TECHNISCHE UNIVERSITÄT DRESDEN

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

WS2019/20 – Design Patterns and Frameworks

Variability Patterns

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

Task 1 Template Method vs Template Class

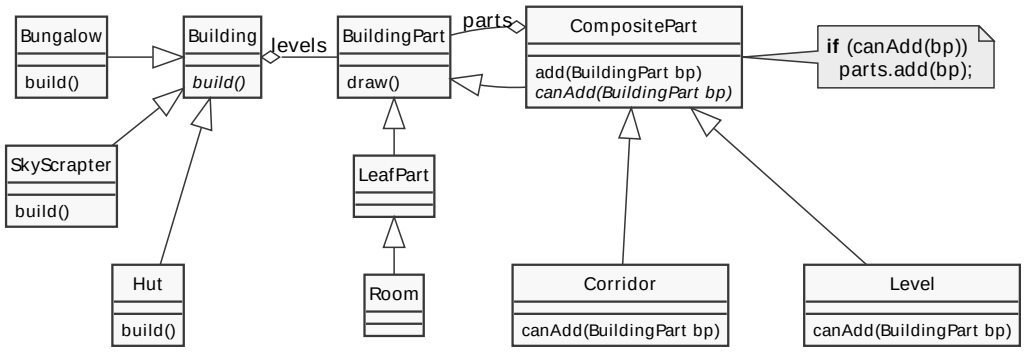
This exercise focuses on patterns for variability as introduced in the *Gang of Four* book [1]. Consider, for example, you have to write a tool for architects that visualizes buildings of different types. Usually, a building is structured from levels, levels are structured from corridors, and corridors from rooms.

There are different classes of buildings: skyscrapers, bungalows, and huts.

a) Create a hierarchy of building types and another hierarchy defining the building's structure. Use `TemplateMethod` to make sure structural constraints (for example, only corridors may contain rooms) are maintained for the building parts of a concrete building.

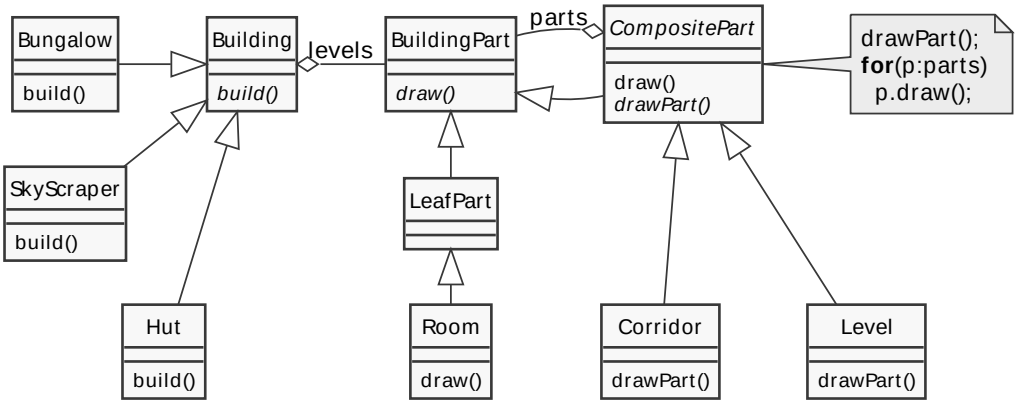
Hint: Apply `Composite`*, to define the building's structure.*

Solution: The following class diagram shows a possible solution. It uses the `Composite` pattern to define the various elements of buildings. Elements can be added to a building using BuildingPart's `add()` method. For composite parts, this method is implemented using `TemplateMethod`. `add()` is the template, `canAdd()` the hook. `canAdd()` can now be implemented variously so as to enforce the structural constraints as needed.



