**Design Patterns and Frameworks**
Dipl.-Inf. Steffen Zschaler
INF 2097
`http://st.inf.tu-dresden.de/teaching/dpf`

**Exercise Sheet No. 4**
Software Engineering Group
Institute for Software and Multimedia Technology
Department of Computer Science
Technische Universität Dresden
01062 Dresden

# Extensibility Patterns

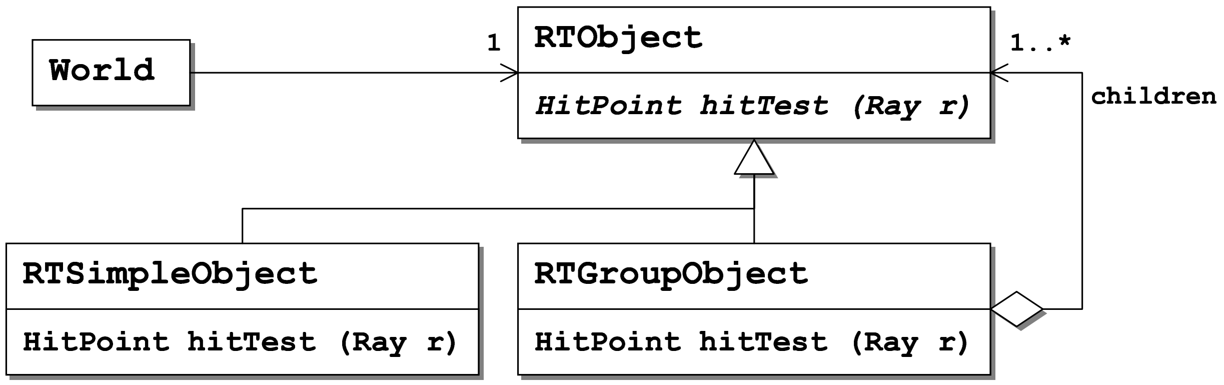
## Task 1: Raytracer Objects

Raytracing is a technology for generating photo-realistic images from scene description data. Very advanced types of this technology have recently been used for creating quite a few, and very impressive, animated pictures (such as "Shrek", or "Nemo").

Imagine you have to realise a raytracer application. At the core of such an application is a data structure which maintains and represents the scene graph or the "world". Objects can be very simple objects which have a geometrically defined shape and a typically algorithmically defined material, but they can also be arbitrarily complex compositions of other objects. The scene graph is then repeatedly traversed to check for intersections with various optical rays.

1a) **Task:** Design a class structure which can represent a scene graph. What design pattern is applicable?

