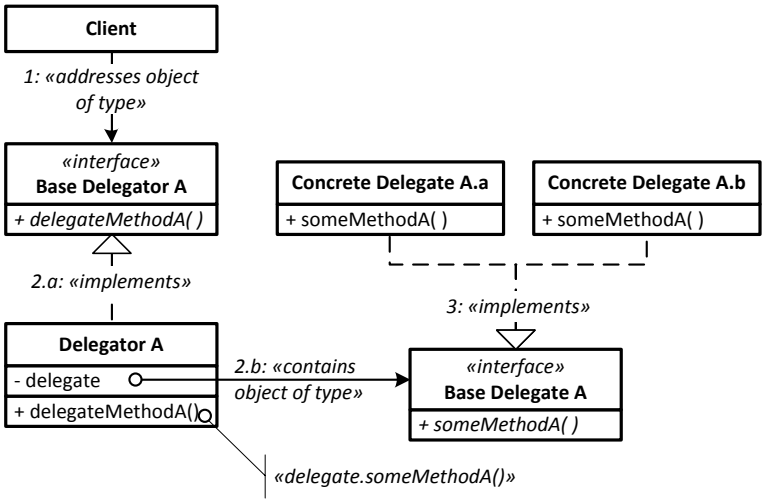
BASICS

WHEN/WHAT?

Delegating actions to other objects
The main goal of the Delegate is to delegate the execution of Actions to other objects it can either retrieve or instantiate based on specific parameters.

Plug & play code:
a cleaner way to define who executes what, how and when
If and when you want to have a flexible way to execute specific processes, you can follow several different routes, including conditional execution, where a Switch or and if/then/else statement separates the different approaches.
In the Delegate Pattern, you extract these approaches and place them in a separate Class, allowing you to create "plugins" that execute specific actions in a specific way. To change the way an action is executed, you simply replace "Delegate Object A.a" for "Delegate Object A.b".

CLASS DIAGRAM



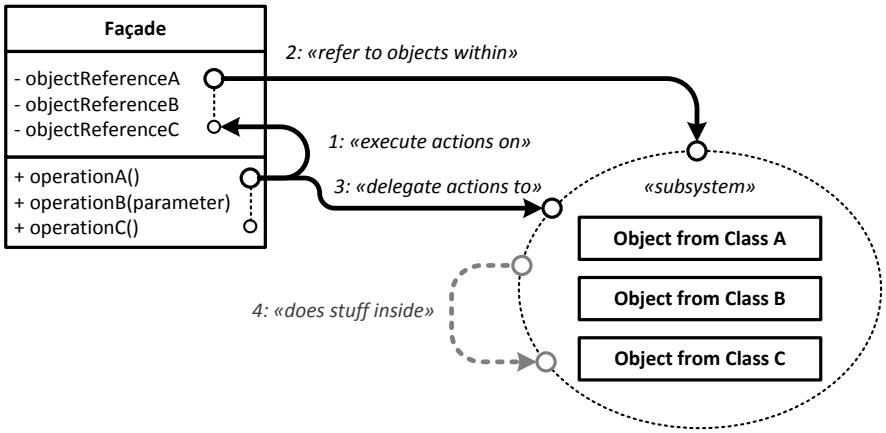
OTHER INFO

Can offer multiple actions to be executed via Delegation
The Delegate can expose one or more methods to be executed by Delegation via another object.

Allows your system to expand possibilities without the need to rewrite your base code
In most cases, where the selection and use of solutions are hardcoded in the application, adding new functionalities lead to a partial re-write and update of that code. Instead, when you use the Delegate Pattern wisely, you can add any new Delegate to the list, without having to change any line of code.

DESIGN PATTERNS #10

FAÇADE



Visual summary of the Façade Pattern

Link: Façade

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.

DEPENDENCIES

Operation A, B, C:
1: Executes actions on object reference A, B or C
Object references A, B, C:
2: Refer to objects within the subsystem
Operation A, B, C:
3: Delegate actions to the subsystem
Objects from Class A, B, C
4: Can do stuff inside the subsystem

RESULT

Simplification of calls to- and actions on the subsystem

BASICS

WHEN/WHAT?

When you want to wrap a subsystem
The Façade wraps a subsystem by offering a simplified interface to the actions in that subsystem. It deals internally with alle the actions you would otherwise have to code each time you access those functionalties.

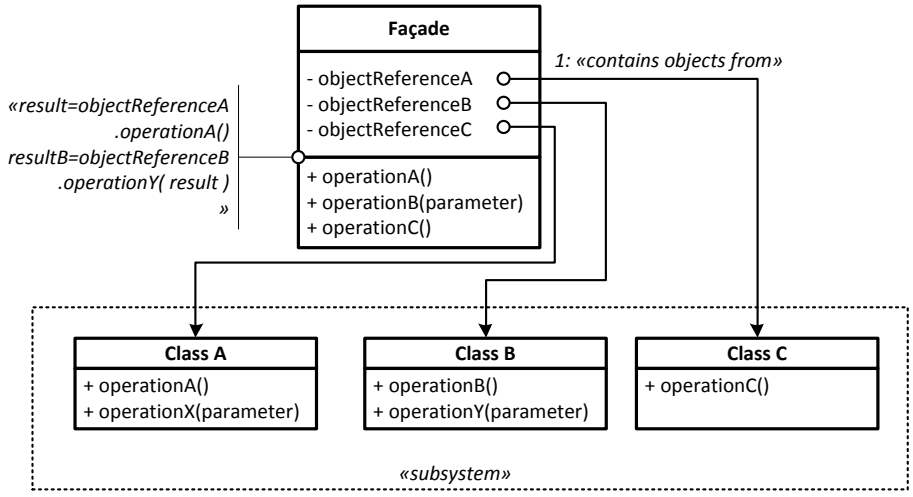
Simplification of the use of that subsystem
The simplification can happen, for instance, by gathering a sequence of specific actions into one or two methods and do all the hard stuff inside these methods.

OTHER INFO

Delegation
The Façade operates by delegating all tasks to the objects in the subsystem it addresses.

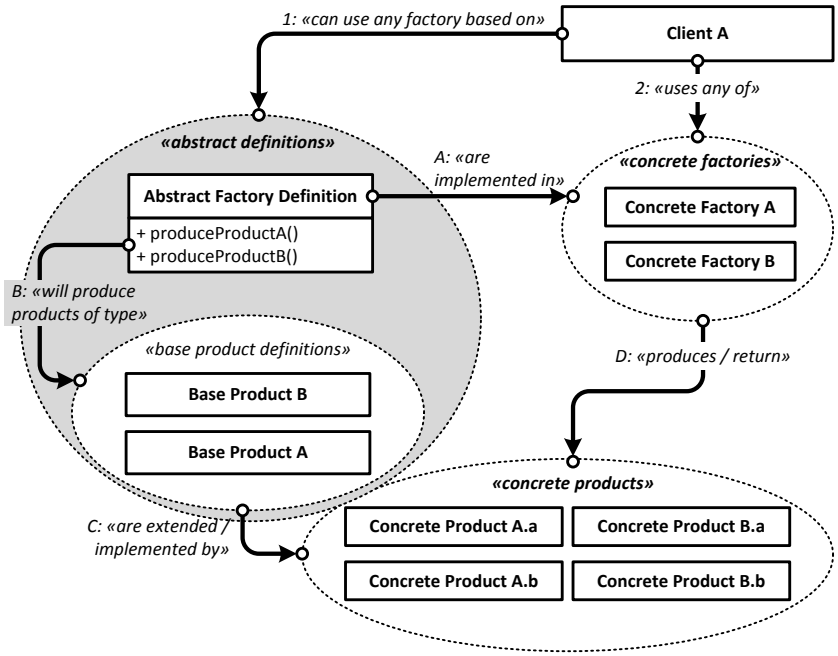
Multiple Façades for multiple tasks
The Façade can be implemented many times to perform specific and completely different tasks on that and other subsystems in your project.

CLASS DIAGRAM



DESIGN PATTERNS #11

FACTORY, ABSTRACT



Visual summary of the Abstract Factory Pattern

Link: Factory, Abstract

INTENT

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

DEPENDENCIES

Client A:
1: Can use any factory of abstract definitions
2: Uses any of Concrete Factory A, B ..

Abstract definitions:
A: Are implemented in Concrete Factory A, B
B: Will produce prodcuts of type Base Product A, B

Base Product A, B
C: Are extended / implemented by Concrete Products

Concrete Factories A, B
D: Produces / return Concrete Products A.a, A.b, B.a, B.b

RESULT

Blueprints for interchangeable factories with very specific production lines producing specific products

BASICS

WHEN/WHAT?

When you need to create a line of different factories
The Abstract Factory is a Pattern that describes how you can define the interfaces for a set of factories you can interchange for any of the other, to produce very specific products.