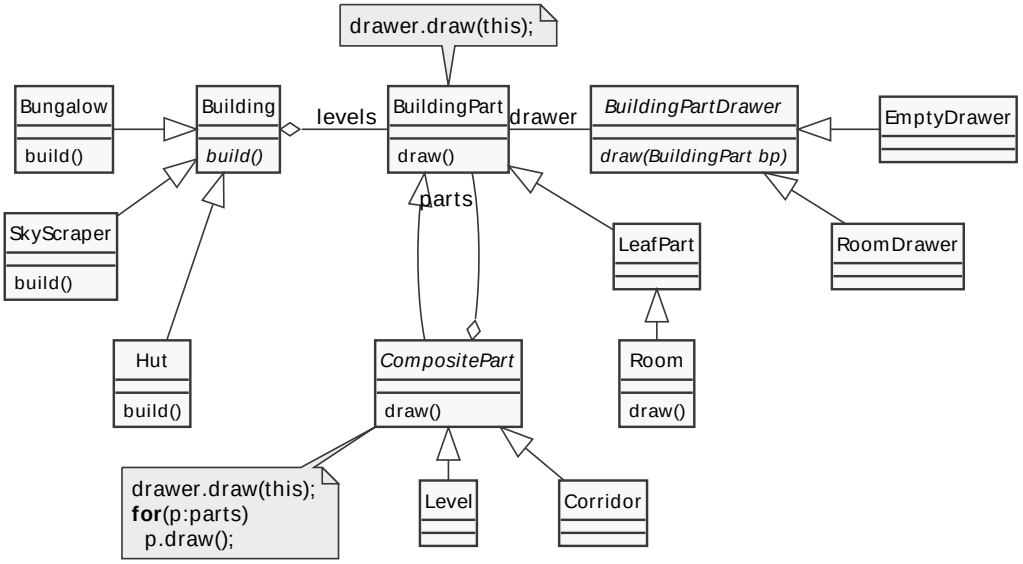
b) Design an iterator algorithm that walks over all types of buildings and draws them room by room on the screen (we assume that only rooms draw themselves). Apply `TemplateMethod`.

Solution: For the following class diagram a similar technique is used. Here, `draw()` is the template and `drawPart()` the hook.



c) Now, change the `TemplateMethod` into a `TemplateClass` pattern (or `Strategy`). Zip out all hook methods from the concrete template class and put them into a separate hierarchy. Which advantages and disadvantages has your new design?

Solution: The new design, depicted below, creates a `BuildingPartDrawer` for printing (pattern `Objectifier`, `TemplateClass`, or `Strategy`). It creates two objects for drawing a building item at runtime. Hence, it wastes space and allocation time. Also, polymorphic dispatch must be done twice, if the template and the hook classes can be varied. Hence, an application will be slower.



d) So far, only rooms are drawn. Now, draw all elements of a building (building, level, corridor, room) on the screen. Note that for every class of building and every building element you have to vary the behavior separately; that is, different buildings require different ways of drawing their individual elements.

Hint: Again use `TemplateClass`.

Why is it impossible to use `TemplateMethod`?

Solution: The design from the previous subtask (see above) can be used without problems; only the implementations for the classes `LevelDrawer` and `CorridorDrawer` have to be added, substituting empty printers.

The design would be impossible to do with `TemplateMethod`, because with that pattern, levels are printed as levels, corridors are printed as corridors, and rooms are printed as rooms, independent of which building they are used for. With the above design, however, levels, corridors, and rooms can be configured with building-specific printer objects. (Of course, one would use an `AbstractFactory` for the printer objects, which allocates precisely what a building needs.)

