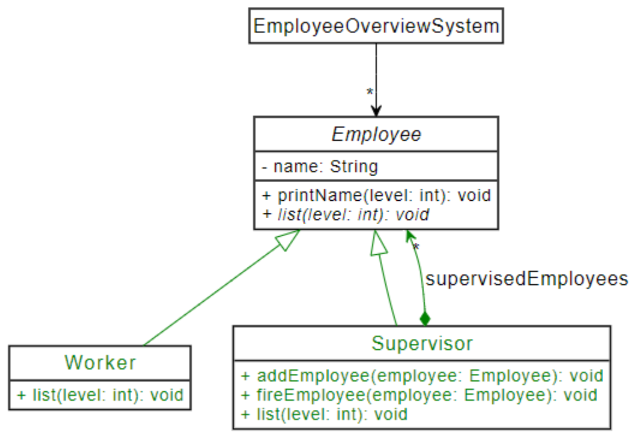
# Patterns EIST2021

Einführung in die Softwaretechnik (IN0006) (Technische Universität München)

## Composite Pattern:

– composes objects into tree structures and then work with them as if they were individual



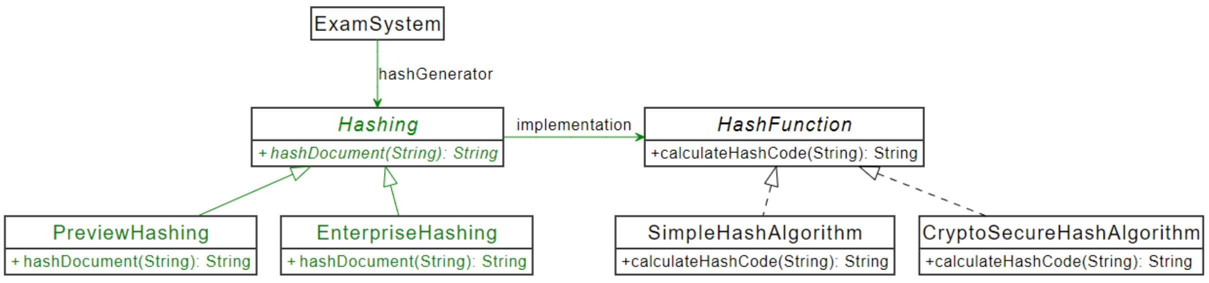
EmployeeOverviewSystem → Client
Employee → Component
Worker → Leaf 1
Supervisor → Leaf 2
`list()` → `operation()`

A Client has Components which it uses by calling their methods. Sometimes Components can be summarized into one superclass from which they then can inherit. However, the Client doesn't care which type of subcomponent (Leaf of the tree structure which cannot have children or a Composite which is an inner node which can have children) he calls. The right subcomponent is decided during runtime and the best suited one is being chosen.
In the example above the EmployeeOverviewSystem calls the methods of the Employee superclass. If the list method is called, the type of the subclass is needed because the execution might differ.

## Bridge Pattern:

– organizes set of closely related classes into two separate hierarchies (split up into abstraction and implementation)

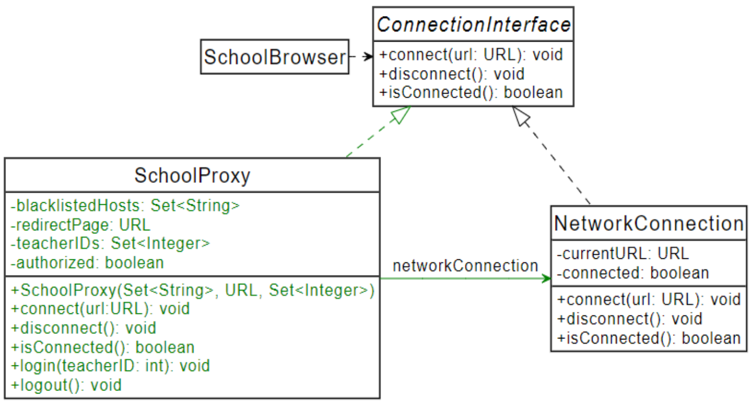
ExamSystem → Client
Hashing → Abstraction
HashFunction → Implementor
PreviewHashing, EnterpriseHashing → Refined Abstraction
SimpleHashAlgorithm, CryptoSecureHashing → Concrete Implementor
`hashDocument()` → `operation()`
`calculateHashCode()` → `operationImpl()`