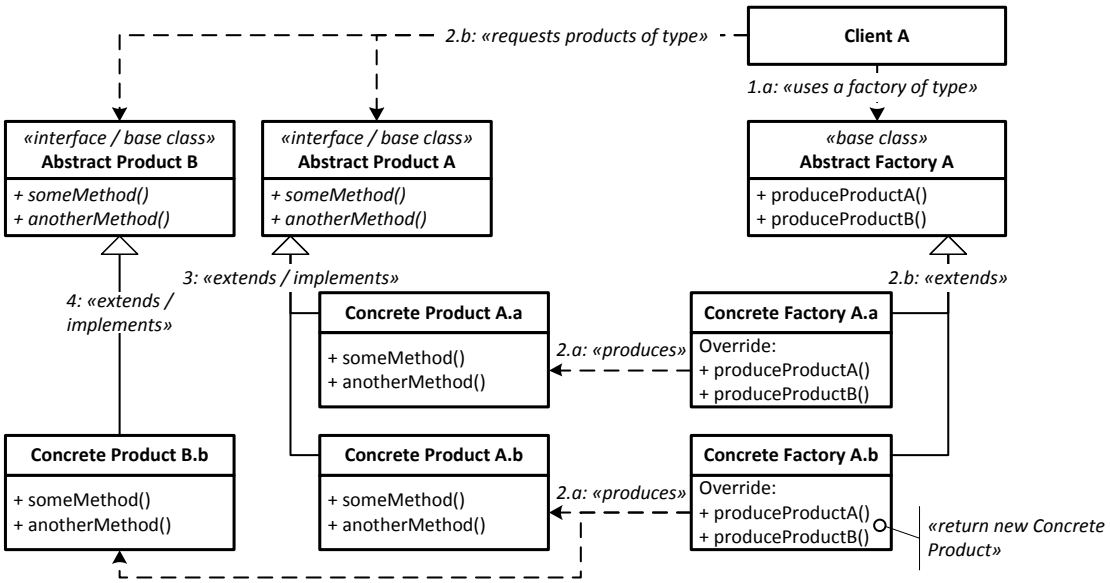
Blueprint for Concrete Factories
The Abstract Factory is basically a Blueprint for Concrete Factories. The Abstract Factory Pattern uses this principle to create multiple Concrete Factories based on the Abstract Factory and to allow you to choose any of them.

OTHER INFO

Different implementations of a Factory per context
The Abstract Factory Pattern becomes useful when your code works within one or more specific Contexts, use the same basic Products to perform its actions, but needs a different Implementation for each specific Context.

Simple Factory is more to the point
If you do not need to create many different factories based on the same template, use the Simple Factory instead.

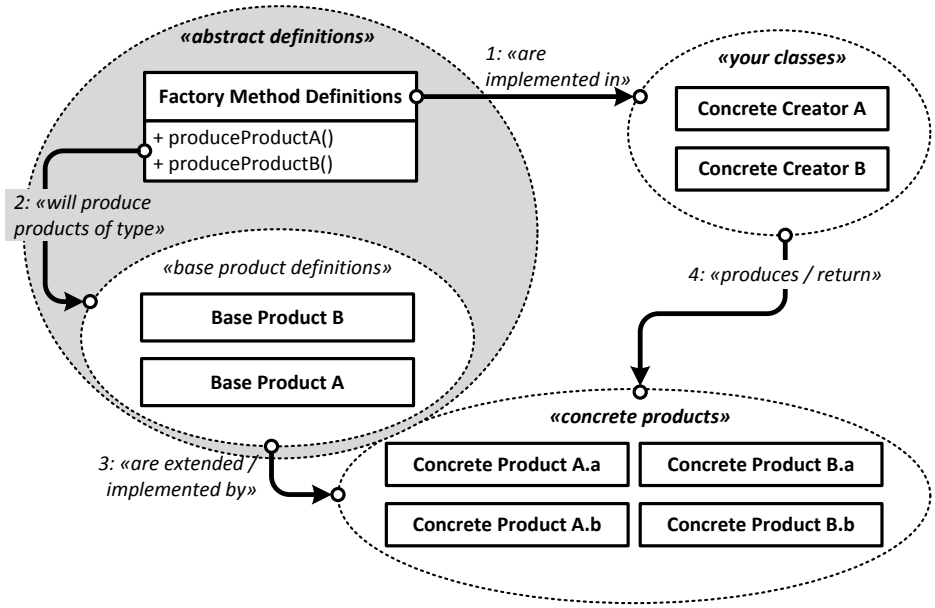
CLASS DIAGRAM



DESIGN PATTERNS #12

FACTORY METHOD

Link: Factory Method



Visual summary of the Factory Method Pattern

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.

DEPENDENCIES

Factory Method Definitions:
1: Are implemented in your classes
Factory Method Definitions:
2: Will produce products of type Base Product A, B ..
Base Product A, B
3: Are extended / implemented by Concrete Products
Concrete Creators A, B
4: Produces / return Concrete Products A.a, A.b, B.a, B.b

RESULT

Re-usable definitions for factory methods to produce concrete products in several classes

BASICS

WHEN/WHAT?

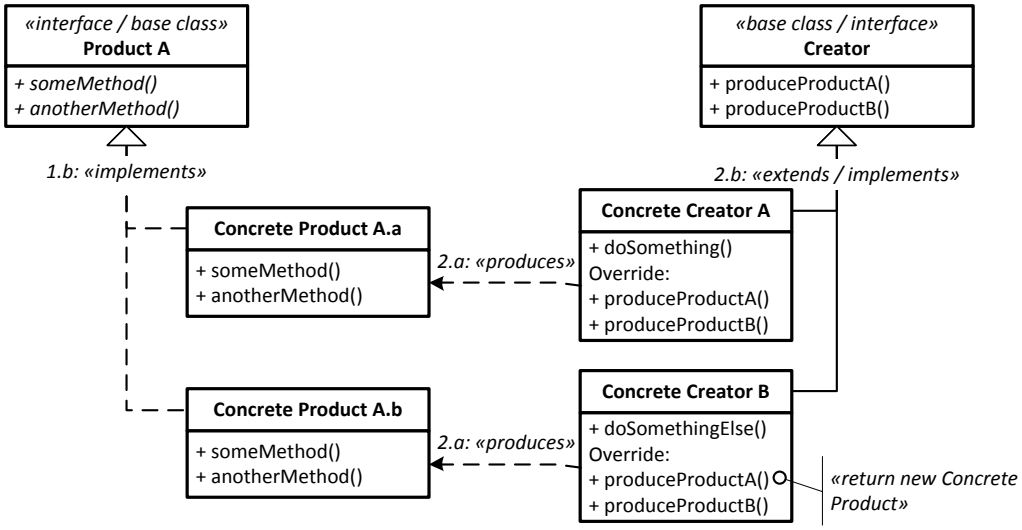
When you need to define a blueprint for Factory Methods
The Factory Method is mostly a way to define the blueprint with which you will create Factory Methods within your own Classes.

Starting point for Abstract Factory
As you will find when you study the Abstract Factory Pattern, the Factory Method Pattern is almost completely implemented there as well. The main difference is that the Factory Method assumes you will implement the Factory Methods in your own Concrete Classes, while the Abstract Factory assumes you will extract and abstract these Factory Methods into separate Concrete Factories.

OTHER INFO

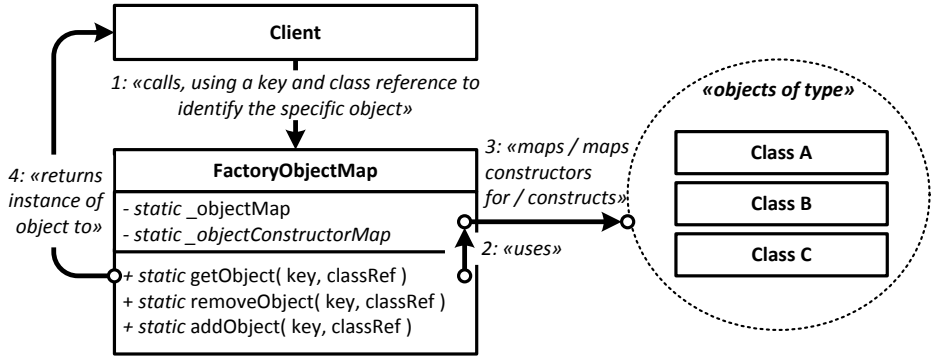
Simple Factory is more to the point
If you do not need to create blueprints for your Factory Methods, but simply want to implement them, use the Simple Factory instead.

CLASS DIAGRAM



DESIGN PATTERNS #13

FACTORY OBJECT MAP



Visual summary of the Factory Object Map Pattern

Link: Factory Object Map

INTENT

Ensures that each entity is represented only once in your application by mapping them against their identity and class reference. Centralized lookup and creation of objects in and from that map.

DEPENDENCIES

Client:
1: Calls Factory Object Map, using a key and class reference to identify the specific object
getObject in Factory Object Map:
2: Uses Object Map to try and retrieve the requested object, creates and maps new object when Object Map does not contain object yet.
Object Map and Constructor Map:
3: Maps objects and constructors
getObject
4: Return instance of object to Client

RESULT

Creation and mapping of one instance of a specific object of any type to be retrieved and used anywhere

BASICS

WHEN/WHAT?

When you want to encapsulate & control object mapping/object creation
The Factory Object Map encapsulates the object creation process. As it maps all objects created before, it takes full control over that object creation. Due to the encapsulation of the creation-process, the Factory Object Map automatically takes care of mapping of newly create objects as well.

When you want only one instance of a specific entity
The Object Map holds all objects created during runtime. If and when you request an object with a specific ID, it will attempt to resolve it from the map first. Only when the object with that ID is not in the map, the Factory Object Map will produce a new one.

Inversion of control and dependency injection
The Factory Object Map allows for implementations in which the concrete object constructors are defined separately from the implementation of the Factory Object Map, making is possible to change the concrete classes used for instantiation of objects without impact on the implementation of the Factory Object Map or the code using the Factory Object Map.