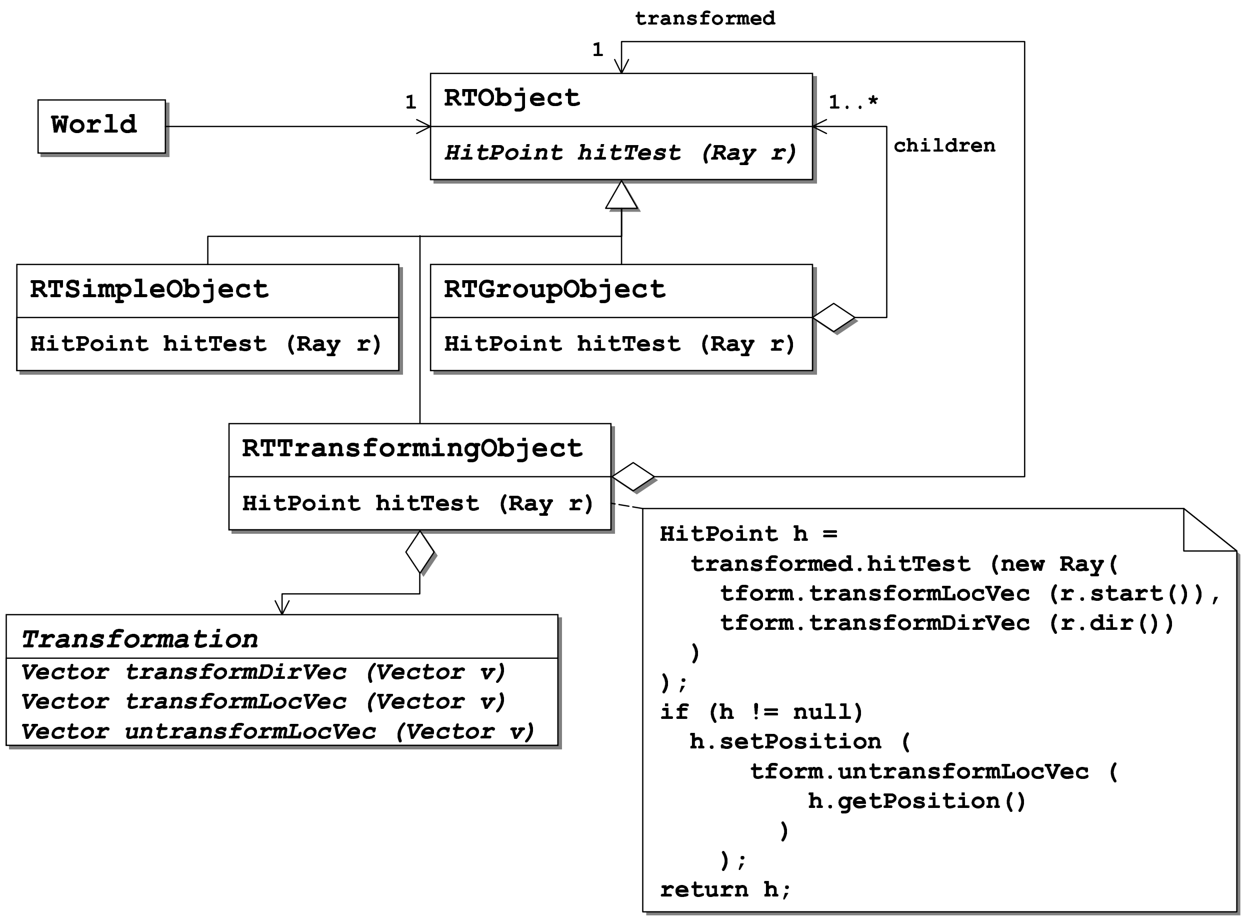
**Solution:** We use COMPOSITE to represent the arbitrary grouping of objects into a scene graph hierarchy. All classes have an operation `public HitPoint hitTest (Ray r)` which is abstract in `RTObject` and is implemented in the subclasses.



1b) **Task:** It is often useful to be able to transform existing objects in a scene graph—for example, by moving, scaling or rotating them. This should be possible for simple and complex objects alike. Enhance the class structure from the previous subtask to enable transformation of objects. Which design pattern do you use?

**Hint** A standard raytracing trick is to transform the ray vectors (a start location and a direction) instead of the objects. This is mathematically equivalent and makes life much easier for most situations.