ExamSystem
hashGenerator
Hashing
+ hashDocument(String): String
implementation
HashFunction
+calculateHashCode(String): String
PreviewHashing
+ hashDocument(String): String
EnterpriseHashing
+ hashDocument(String): String
SimpleHashAlgorithm
+calculateHashCode(String): String
CryptoSecureHashAlgorithm
+calculateHashCode(String): String

A Client first uses an Abstraction of the real Function/Algorithm. The Abstraction then calls the suitable method of the real Function/Algorithm of the Implementor.
The Abstraction is split up in different types (Refined Abstraction) which then call different implementations (Concrete Implementor) of the Implementor.

## Proxy Pattern:

- provides a substitute/placeholder (Proxy) for another object (Real Subject)
- The Proxy provides and controls the access to the Real Subject
- Allows to differentiate whether a request went through to the Real Subject or not



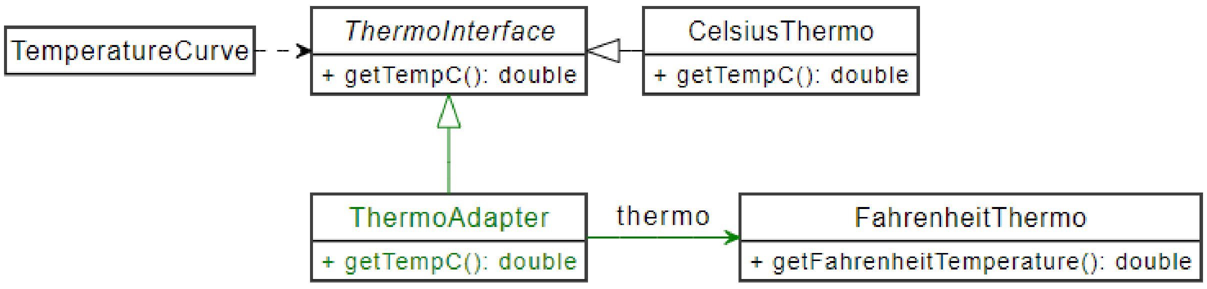
SchoolBrowser → Client
ConnectionInterface → Subject
SchoolProxy → Proxy
NetworkConnection → Real Subject
`connect(), disconnect(), isConnected()` → `request()`

A Client first calls the method he/she needs on an Interface (Subject) which provides the needed methods. Then, during runtime, the Proxy is being called first, which then delegates to the implementation of the same method of Real Subject after (here e.g., checking if the User is authorized or not).

## Adapter Pattern:

- provides an adaption to a functionality which already exist to implement a new functionality

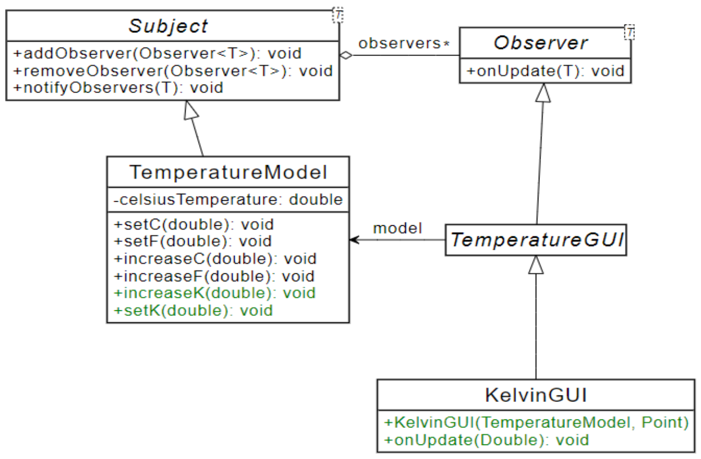
TemperatureCurve → Client
ThermoInterface → Client Interface
ThermoAdapter → Adapter
FahrenheitThermo → Existing Object
CelsiusThermo → New Object



