Worksheet: Creation of Objects 2

Goals

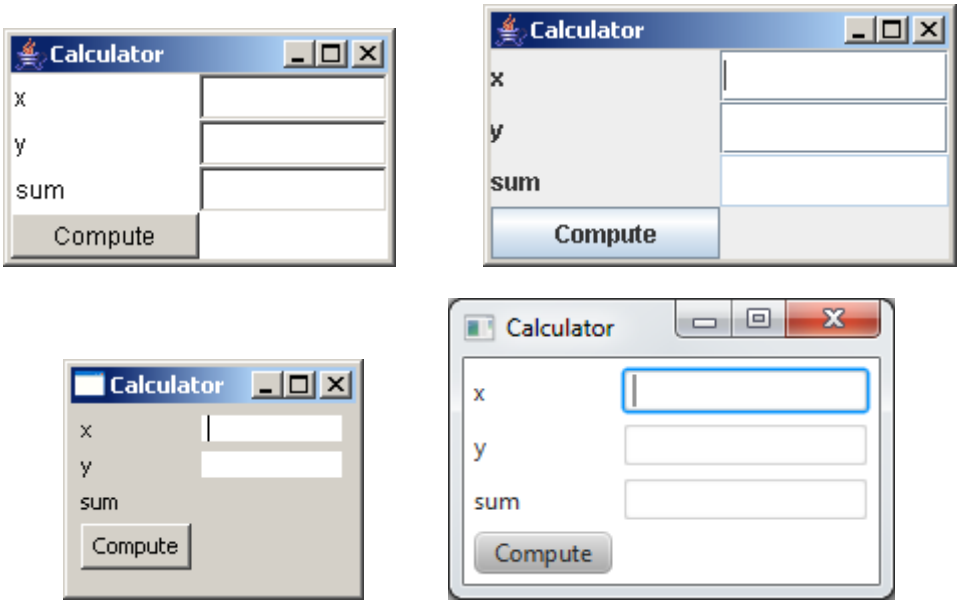
A *factory method* helps decoupling the creation of an object from its use. This approach can still be too rigid, for example if the decision which classes are to be instantiated should be taken at runtime.

By combining the *factory method* pattern with a *strategy* pattern, a more flexible factory implementation can be derived: The *Abstract Factory* pattern.

You will derive this design pattern with the following tasks.

Context

In this week's project *12_Factory.zip* on the file server (active directory) you find the implementation of a simple calculator (which actually can only add two integers - but the actual use case is negligible and therefore kept easy). The implementation of the calculator uses an abstraction `Components` (defined as an interface) to build a graphical user interface (GUI). Implementations of this abstraction are available for the GUI toolkits *AWT*, *Swing*, *SWT* and *FX*.



Class `Gui01FactoryMethods` contains a running implementation of the calculator, based on simple *factory methods*. These methods decide at runtime with the help of a `switch` statement which concrete components have to be instantiated.

Tasks

Study class `Gui01FactoryMethods`. Do you recognize the code-smell? It was particularly difficult for us when we added FX as another variant recently.

As an antidote to such code smells the *strategy pattern* can be used. Apply this pattern to improve the code which creates the components, i.e. to get rid of the case statements.

a) Draw a simplified class diagram based on the given implementation of `Gui01FactoryMethods`. *Simplified* means that the general structure should be recognizable, but you do not have to explicitly add all existing parallel concrete classes.

b) Draw the class diagram of the *strategy pattern* adapted to this use case.
Think about the methods which have to be defined in the strategy interface for this case.

c) Implement the design you found in b) in a new calculator class and test it with at least one GUI variant.

d) It is difficult to completely get rid of the coupling to concrete classes, because somehow the strategy to be used has to be "set" (in order to be used from all over in the code). Where would you store this strategy and who stores it there initially? Propose several approaches and indicate for each approach where something needs to be changed in order to use a different strategy.

e) Which classes do you need to adjust or to add in order to support an additional GUI technology?

f) Assume that we want to support checkboxes in addition to windows (`Frame`), fields (`Field`), buttons (`Button`) and labels (`Label`): Which classes do you need to adjust or to add?

SWT Remarks:

SWT (Standard Widget Toolkit) is a GUI-toolkit that was developed in the contex of Eclipse. The SWT implementation of our components can only be compiled without errors if the corresponding `swt.jar` file is on the class path.

We added a plugin to the build.gradle file which should load these platform specific dependencies into your project. However, if that does not work, you can simply ignore these errors or delete the `ComponentsSWT` class (together with the corresponding cases in the switch statements in class `Gui01FactoryMethods`).

If you do not use Gradle and still want to try out SWT, you must add the `swt.jar` file corresponding to your operating system to your project's class path. To do so, go to
http://download.eclipse.org/eclipse/downloads/,
select an Eclipse version (for example, 4.9), search for SWT and download the corresponding release. For Version 4.9 the following downloads are provided:

The following releases are also available on the file server (AD):

- swt-4.9-win32-win32-x86.zip — Windows (32 bit version)
- swt-4.9-win32-win32-x86_64.zip — Windows (64 bit version)
- swt-4.9-gtk-linux-x86.zip — Linux (32 bit version)
- swt-4.9-gtk-linux-x86_64.zip — Linux (64 bit version)
- swt-4.9-gtk-linux-ppc64le.zip — Linux (64 bit version for Power PC)
- swt-4.9-cocoa-macosx-x86_64.zip — Mac OSX (64 bit version)

Extract the corresponding zip file and add the file `swt.jar` contained in the zip file to your project.

If you work with Eclipse, you can alternatively also directly add the corresponding jar file that is stored in the plugins directory of your Eclipse installation as an external jar file to the classpath. On Windows (64bit) / Oxygen. This is file:

`eclipse\plugins\org.eclipse.swt.win32.win32.x86_64_3.108.0.v20180904-1901`.