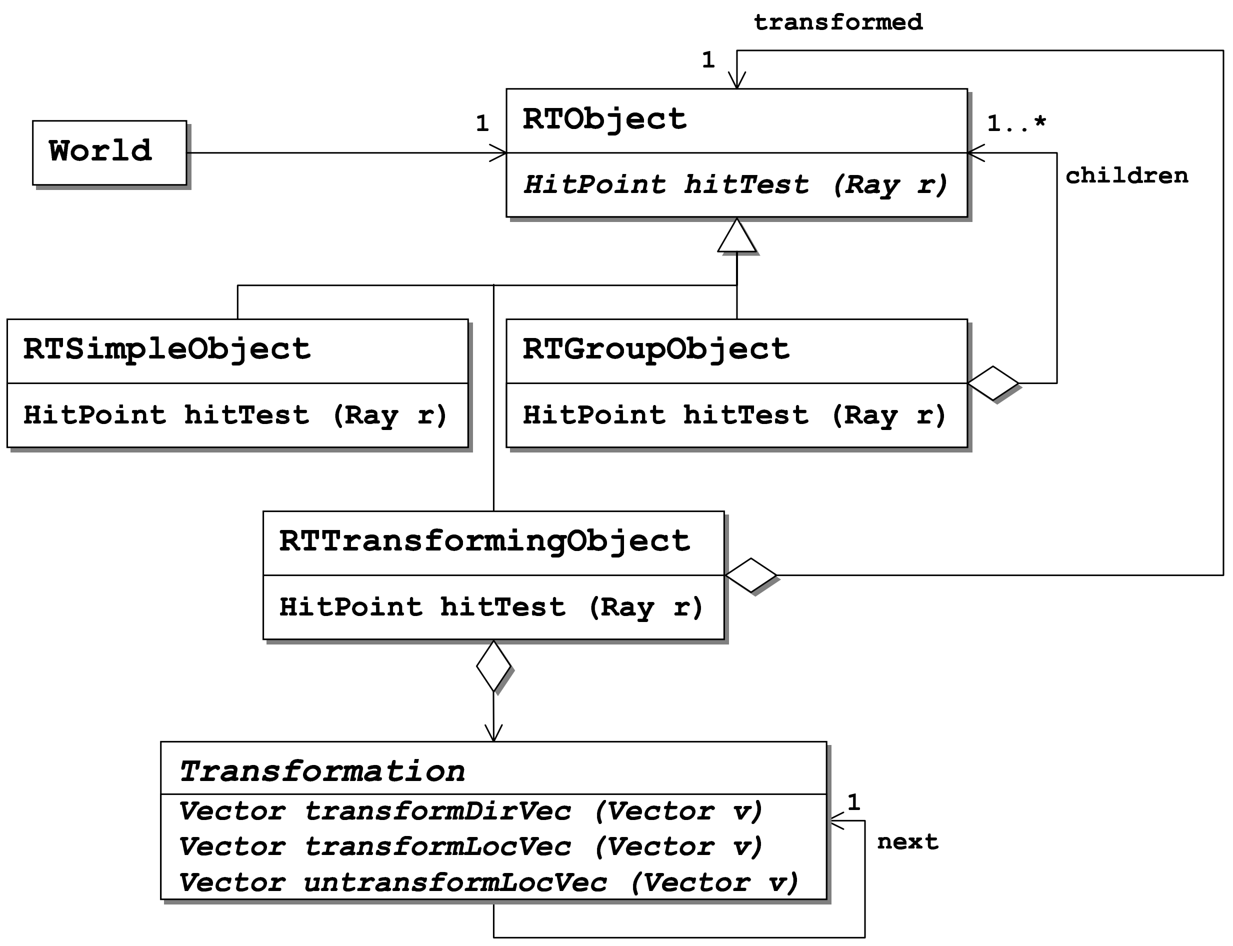
**Solution:** We add a Decorator `RTTransformingObject` which transforms the ray coordinates. In our solution we even used an additional pattern by reifying the transformations as a Strategy.



```
HitPoint h =
  transformed.hitTest (new Ray(
    tform.transformLocVec (r.start()),
    tform.transformDirVec (r.dir())
  )
);
if (h != null)
  h.setPosition (
      tform.untransformLocVec (
          h.getPosition()
        )
    );
return h;
```

$\boxed{\text{1c)}}$ **Task:** When transforming an object it is typical to construct the transformation from the basic transformation scaling, rotation, and translocation by putting them in a sequence of transformation operations. Extend the class structure to represent such sequences of transformations. Discuss different solutions.