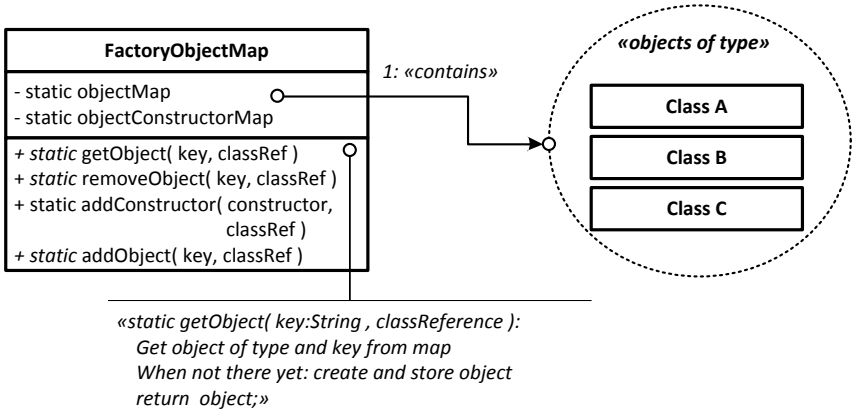
OTHER INFO

Extended Identity Map
The Extended Identity Map might as well have been called Factory Identity Map when I wrote that variation down. It does the same as the Factory Object Map, but in a more simple structure.

One Factory Object Map to do it all
Like the Object Map, the Factory Object Map can be used to store any and all maps you need in your application, limiting the amount of Factory Objects Maps in your project to one.

Using Constants to identify the classes / base interfaces
Instead you could – and probably should – create and use constants to identify and map the classes you use to create and retrieve the objects.

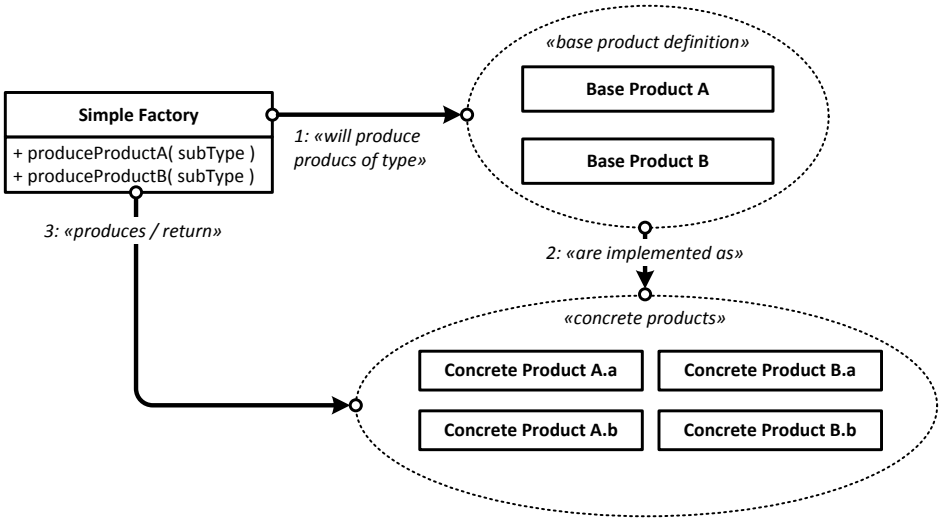
CLASS DIAGRAM



DESIGN PATTERNS #14

FACTORY, SIMPLE

Link: Factory, Simple



Visual summary of the Simple Factory Pattern

INTENT

Implement a method or class to create objects based on specific parameters (the Context or product type). The Simple Factory chooses and instantiates the required class and returns the result to the caller.

DEPENDENCIES

Simple Factory:
1: Will produce prod. of type Base Product A, B
Base Product A, B:
2: Are implemented on Concrete Products
Simple Factory Methods:
3: Produce / returns Concrete Products A.a, A.b, B.a or B.b

RESULT

A very simple way to produce concrete products based on specific parameters

BASICS

WHEN/WHAT?

To create a simple factory without the boilerplate code
You use the Simple Factory to create a factory without the boilerplate-code of the Abstract Factory and Factory Method.

Instantiates objects based on a context
The Factory intantiates (produces) objects based on a specific context or type. For instance, cars of type "A", "B" and "C" which all share the same interface, but each have a different implementation.

OTHER INFO

Can be implemented in separate Class or in your code
The Simple Factory can be implemented in your code or in a separate Factory Class.

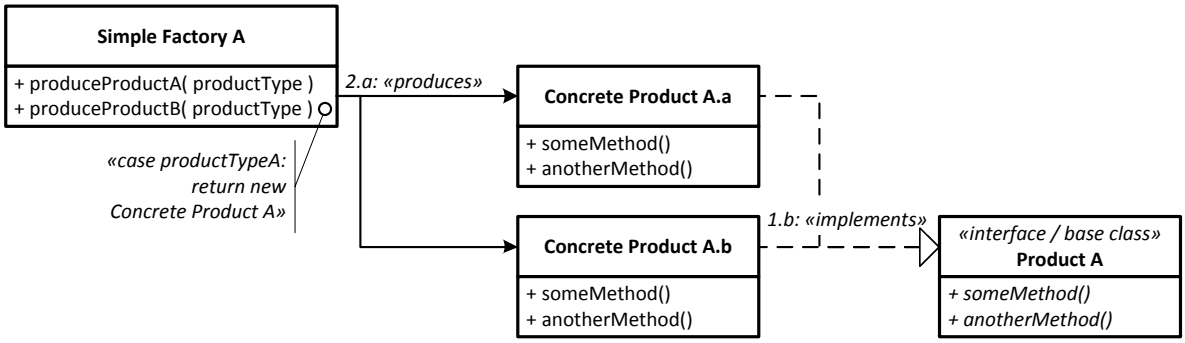
Returns products
Like its bigger brothers the Simple Factory produces and returns products.

Can have multiple factory methods
The Simple Factory allows you to inplement multiple factory methods.

Each factory method has an object subtype
Each method in the Simple Facotry produces one or more products of a specific subtype. Each of these products share the same Interface but comes from a differen Class and has a different implementation.

Useful for Builder, Parser and Interpreter
The Simple Factory is a very useful Pattern for the Builder, Parser and Interpreter as it can produce the objects these Patterns need to build a Composite structure.

CLASS DIAGRAM



DESIGN PATTERNS #15

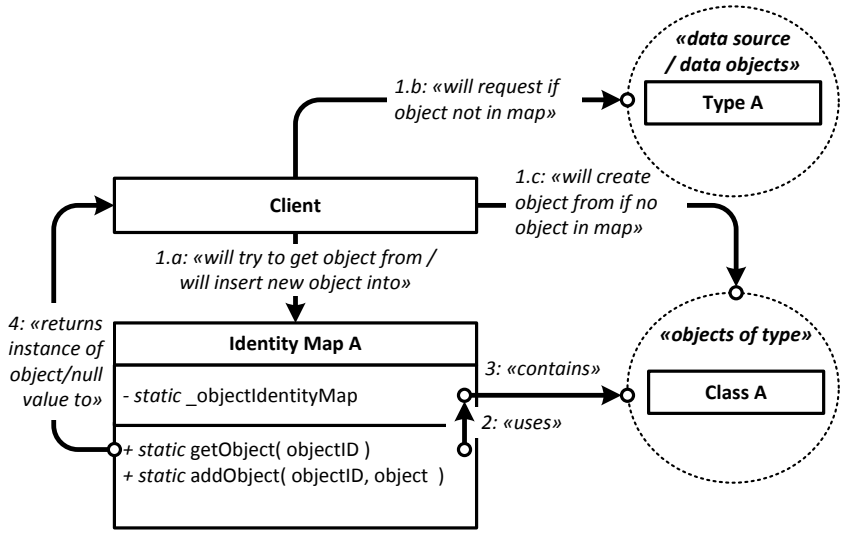
FLYWEIGHT *

Link: Flyweight

DESIGN PATTERNS #16

IDENTITY MAP

Link: Identity Map



Visual summary of the Identity Map Pattern

INTENT

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.

DEPENDENCIES

Client:
1.a: Will try and get object form Identity Map A / will insert ne object into Identity Map A
1.b: Will request object from data source / data objects when not in map
1.c: Will create object from Class A if object not in map
getObject in Identity Map A:
2: Uses objectIdentityMap to retrieve object
Object Identity Map:
3: Contains objects of type Class A
Identity Map A:
4: Returns instance of object or null value to client

RESULT

A container for (data) objects which can be retrieved once created,
so that each (data)item has one single object to contain and represent it

BASICS

WHEN/WHAT?

Storing objects into a map
The Identity Map Pattern stores objects into a map, using an identity to retrieve them again

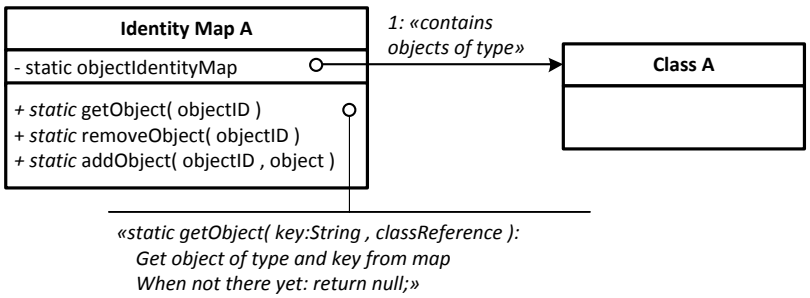
Loading objects only once
The idea of the Identity map is to load objects only once. In the case of a database, instead of requesting the same data-objects over and over again from the external source, you only request the data objects you do not have yet. This reduces the load on your backend-system.

