TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

WS 2019/20 – Design Patterns and Frameworks

Extensibility Patterns: Extension Access

Professor: Prof. Dr. Uwe Aßmann
Lecturer: Dr.-Ing. Sebastian Götz
Tutor: Dr. rer. nat. Marvin Triebel

Task 1 Extensible Insurance Contracts

Insurance contracts are documents that have longevity and are treated by many different people for many different reasons during their life-time. In a software system for the management of insurance contracts, each of these clients needs a custom interface to the insurance contract object. Sometimes the same person needs to treat very different types of insurance contracts in the same manner. Occasionally, new types of treatment need to be added dynamically, without affecting any of the preexisting code.

a) To cope with these issues the `Extension Object` pattern [2] can be employed. What are its participants? How do they collaborate to support solving the above problems?

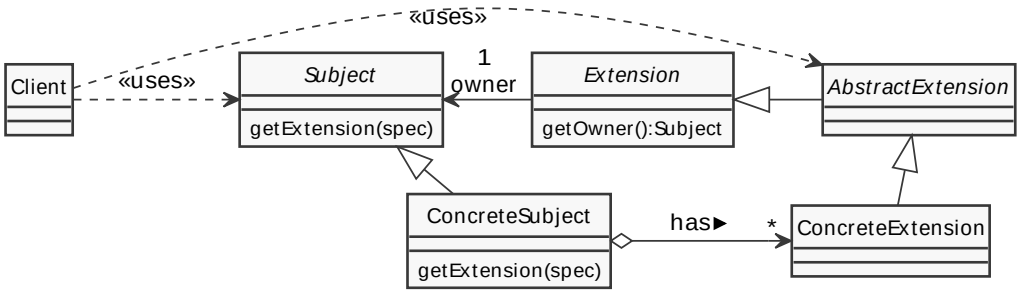
Solution: The participants of the `Extension Object` are as follows [2]:

- **Subject** defines the identity of an abstraction. It declares the interface to query whether an object has a specified extension,e.g., by means of a string.
- **Extension** the base class for all extensions. It defines some support for managing extensions themselves. Extension knows its owning subject.
- **ConcreteSubject** implement the `getExtension` operation to return a corresponding extension object when the client asks for it.
- **AbstractExtension** declares the interface for a specific extension.
- **ConcreteExtension** implement the extension interface for a particular component. Store the state associated with a specific extension.

Similarly, according to [2], the collaborations below are established:

- A client asks a `Subject` for a specific `Extension`.
- When the extension exists the `Subject` returns a corresponding extension object. The client subsequently uses the extension object to access additional functionality.
- If the `Subject` does not support an extension it returns null to signal that it does not support the extension.

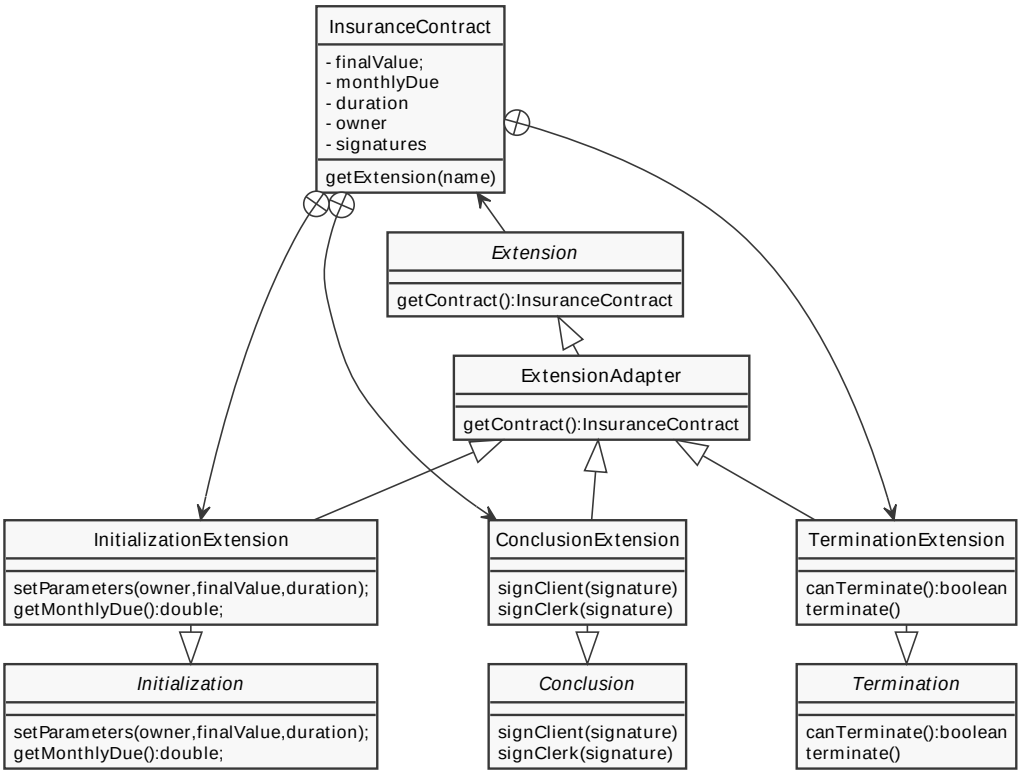
The general structure of the `Extension Object` pattern is illustrated below.



b) Use the `Extension Object` pattern to design and implement an insurance contract management system. Support the following phases:

- *Initialization*: The contract object has just been created and needs to be filled with the correct data.
- *Conclusion*: The contract has been accepted and needs to be signed by all parties.
- *Termination*: The contract's duration has passed and all the money goes to the company.

Solution: The following class diagram illustrates the implementation of our `Insurance Contract` and the three `Extensions` for each of the phases:



c) What would be needed, to support the phase *Incident*, i.e., to handle the occurrence of an insurance case covered by the insurance?

Solution: The only elements to implement are a new `Incident` interface with the necessary method signatures and the implementation of the corresponding functionality in a subclass of the `ExtensionAdapter` (within the `InsuranceContract` class). As soon as this `IncidentExtension` has been registered with the `InsuranceContract` object, the new extension can be used.

d) How can different document types, for example, insurance contracts and insurance letters, support the same phases?

Solution: By providing an interface `IExtensible`, which provides access to extensions through a `getExtensions` method. If extensions are declared with a global unique identifier, any object that implements this interface can then be extended in the same way, independent of its class. This even permits virtual dynamic class changes.

> *An implementation of this is in Microsoft's most basic COM interface `IUnknown`. This interface provides two features, i.e., reference counting and access to other interfaces of a COM object using globally unique interface identifiers.*

Task 2 Homework (optional)

The homework assigns you to try implement role-based design patterns and apply them to the file system scenario from the previous exercise.

To implement role-based design patterns use one of the more advanced design patterns from the lecture, e.g., the `Extension Object Pattern` [2], the `Role Object Pattern` [1], or the `GenVoca Pattern` [3].

a) Implement the `Observer` design pattern based on roles.

b) Implement the `Composite` design pattern based on roles.

References

[1] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer, 1998.

[2] Erich Gamma. Extension object. In *Pattern languages of program design 3*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., 1997. URL https://klevas.mif.vu.lt/~plukas/resources/Extension%20Objects/ExtensionObjectsPattern%20Gamma96.pdf.

[3] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570. Springer, 1998. URL citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.7182&rep=rep1&type=pdf.