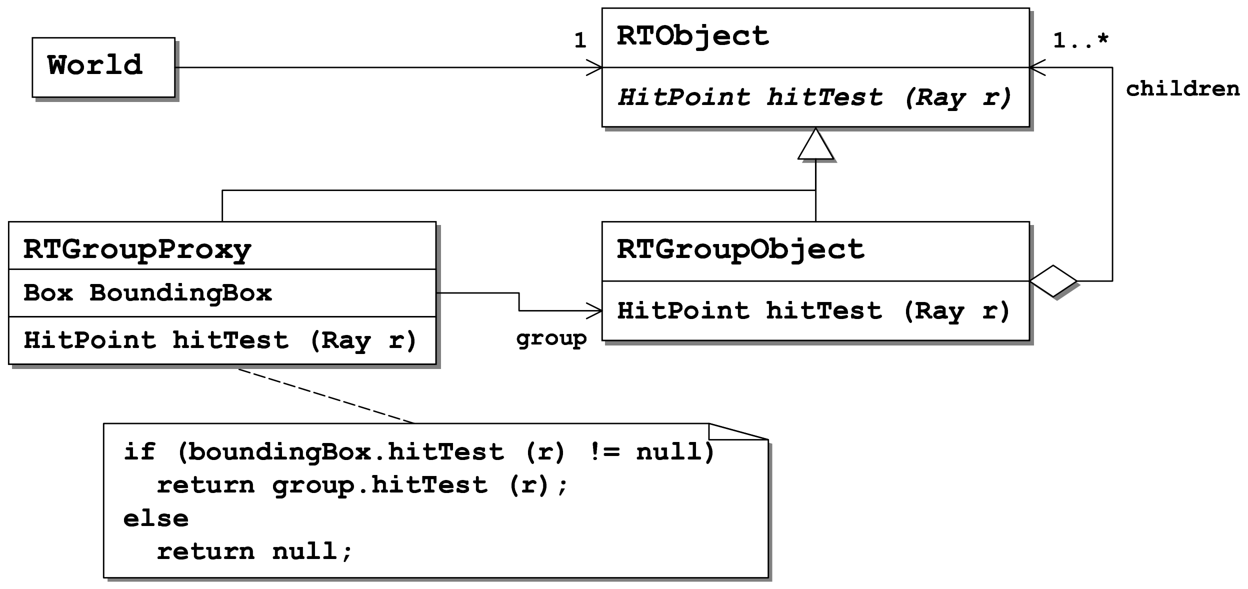
**Solution:** Chaining transformations can already be done using the design from the last subtask and simply wrapping multiple `RTTransformingObject` instances around the actual object to be transformed. A maybe more efficient way to go about it is to use a Chain of Responsibility of the `Transformations`:



If we change this slightly by having the `RTTransformingObject` invoke each elementary transformation in the sequence instead of passing the call from one transformation to the next, we can even pre-process the transformation chain and optimise its performance by condensing it into a single transformation.

1d) **Task:** When tracing complex objects it is often worthwhile to first check whether the ray will ever intersect their bounding box (i.e., the smallest box completely enclosing the object). Enhance the class structure to add this functionality. What design pattern could you use? Are there other ways of doing it?

**Solution:** We use a PROXY `RTGroupProxy` which gets the bounding box and checks against that before allowing the actual hit point to be computed. We could also have included the functionality directly into the `RTGroupObject`, but that would have mixed it with the core functionality (while keeping memory footprint slightly lower). Another possibility would have been to use a DECORATOR. This would have made it possible to apply the optimisation also to transformed objects.



```
if (boundingBox.hitTest (r) != null)
  return group.hitTest (r);
else
  return null;
```