Centralizing data management
The Identity Map centralizes data management within your project. There is only one class (per type of object) to address for the objects (of that type) that you might want to use in your project. When using the Identity Map, you are sure that the object you request is the only one representing that specific entity with that specific ID.

OTHER INFO

Database only?
While the Identity Map is associated to be used in conjunction with a database (or external data source), it is not the only use-case where you might like and want to store objects under specific identities.

CLASS DIAGRAM



DESIGN PATTERNS #17

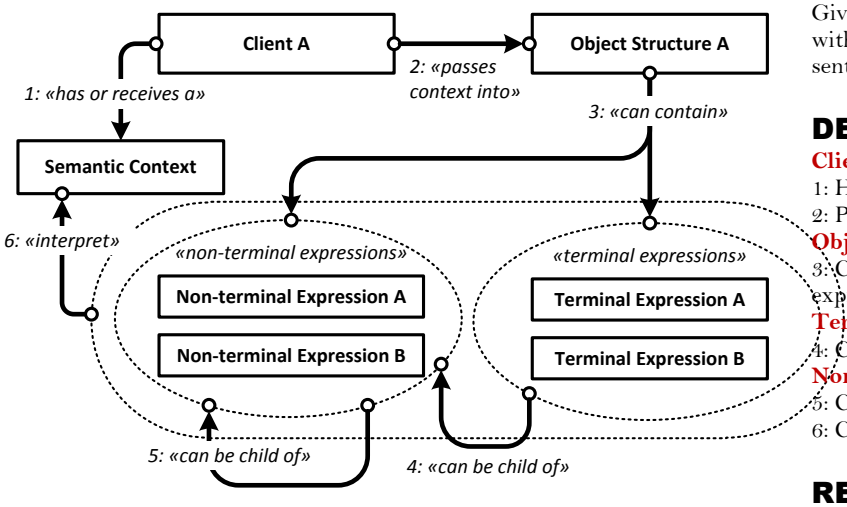
INJECTOR *

Link: Injector

DESIGN PATTERNS #18

INTERPRETER

Link: Interpreter



Visual summary of the Interpreter Pattern

INTENT

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

DEPENDENCIES

Client A:
1: Has or receives a Semantic context
2: Passes context into Object Structure A
Object Structure A:
3: Can contain non-terminal experssions and terminal expressions.
Terminal expressions:
4: Can be child of non-terminal experessions
Non-terminal expressions:
5: Can be child of other non-terminal expressions
6: Can interpret (like the terminal ones) the Semantic Context

RESULT

An object structure interpreting and representing the semantic relationships as present within the Semantic Context Object

BASICS

WHEN/WHAT?

To interpret a semantic structure
The Interpreter takes a semantic structure and translates it into an object model, representing that structure. The semantic structure can be natural language, but also program-code and instructions you can give to machines.

Deconstructing and interpreting semantic structure
Semantic structures are comprised of words or items of meaning placed within a specific context and order. The context, order and meaning of each individual item in the total structure combined defines the meaning and intend of a sentence, line, paragraph, method and set of instructions.

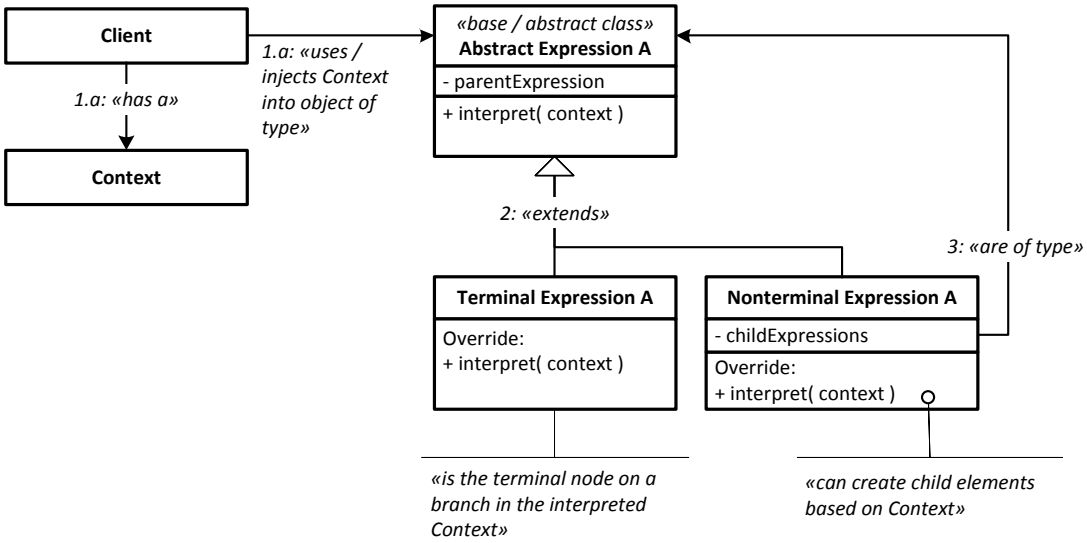
OTHER INFO

Composite pattern
The Interpreter creates a structure using objects with a very similar setup as the Composite pattern.

Visitor
The Visitor pattern can be used to run through the composite object that is the result of the interpretation, to read the structure and distill meaning from it.

Compilers, bots, search engines and word processors
The Interpreter pattern can be used to create compilers; to create bots that reply to queries from people; in search engines to break down your search request and interpret the content from the pages the search bots spider; in word processors to check if your sentences are setup properly.

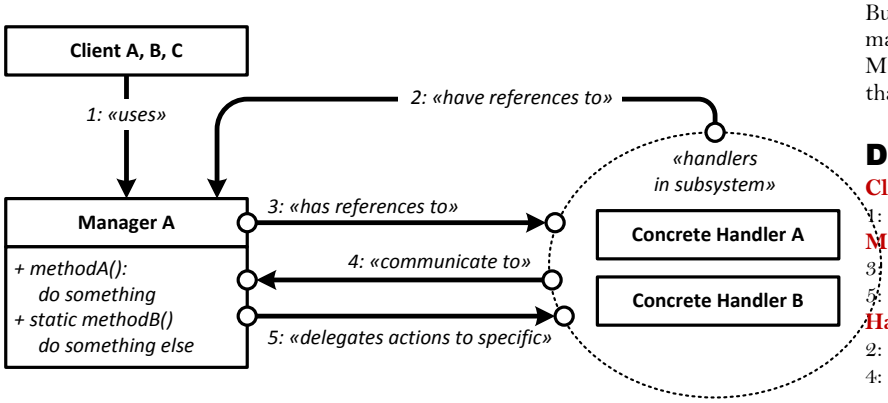
CLASS DIAGRAM



DESIGN PATTERNS #19

MANAGER

Link: Manager



Visual summary of the Manager Pattern

INTENT

Define an object that encapsulates and centralizes Business Logic for a specific set of actions and acts as a manager for the processes that happen in the subsystem. Make these actions easily available to any object or Class that needs it.

DEPENDENCIES

Client:
1: Uses Manager A
Manager A:
3: Has references to handlers in subsystem
5: Delegates specific actions to Concrete Handler A, B
Handlers in subsystem
2: Have references to Manager A
4: Communicate to Manager A

RESULT

Centralization of communication between objects and centralized management of processes in your subsystem

BASICS

WHEN/WHAT?

Centralizing actions and Business Logic
The Manager Pattern centralizes actions on your system and the business logic that otherwise would be scattered over your sybsystem. It uses and addresses the subsystem in a similar way as the Façade. It can be accessed directly and mediate further actions like the Mediator.

Solve issues with Observer Pattern
Sometimes the use of the Observer Pattern can lead to a loss of control: who dispatched what and why? Managers help centralize the communication from different systems and create several hubs with specific scopes of events and dispatches.

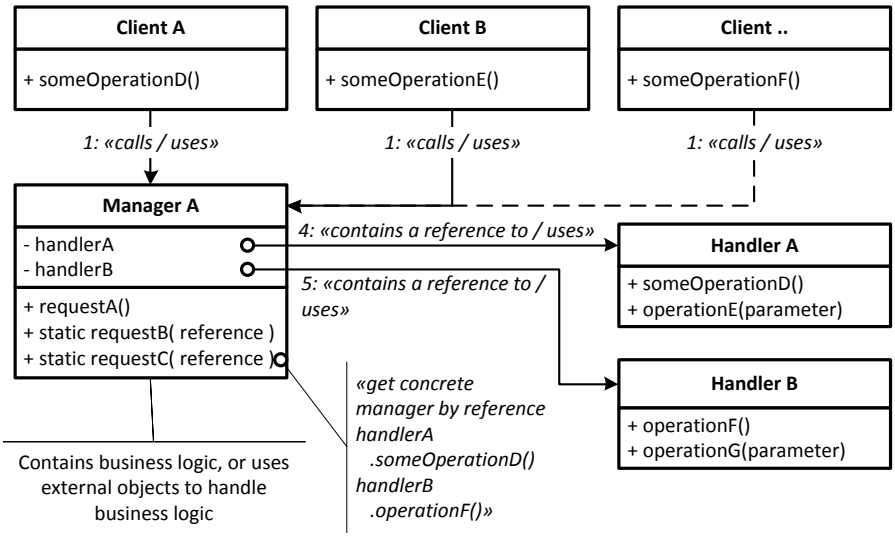
OTHER INFO

Façade and Mediator patterns combined
The manager can be seen as a merge of the Mediator and Façade pattern, combining the aspects of both Patterns to create an object that allows you for more control over your application.