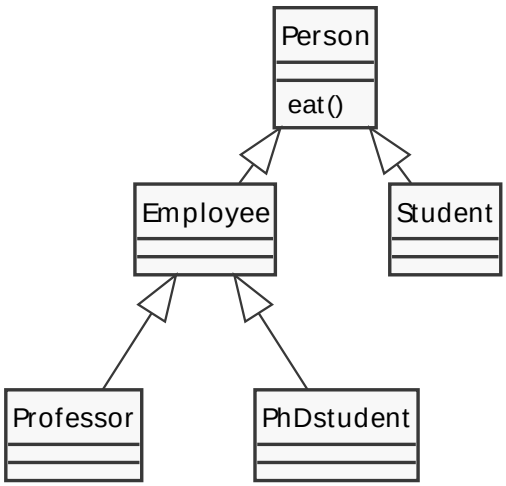


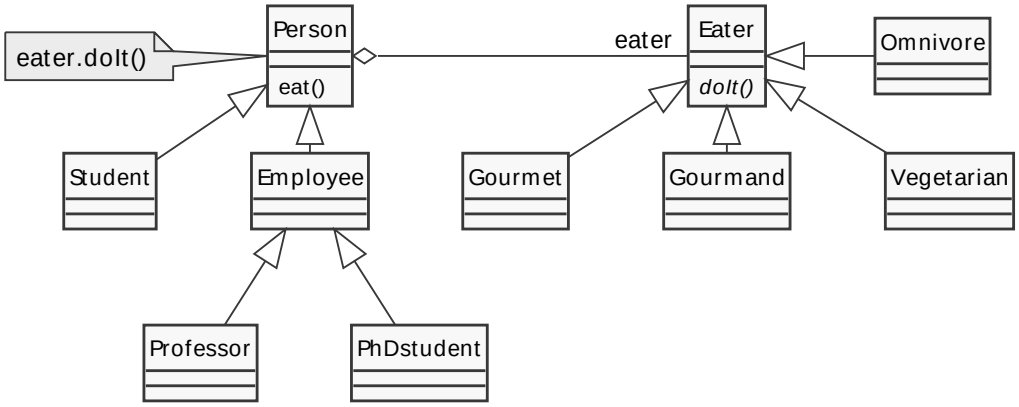
Figure 1: Hierarchy of persons.

Task 2 Objectifier, Reifying Methods

Consider the simple class hierarchy depicted in Figure 1.

a) Reify the method `eat` to the pattern `Objectifier` (or `Strategy`). Distinguish omnivores, vegetarians, gourmets, and gourmands.

Solution: The Person class gets a reference to the new class hierarchy of Eaters, as depicted below:



b) Which linguistic process corresponds to the reification of methods, i.e., to the `Objectifier`?

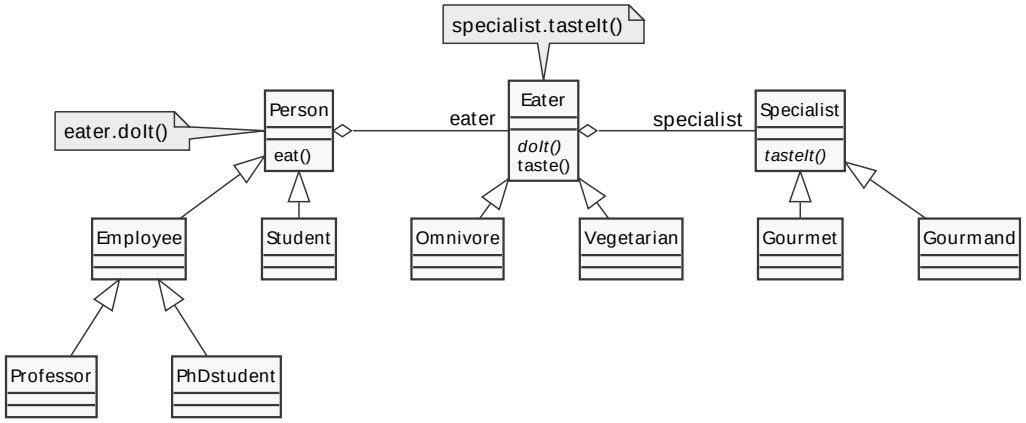
Solution: Turning a verb into a noun.

c) What is the problem, if you group all 4 classes of eaters into one class hierarchy?

Solution: They consider different facets of eaters, i.e., do not partition the class `Eater`. Hence, they are not really comparable and should be split into two dimensional hierarchies.