## Task 2: Record Extension

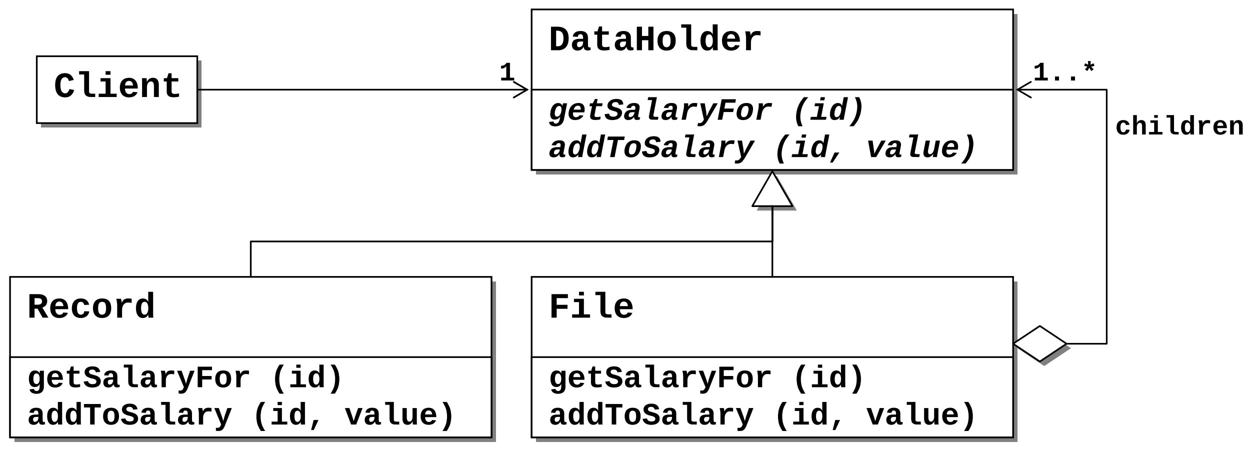
Design an access layer for record-oriented employee data, which is held in several files. Your access layer should support the following operations:

- `getSalaryFor (id)` returns the salary for the employee with the given id.
- `addToSalary (id, value)` increases the salary for the employee with the given id.

Employee data is stored in hierarchically organised files. Each file contains a sequence of records, each of which either stores information about one employee or references another file with further employee records. This structure has been used to reflect the hierarchy of the companies organisation in the structure of the data storage.

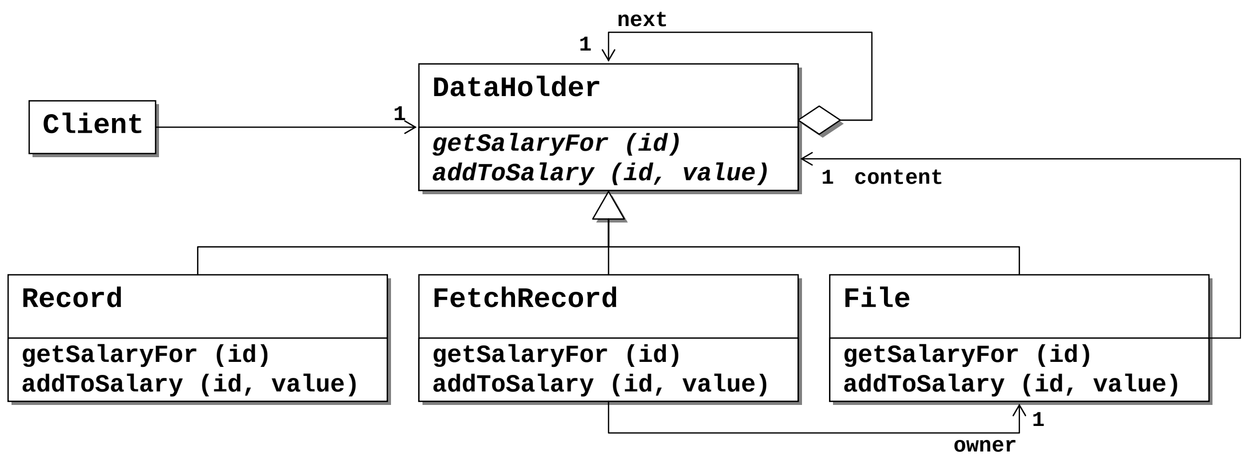
2a) **Task:** Design the core class structure. Your design should hide from the client all information regarding the specific file in which a record is situated, while maintaining this information internally. Which design pattern could you use?

**Solution:** We can use COMPOSITE as in the figure. The top-level file probably contains links to the files containing the actual data (perhaps one per department).