The Business Logic: in the Manager or in separate classes?
The Business Logic can be put into separate classes as well within the Manager itself.

Singleton or Multiton and keys
The Manager can implement either the Singleton or Multiton pattern to offer a specific instance of the Manager. You address a specific instance using the static methods and keys that identify the specific manager.

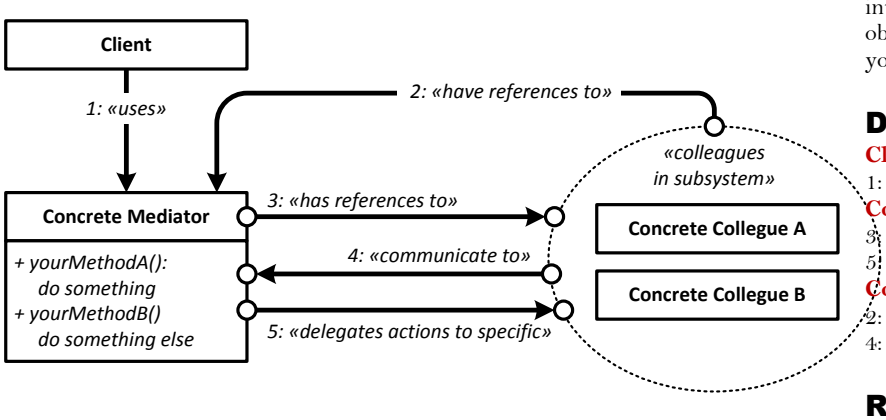
CLASS DIAGRAM



DESIGN PATTERNS #20

MEDIATOR

Link: Mediator



Visual summary of the Mediator Pattern

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.

DEPENDENCIES

Client:
1: Concrete Mediator
Concrete Mediator:
3: Has references to Concrete Colleagues in subsystem
5: Delegates specific actions to Concrete Collegues A, B
Concrete Collegues in subsystem
2: Have references to the Concrete Mediator
4: Communicate to the Concrete Mediator

RESULT

Centralization of communication between objects in your subsystem

BASICS

WHEN/WHAT?

Centralization of communication between objects
More than wrapping the Classes and objects it "wraps" the communication that takes place between them, by centralizing and managing the communication that is needed for these Processes within the Mediator

Hub or man in the middle
The Mediator acts as a hub or "Man in the Middle" for the Communication between separate objects and Subsystems

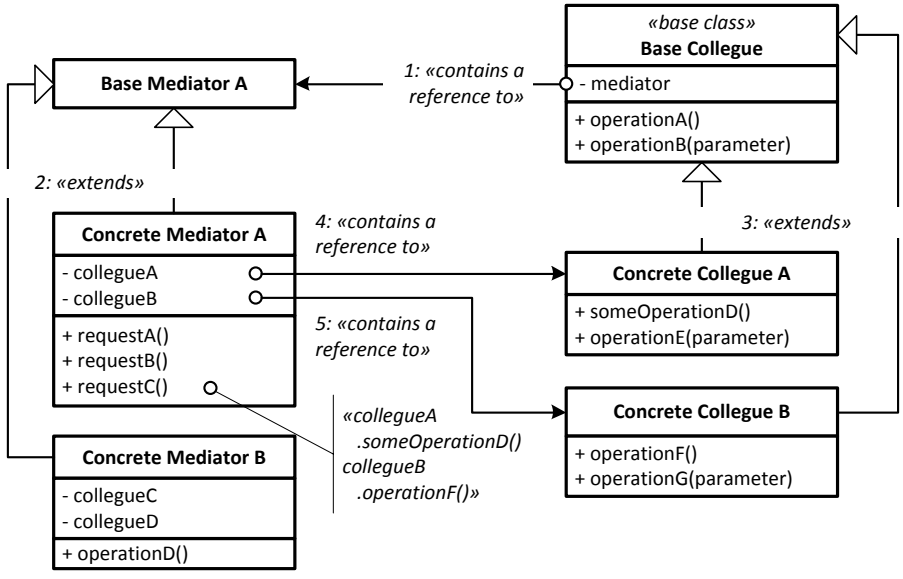
Abstracts and decouples the dependencies between objects in the subsystem. The Mediator allows objects in a Subsystem to communicate between each other, without the need for these objects to have knowledge of the other objects in the Subsystem.

OTHER INFO

Using Inversion of Control and Dependency Injection
The Mediator is another Pattern that applies the principles of Inversion of Control. Instead of objects deciding who should be created and who will be communicated to (the wiring), the Mediator takes over creation and the wiring and successively Injects itself into each player.

Can Inject itself into the subsystem
In one of the possible implementations of the Mediator, the Mediator can inject itself into the subsystem to be addressed directly by that Subsystem. The benefits of this injection is simplification of the code and your processes. De disadvantage is a tight coupling between the subsystem and the (Interface of) the Mediator that is Injected.

CLASS DIAGRAM



DESIGN PATTERNS #21

MULTITON *

Link: Multiton

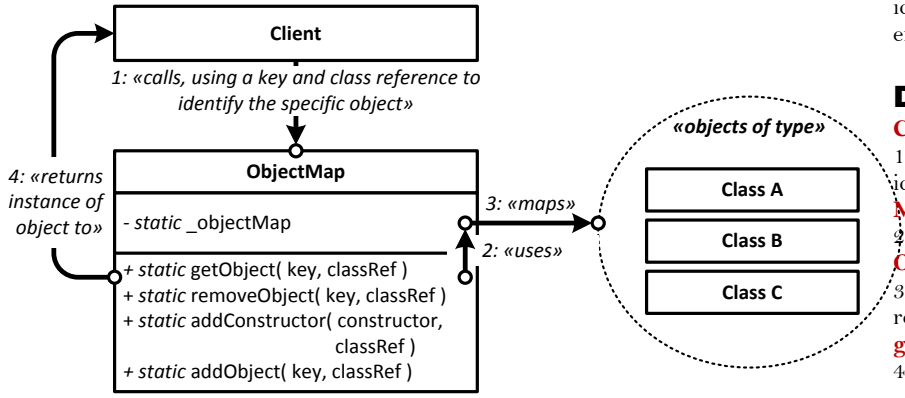
DESIGN PATTERNS #22

OBJECT MAP

Link: Object Map

INTENT

Use one single point of entry to store objects of any type in memory and retrieve them using a unique identifier. Assure that only one instance of an object exists for each specific data-item.



Visual summary of the Object Map Pattern

DEPENDENCIES

Client:
1: Calls ObjectMap using a key and class reference to identify the specific object
Methods in Object Map:
2: Uses objectMap to perform their actions
Object Map:
3: Maps objects of type Class A, B, C under class reference and identity key
getObject in Object Map
4: Returns an instance the object to the Client

RESULT

Mapping of one instance of a specific object of any type to be retrieved and used anywhere

BASICS

WHEN/WHAT?

Centralizing data management
The Object Map centralizes data management within your project. There is only one object and one class to address for all objects you might use in your project. When using the Object Map, you are sure that the object you request is the only one representing that specific entity with that specific ID.

Storing objects in memory using two keys
The Object Map store objects in memory using two keys: the Object Key (or ID) and a reference to the Class the object is derived from.

Retrieving, re-using objects and allowing for data-persistence
The main goal of the Object Map is to make it easier to store objects in such a way that you can retrieve them easily from anywhere. This promotes re-use of objects and makes it easier to create persistent objects with only one instance for data that comes from the outside.

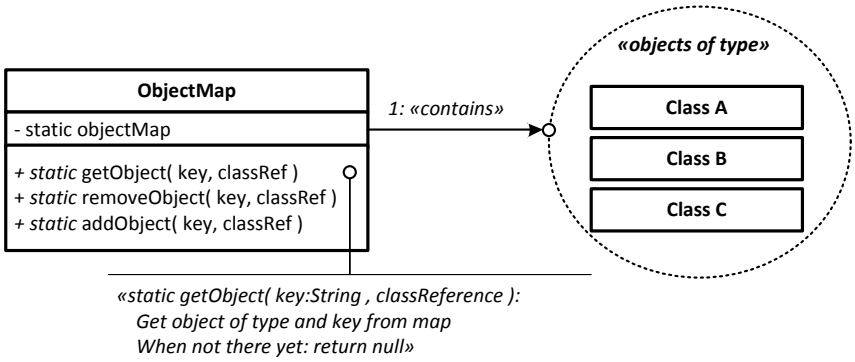
OTHER INFO

A pattern to de-couple dependencies
The Object Map has a second use, which is the de-coupling of dependencies from objects and within your structure. Instead of sending specific objects down a chain of objects, you can use abstract keys or identifiers(for instance based on Constants or identifiers in XML or JSON data) that refer to the specific objects you want and need.

Storing lists instead of objects
In some cases, instead of storing objects you might want to store lists of objects, which you can access and modify anywhere in your application.

Factory Object Map
The Factory Object Mpa is a specific version of the Object Map, taking care of obejct creation and mapping, instead of leaving this procures up to the Client, increasing the level of control on the process.