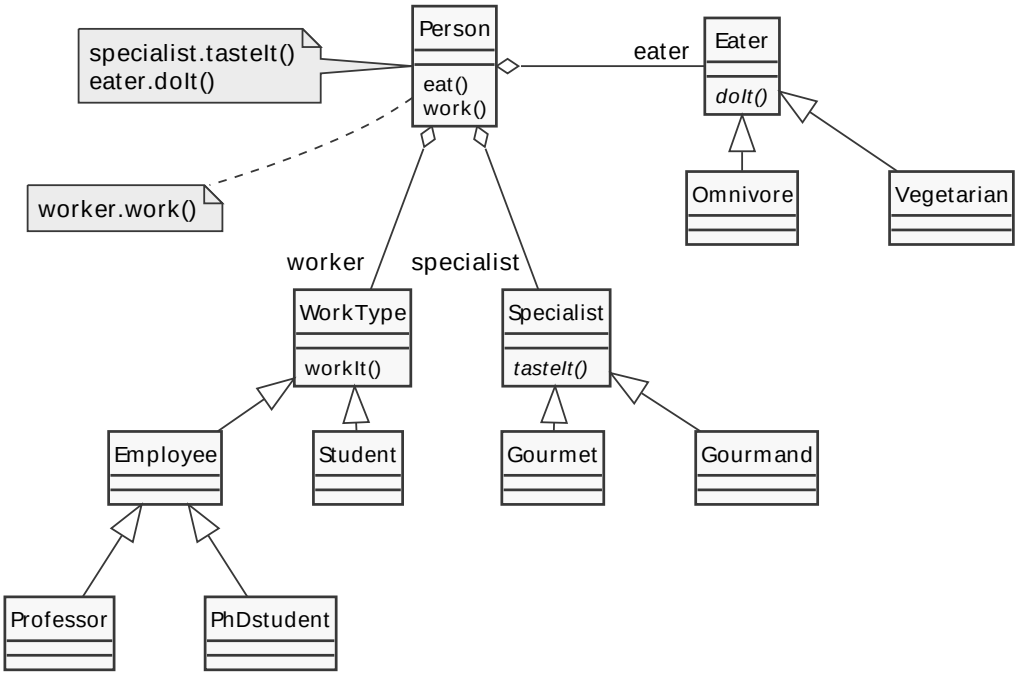
d) Split the eater hierarchy with a simple `DimensionalClassHierarchies` (or `Bridge`) pattern.

Solution: Splitting the `Eater` hierarchy into another `Bridge` gives rise to a sequenced double bridge:



e) Now split all facets of a person (including the `Eater` hierarchy) into `Bridges` using `Person` as the central class.

Solution: The solution changes the inheritance between `Person` and `Employee`/`Student` into an aggregation to a new class `WorkType`:



This way, no facet gets special attention as the primary facet. For performance reasons, it may be useful to make one of the facets the primary one in the implementation, but this should be an implementation decision based on usage, rather than a design decision based on no good reason at all.

Task 3 Comparison of Variability Patterns

a) Compare `Bridge` and `TemplateMethod`. What are commonalities, what are differences?

Solution: Bridge and `TemplateMethod` have in common that they define abstract methods in super classes which are implemented in subclasses. The difference is that an instance of a `TemplateMethod` implements the abstract method, but a `Bridge` hides how the abstract methods are implemented; interface and implementation are split. In a `Bridge`, abstraction and implementation can be refined separately; this is impossible with a `TemplateMethod`.

b) Compare `TemplateMethod` and `Strategy`. What are commonalities, what are differences?

Solution: `TemplateMethod` is used when parts of an algorithm should be varied. With a `Strategy`, the entire algorithm is varied.

The pattern `TemplateMethod` is checked by the compiler. It can already discover inconsistencies in the inheritance relation, and the typing. With `Strategy`, problems occur at runtime and cannot be discovered statically.

c) Compare `TemplateClass` and `GenericTemplateClass`.

Solution: `TemplateClass` uses polymorphism to dispatch to the concrete hook classes. In `GenericTemplateClass`, the polymorphic dispatch is expanded at compile time, by the generic expansion. Hence, there is more type safety. The design pattern `GenericTemplateClass` flattens the inheritance hierarchy of the hook classes. Hence, if there are too deep inheritance structures resulting, the designer can flatten parts of the hierarchies by generic expansion.

Task 4 Homework for Next Exercise (optional)

In this task you shall investigate *Search*,[1] a small framework encapsulating a variety of search algorithms.

a) Go through the code and identify three *Variability Patterns* introduced in the framework and outline their intend.

b) Create a class diagram for each found *Variability Pattern* highlighting the employed design pattern.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[1] https://github.com/Eden-06/Search