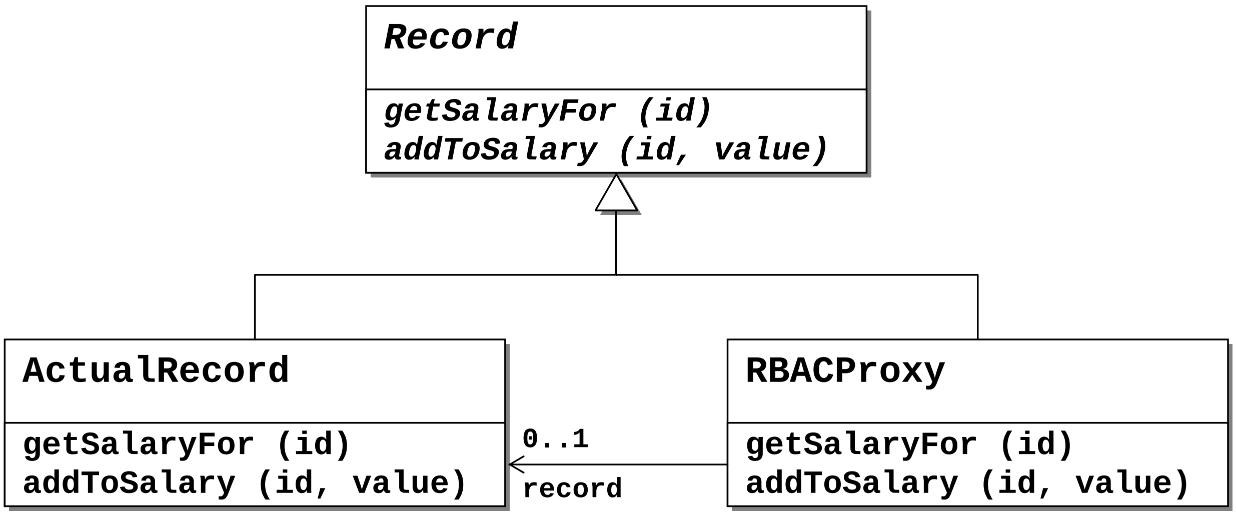
2b) **Task:** The employee database can become rather big. It is therefore not very intelligent to load all records into memory when the database is first opened. Rather, we would like to load only the records that are accessed. Use Chain of Responsibility to implement this feature.

**Solution:** We change the composite pattern into a chain pattern, where each data holder knows its next data holder. Initially, we have a `File` instance for the top-level file, containing only one instance of `FetchRecord.` When `getSalaryFor()` is called on a `FetchRecord,` this will load the corresponding record (respectively open the corresponding file) and link a corresponding instance directly before itself. The `getSalaryFor()` call will then be redirected to the new instance.



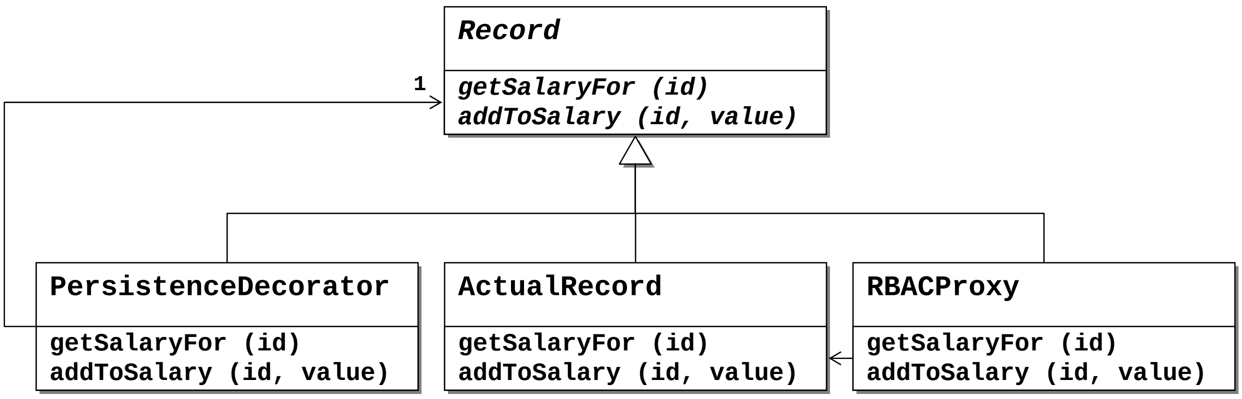
2c) **Task:** Employee information is rather sensitive data, so we do not want everybody to be able to access it. Enhance the design to allow for role-based access control (RBAC) to data.