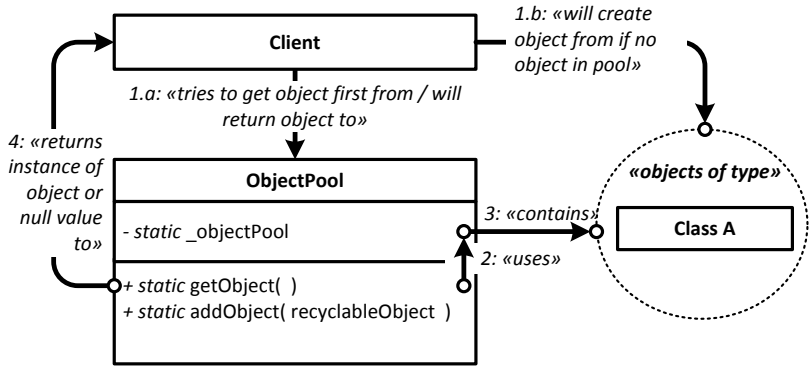
CLASS DIAGRAM



DESIGN PATTERNS #23

OBJECT POOL

Link: Object Pool



Visual summary of the Object Pool Pattern

INTENT

Manage the reuse of objects when a type of object is expensive to create or only a limited number of a kind of object can be created.

DEPENDENCIES

Client:
1.a: Will try to get object from the Object Pool / will return object to object Pool when no longer needed
1.b: Will create object from Class A is nothing is in Object Pool

getObject in Object Pool:
2: Uses object pool in static variable
4: Returns instance of object or null value to Client

Object Pool:
3: Contains objects of type Class A

RESULT

A container for objects which can be re-used after they have been discarded, leading to a reduction of new objects being created

BASICS

WHEN/WHAT?

When you want to reduce the amount of new objects created to a minimum
The Object Pool allows you to recycle and re-use objects which have already been created. This is possible by returning objects no longer in use into the Object Pool.

Releasing the garbage collector
The Object Pool can be seen as a solution to bypass the garbage collector. Instead of creating a lot of waste (your discarded objects) you simply recycle what you already have and only create new objects when your recycle-bin is empty.

Factory Object Pool: To centralize and manage Object Creation
Instead of leaving it up to your code, the Factory Object Pool takes complete care of the provision and management of object creation

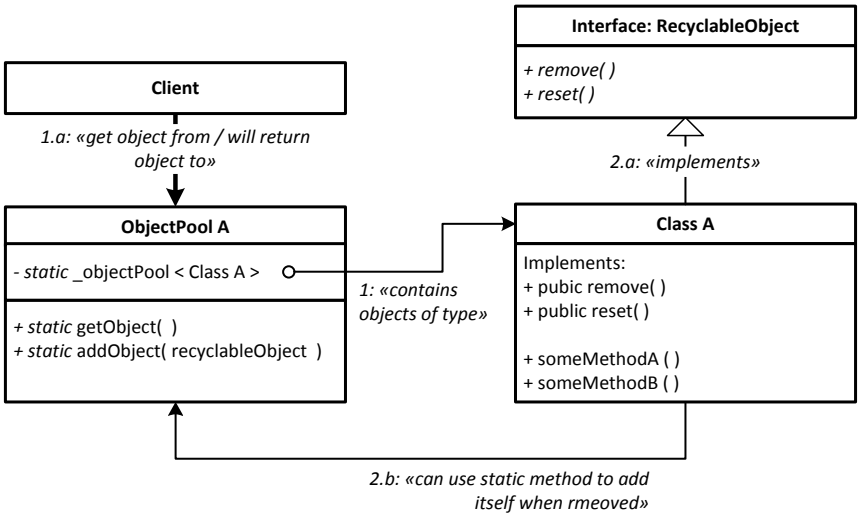
OTHER INFO

Implementing recycle interface
Like with the Object Pool it is recommendable for your objects to implement a recycle-interface and the code to reset the object and drop it into the Object Pool. This way, when you kill the object, it will put itself into the Object Pool. When and where you reset the object, so that it returns to a clean state without any values from previous sessions.

Different ways to handle empty pools
There are three different ways to deal with an empty Pool: return a null value (the **Simple Object Pool**), create and return a new object (I will refer to this as the **Factory Object Pool**) and blocking the client until an object becomes available from a different source (the **Halting State Object Pool**).

The gnarly issue of previous use
When an object is returned to the Object Pool, it has been used by other objects and possibly event listeners are registered to the object and by the object. References like this need to be broken to prevent unwanted behavior to happen. One way to ease this up is by using the Smart Reference Proxy.

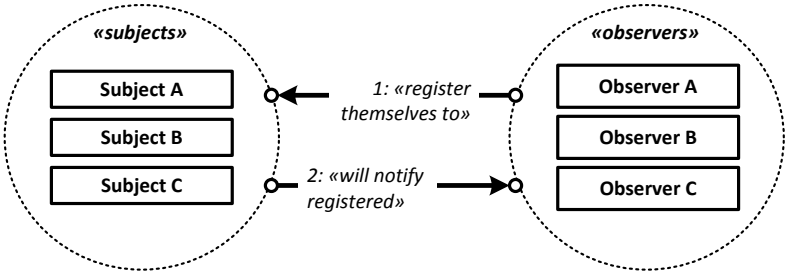
CLASS DIAGRAM



DESIGN PATTERNS #24

OBSERVER

Link: Observer



Visual summary of the Observer Pattern

INTENT

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

DEPENDENCIES

Observers:
1: Register themselves to one or more Subjects
Subjects:
2: Will notify registered Observers

RESULT

A more flexible way for objects to communicate with each other

BASICS

WHEN/WHAT?

One object, Multiple Observers
One object can be Observed by many Observers.

Decoupling of Dependencies
The Observer object (or Subject) does not have to have any relationship with the Observer, or has to know of the existence of an of the (possible) Observers that will "Observe" the changes or Events from the Subject

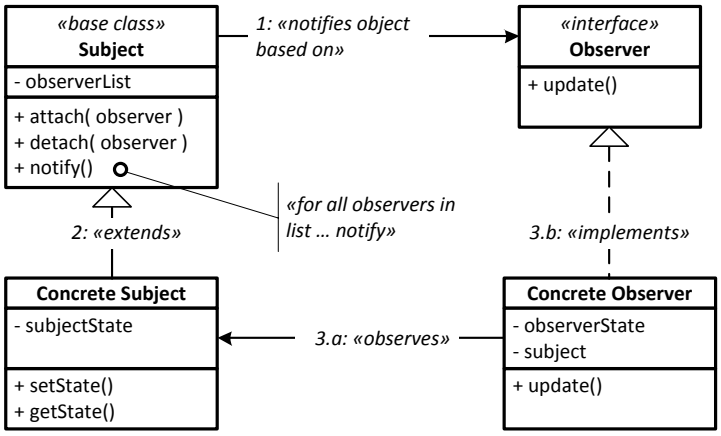
Events and Messages
The Observer Pattern can be used for two main types of notification: Events and Messages.

OTHER INFO

Messages?
As discussed in the book, Messages can represent anything, including Events, Requests, Updates and Instructions. Events are only dispatched when something happened. Events can not be used to send requests or instructions.

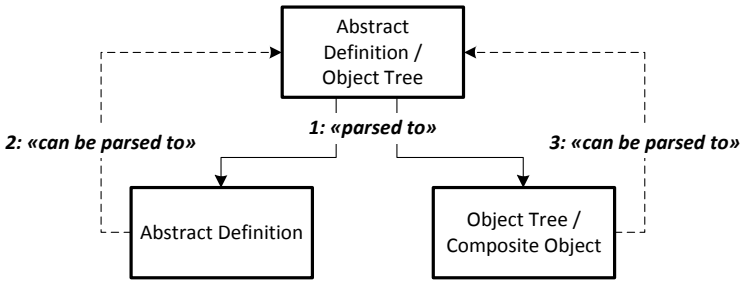
Event and Message Bus:
routing the Events and Messages through a specific bus
Instead of implementing the Obseerver Pattern directly on the object that is to be observed, an alternative approach can be to use and refer to an Event Bus. This Bus is an object to which Observers are bound and Events are dispatched.

CLASS DIAGRAM



DESIGN PATTERNS #25

PARSER



Visual summary of the Parser Pattern concept

Link: Parser

INTENT

To convert or parse one structure into another structure: containing similar, or the same, data.

DEPENDENCIES

Abstract Definition / Object Tree:
1: Can be parsed to Abstract Definition / Object Tree
Abstract definition:
2: Can be parsed to Abstract definition / Object Tree
Object Tree / Composite object:
3: Can be parsed to Abstract definition / Object Tree

RESULT

Any structure can be transformed into any other structure

BASICS

WHEN/WHAT?

To parse/convert one structure into another
The Parser can be used to parse or convert one structure into another. This can be XML to objects or objects to XML or from one abstract definition into another; for instance: XML to JSON and the other way around. This can also mean one object structure into another object structure, representing the same thing in a different way. Think for instance of datagrams being parsed to visual elements on screen.

To read and use the contents of a structure
Any structure contains data, and sometimes you need specific elements from that data to be used or presented somewhere else. The Parser can traverse the structure and distill those elements and pieces of information you need.

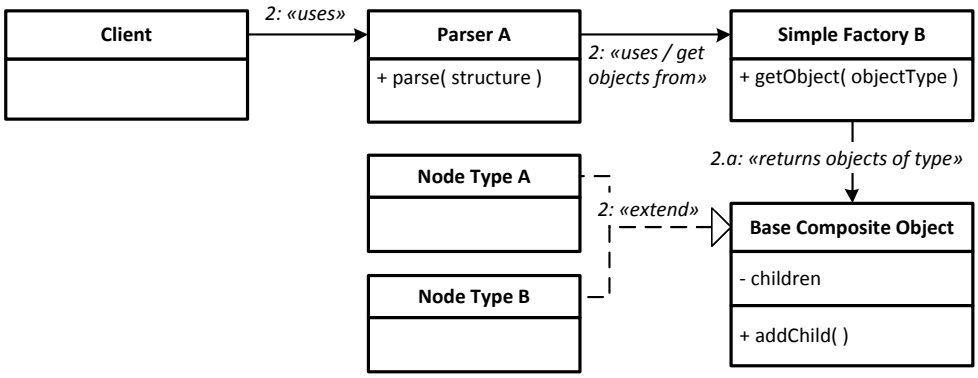
OTHER INFO

Any direction
The Parser is by default a Pattern that can Parse anything to anything as long as it is available and possible. So you can Parse one Abstract Definition to another (XML to JSON, XML to a Custom Structure, XML to a Custom Structure to an Object Tree, XML to an Object Tree, back to XML).