A Client calls a Client Interface which, however, only provides one method type which cannot be applied to every object that is similar to the Existing Object. However, that isn't necessary because one can implement an Adapter which suitably adapts to the wished function – the New Object.

## Observer Pattern:

- especially important in combination with the MVC
- provides an Observer which observes when a Subject which frequently changes its state, does change its state



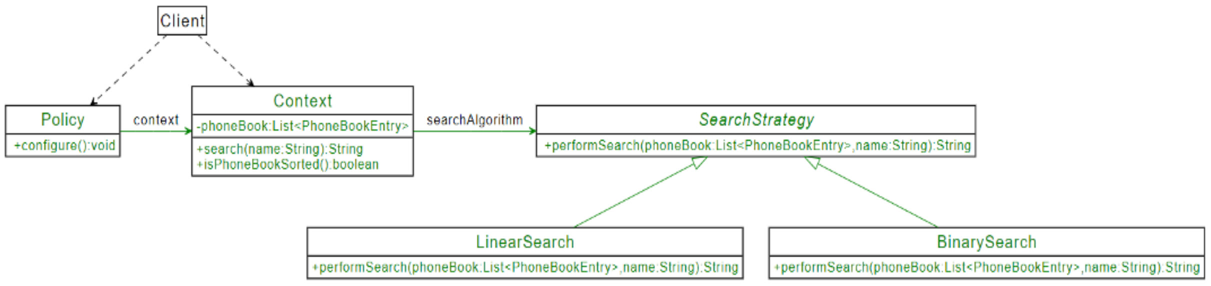
Subject → Subject (to observe)
Observer → Observer (which observes)
TempertrureModel → Concrete Subject
TemperatureGUI → Concrete Observer
`onUpdate()` → `update()`
`notifyObservers()` → `notify()`
celsiusTemperature → state

A Subject which changes its state regularly is being observed by an Observer. The Subject and Observer are usually abstract classes. Concrete Subject and a Concrete Observer provide the implementation of Subject and Observer.
The Concrete Subject usually has some kind of state which the Observers are interested in to observe. Usually, the Observer is for example a GUI that needs to be updated to always show the right sates of the Subjects it observes.
**Notification and Pull:** Every time state of the Subject changes, `notify()` is called which calls `update()` on the Observers (they can decide to pull the state)
**Notification and Push:** Subject includes the state that has been changed in each `update()` → `update(state)`
**Periodic Pull:** An Observer periodically pulls state of the Subject (e.g., every five seconds)

## Strategy Pattern:

- allows choosing between different strategies/algorithms based on a policy

Client → Client
Policy → Policy
Context → Context
SearchStrategy → Strategy
LinearSearch, BinarySearch → Concrete Strategy


Client
Policy
+configure():void
context
Context
-phoneBook:List<PhoneBookEntry>
+search(name:String):String
+isPhoneBookSorted():boolean
searchAlgorithm
SearchStrategy
+performSearch(phoneBook:List<PhoneBookEntry>,name:String):String
LinearSearch
+performSearch(phoneBook:List<PhoneBookEntry>,name:String):String
BinarySearch
+performSearch(phoneBook:List<PhoneBookEntry>,name:String):String

A Client needs to first configure the Policy which decides what Strategy is being used at runtime. Having decided what Policy exists the Context object is being called and the Policy is used to determine which Concrete Strategy suits the Context best.