**Solution:** We use a protection proxy that we hand to our clients. This proxy collects all references to the actual record and performs RBAC on each access..



2d) **Task:** Add the capability to keep the physical records in the files updated whenever `addToSalary()` has been called. Realise it so that files can be opened in read-only or in read-write mode.

**Solution:** We use an additional Decorator `PersistenceDecorator` which saves the record to the disk when it has been changed. `FetchRecord` gets an additional parameter which tells it whether to create the additional decorator.



## Task 3: Discussion of Patterns

3a) **Task:** Compare Proxy and Mediator. What are commonalities, what are differences?

**Solution:** Proxy and Mediator do not have much in common. With Proxy, both the subject and the proxy inherit from a common superclass. The Mediator only "knows" the participants. In Proxy, the flow of control is always directed from the client to the subject; in Mediator, it is bidirectional.

It is possible to combine both patterns: a mediator can mediate proxies. However, a proxy can also hide a mediator (see CORBA ORB).

3b) **Task:** Enumerate 4 different types of Proxies, and tell what they do.

**Solution:**

1. A *remote proxy* provides a local representative for an object in a different address space.
2. A *virtual proxy* creates expensive objects on demand.
3. A *protection proxy* controls access to the original object. Protection proxies are useful when object should have different access rights.
4. A *smart reference* is a 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 for *garbage collection*
   - checking for *locking*

This list has been taken directly from the GOF book.

3c) **Task:** Compare ADAPTER and BRIDGE. Enumerate commonalities and differences.

**Solution:** ADAPTER and BRIDGE have a few things in common:

- Both enhance flexibility by introducing an additional indirection into the access to an object.
- Both hand on commands from an interface that the object originally did not implement.

The key difference between the two patterns can be found in their intents. ADAPTER focuses on resolving incompatibilities between two existing interfaces. It does not care for the implementation of these interfaces, nor does it care about the possible development of class hierarchies. ADAPTER couples two classes developed independently. Their cooperation has originally not been planed or predicted.

Users of BRIDGE, however, know from the beginning that there will be multiple implementations for the same abstraction.

3d) **Task:** Compare BRIDGE and TEMPLATEMETHOD. Enumerate commonalities and differences.

**Solution:** TEMPLATE METHOD and BRIDGE both declare methods in abstract superclasses and implement them in subclasses.

The key difference is that in the abstract class of TEMPLATE METHOD, one method is implemented. This breaks the complete separation of abstraction and implementation intended by BRIDGE. Abstraction and implementation can no longer be refined independently.

TEMPLATE METHOD uses operations from the same object only. BRIDGE uses operations from another object. This can lead to runtime errors in weakly typed languages (e.g., SmallTalk).

3e) **Task:** Enumerate the cases in which VISITOR can be employed. Characterize the advantages of the pattern.

**Solution:** VISITOR can be used to advantage, when an object structure with many classes must be traversed, and class specific operations are to be executed on each object.

VISITOR helps group into one class operations that belong together semantically, even though they work on different classes. This helps avoid cluttering data classes with lots of operations for different use cases. All operations of the same use case are encapsulated in one VISITOR.

Additionally, VISITOR is perfect when the data structure changes rarely, but new operations are added frequently.

VISITOR is often used together with COMPOSITE.

3f) **Task:** Compare TemplateMethod and Strategy. What are commonalities, what are differences?

**Solution:** In Template Method, the algorithm is fix, but some parts are variable. On the other hand, in Strategy, the algorithm is variable, and the client is fix.