Results can be fed back to a Parser
In principle, a Parser result can be Parsed into yet another structure. An object structure can be parsed into an Abstract Definition and then into another Abstract Definition and then into an object Tree.

Lossful and lossless parsing
Depending on your need and use, the parsing process can be either lossful and lossless. Lossful means that from everything in A, only a part will arrive in B. When you take B, structure A can never be recovered entirely. Lossless means that there is no loss of information between A and B. A can be converted to B and B back to A.

CLASS DIAGRAM



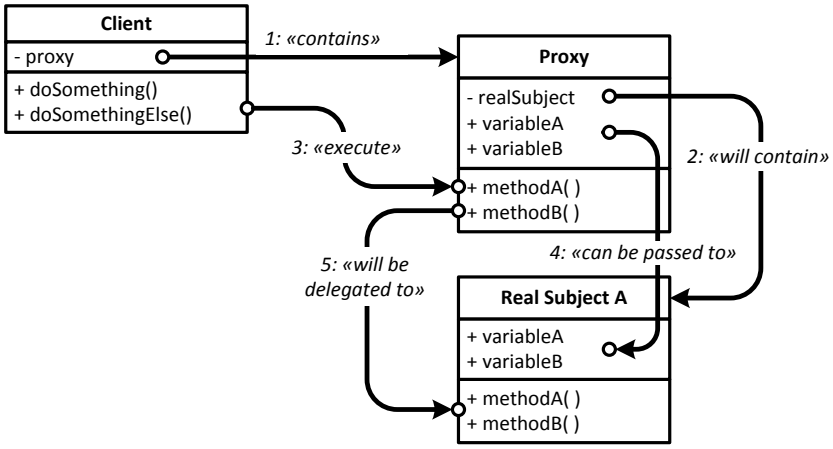
DESIGN PATTERNS #26

PROTOTYPE *

Link: Prototype

DESIGN PATTERNS #27

PROXY



Visual summary of the Proxy Pattern

Link: Proxy

INTENT

Provide a surrogate or placeholder for another object to control access to it.

DEPENDENCIES

Client:
1: Contains Proxy
RealSubject in Client:
2: Will contain Real Subhect A
Methods in Client:
3: Execute methods on Proxy
Variables in Proxy:
4: Can be passed to variables in Real Subject A
Methods calls on Proxy:
5: Will be delegated to Real Subject A

RESULT

A separation between the user and the real object via a "man in the middle", allowing you to control access to it

BASICS

WHEN/WHAT?

Representing something that might not be there yet
The object we represent with the Proxy is usually not yet there when we instantiate the Proxy.

Can be used immediately, even if the object is not there
With the Proxy you do not have to wait for the Actual object to arrive. Whatever you want to do with the object can be done immediately on the Proxy.

Different types
Remote Proxy: representing an object from somewhere else
Virtual Proxy: creating the object when needed
Protection Proxy: enveloping and protecting the original object
Smart Reference: man in the middle when accessing an object

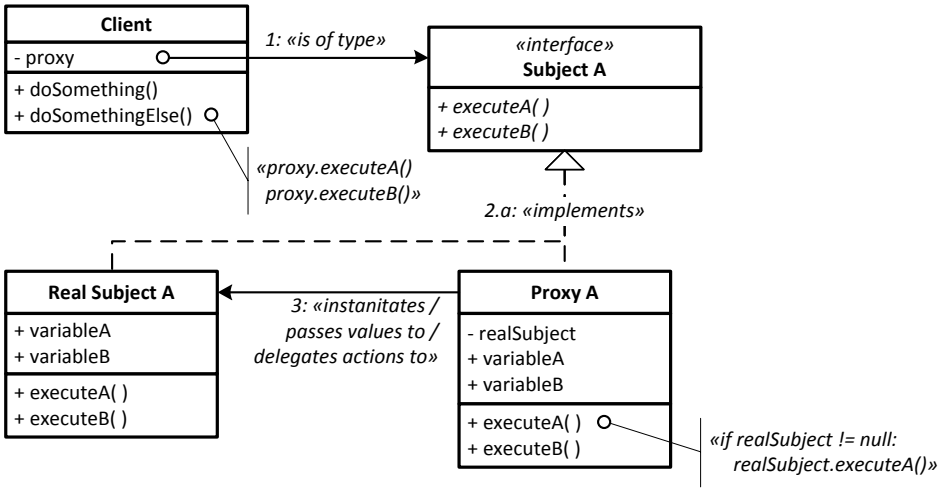
OTHER INFO

Pretends to be, with additional features
Like the Adapter and the Bridge, the Proxy Pretends to be the object it represents. But where Bridge and Adapter rely on the actual object to be there, for the Proxy it does not matter if the object it represents arrives later or not at all

Buffering the values and method calls
The Proxy – in most cases – will act as a buffer for all the method calls and value settings on the actual object. While the actual object is not there, the Proxy will store these settings and queue the calls internally.

Passes all your requests when the object is there
Once the object arrives, the Proxy passes all your requests and executes your settings on the object (B)

CLASS DIAGRAM



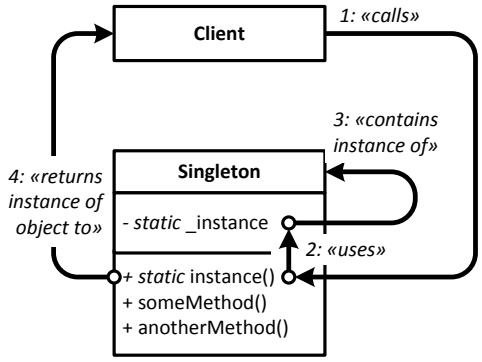
DESIGN PATTERNS #28

REFLECTION *

Link: Reflection

DESIGN PATTERNS #29

SINGLETON



Visual summary of the Singleton Pattern

Link: Singleton

INTENT

Provide a surrogate or placeholder for another object to control access to it.

DEPENDENCIES

Client:
1: Calls static method instance()
Static Methods instance():
2: Uses the instance stored in the static variable _instance or created new instance when not there.
Static method :
3: Execute methods on Proxy
Variables in Proxy:
4: Can be passed to variables in Real Subject A
Methods calls on Proxy:
5: Will be delegated to Real Subject A

RESULT

One single instance of an object that can be retrieved and used anywhere

BASICS

WHEN/WHAT?

Provide one single instance of an object
The Singleton is used to provide one single instance of an object throughout your project.

Replacement for global variables
One use for the Singleton is as a replacement for global variables. The basic idea is that you want to have one single central place to store specific values and references to other objects in such a way that you can reach this from all over your application.

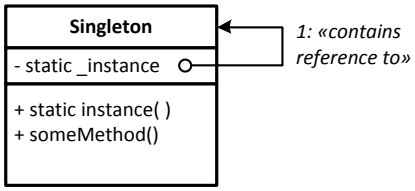
For parts of the system that require only one instance
Some parts of your system require one and only one instance of an object as this instance should be the only one dealing with that subject and information. In games this can be your game-score and player health. In an OS this can be the file system or a printer spooler.

OTHER INFO

The Multiton, Identity Map and Object Map: taking it one step further
The Multiton, Identity Map and Object Map do roughly the same as the Singleton: providing only one instance, but then per entity. For instance: in a multiplayer game, you have "player 1" and "player 2" and each need their own object and representative.

Unit testing
The Singleton is kind of "suspect" in unit testing as "you do not control the instantiation". Nothing prevents you, however, from injecting a controlled version of the singleton instance into the singleton class.

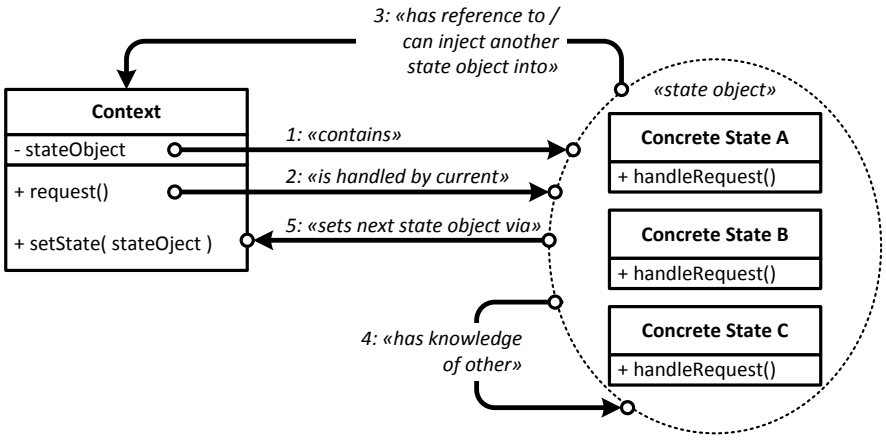
CLASS DIAGRAM



DESIGN PATTERNS #30

STATE

Link: State



Visual summary of the State Pattern

INTENT

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

DEPENDENCIES

State Object in Context:
1: Contains Sate Object
Request in Context:
2: Is handled by State Object
State Object :
3: Has reference to / can inject another object into Context
4: Has knowledge of other State Objects
5: Sets next State Object via setState mehtod

RESULT

A self-organizing delegator that defines internally which concrete implementation should handle the next state of the process

BASICS

WHAT/WHEN?

When you need a self-organizing delegator to handle processes
Due to its setup, the State can be seen as a self-organizing Delegator. The Context delegates actions to the State object. The State object then defines which next State object will deal with the process state that follows.

The State object changes the content of the State container
The State Pattern is a closed universe. Your code, using the State Pattern, has no knowledge on what State should or could be next. This is all dealt with by the State objects themselves.

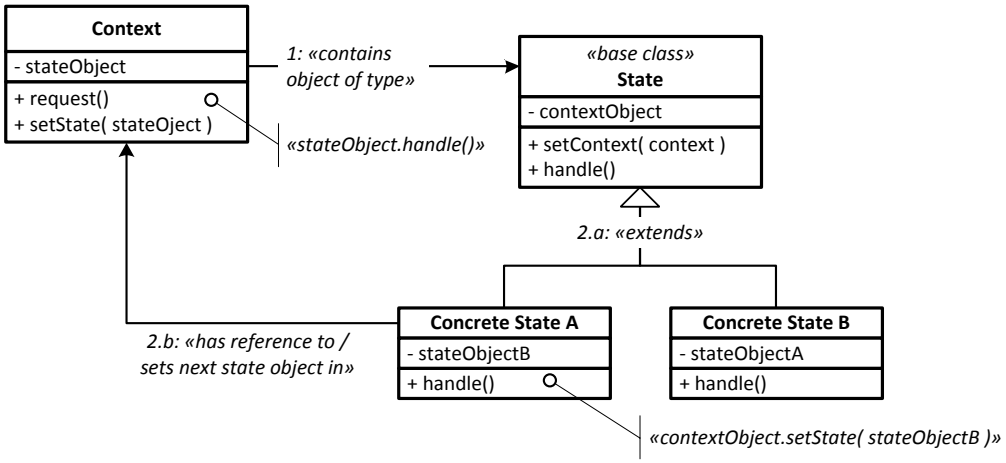
OTHER INFO

Each state object knows what next state will follow
Each state object knows what next state will follow on a specific method call.

Actions in the Context are delegated to the State object
Most if not all actions in the Context are delegated to the State object.

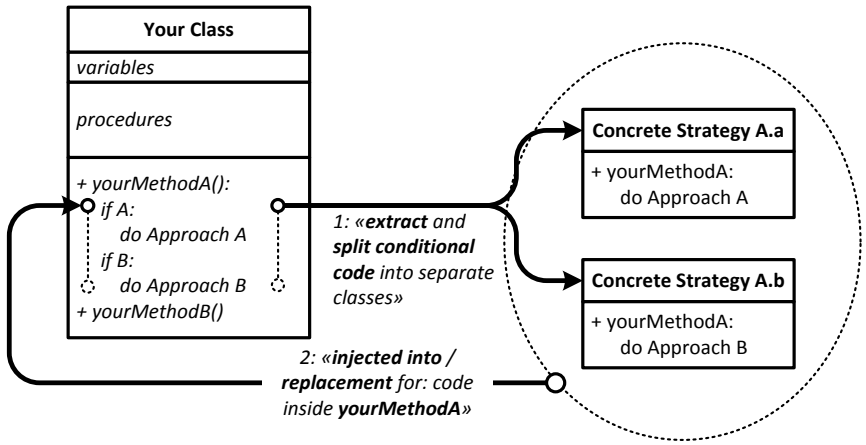
State objects have knowledge of other State objects
State objects have knowledge of other State objects and which State object to choose when the State changes due to a method call.

CLASS DIAGRAM



DESIGN PATTERNS #31

STRATEGY



Visual summary of the Strategy Pattern

Link: Strategy

INTENT

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

DEPENDENCIES

Actions in Your Class:
1: Extracted and split into separate classes
Strategies:
2: Injected into / replacement for code inside your method

RESULT

Classes and Objects that can change part of their behavior

BASICS

WHAT/WHEN?

When only a part of the process changes in a specific Context
The Strategy Pattern is used when only a part of the process in your Class changes in a specific Context.

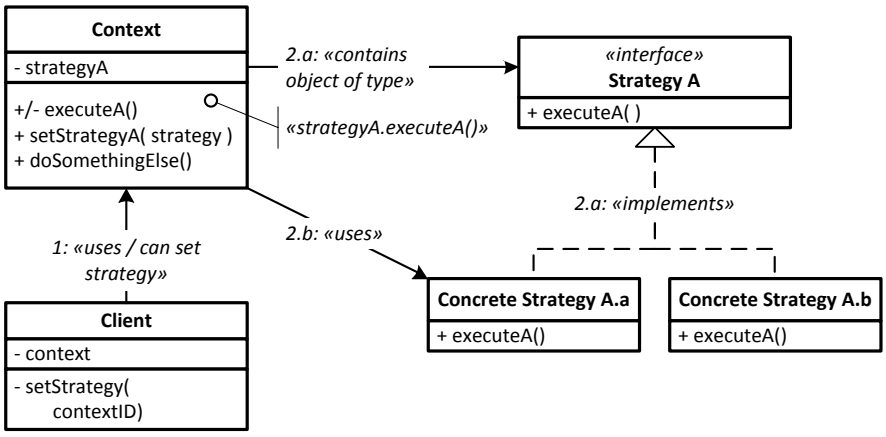
When you want to extract Context specific code into a separate Class
To cater a more dynamic implementation of specific routines, only the context-specific code is externalized (extracted) into a Strategy Class. Depending on the Context of the situation, a different Strategy can be injected into the Context object by your code, leading to a different execution of specific actions.

A handy alternative for conditional execution in your code
Instead of cluttering your methods with conditions, you can simply extract that conditional code into separate Classes, select the one you need and delegate the concrete execution to it.

OTHER INFO

Delegating actions
Like the Bridge and the Delegate Pattern, the Strategy Pattern delegates actions to another object. In the case of Strategy, this is the Strategy Object or concrete implementation of the Strategy per Context.

CLASS DIAGRAM



DESIGN PATTERNS #32

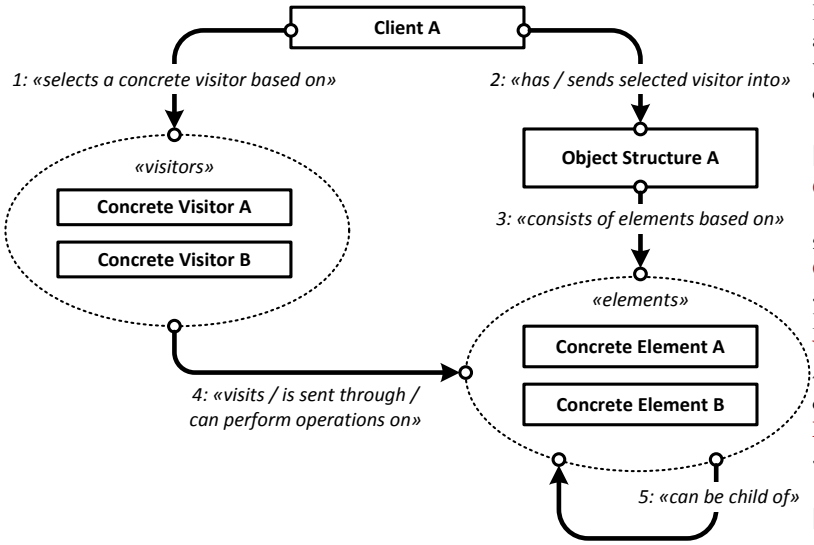
TEMPLATE METHOD *

Link: Template Method

DESIGN PATTERNS #33

VISITOR

Link: Visitor



Visual summary of the Strategy Pattern

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.

DEPENDENCIES

Client A:
1: Selects a concrete visitor A, B
2: Has / sends selected visitor into Object Structure A
Object Structure A:
3: Consists of (composite) elements bases on Concrete Element A, B
Visitors:
4: Visits / is sent through / can perform operation on elements in Object Structure A
Elements in Object Structure A:
5: Can be child of other elements

RESULT

A dynamic solution to run through- and perform actions on an Object Structure from the inside using different type of "visitors"

BASICS

WHAT/WHEN?

When you need to traverse through a composite object
The Visitor can be used to travers through a Composite object and either do something with the object,s or simply read the contents.

As an alternative for the Parser
You could use the Visitor pattern as an alternative for the Parser as both can accomplish the same results. The Visitor requires a bit more within the objects you traverse.

Many different Visitors, one traversing process
The benefit of the Visitor is that you can use many different Visitors in the same traversing process, each delivering a completely different result in that process.

OTHER INFO

Can require specific setup of the objects in the tree
In the basic implementation of the Visitor Pattern, the objects pass the Visitor. This includes an "accept" method on each object: accepting the visitor.

Visitor can pass itself to the next object
There are workarounds possible in which the Visitor can pass itself to each next object and deal with the content there. One workaround is to use an Object Adapter for the objects which are not implementing the Visitor Pattern.

Inverted approach related to the Parser
Like the Command and Observer patterns, the Visitor and Parser are two different approaches to the same problem: how do I work with data and objects in a structure? The Visitor inverts the process by traversing from the inside

CLASS DIAGRAM