
Stellen Sie bitte sicher, dass alles wie vorhergesehen läuft, bevor Sie dieses Übungsblatt abgeben. **Starten Sie den Kernel neu** (in der Menüleiste die Option Kernel→Restart auswählen) und **validieren** Sie anschließend das Übungsblatt (in der Menüleiste auf Validate klicken) um Rückmeldung zu eventuellen fehlenden oder fehlerhaften Eingaben zu erhalten.

Füllen Sie alle Stellen im Übungsblatt aus, welche entweder `DEIN CODE HIER` oder "DEINE ANTWORT HIER" enthalten. Geben Sie unterhalb Ihren vollständigen Namen an.

Wenn Sie Code-Bestandteile aus anderen Quellen (wie z.B. Stackoverflow) kopieren, dann machen sie den kopierten Code in ihrer Quellcodedatei kenntlich und fügen eine Referenz auf die Quelle als Kommentar hinzu.

Wenn Sie die Aufgaben in einer Gruppe erledigen, dann fügen Sie die Namen aller Gruppenmitglieder in der nachfolgende Zelle zu `Name` und zusätzlich als Kommentar am Anfang Ihrer Quellcodedatei hinzu.

In [ ]:
```
NAME = ""
```

Methoden der Softwareentwicklung II

SS 2023

Übungsblatt 6

Abgabe bis **Sonntag, 14.Juni 2023, 23:55 Uhr**

In [ ]:
```
import jagl
import os
import shutil
import re
```

Klausur vom letztem Jahr: Fabrik

Es soll eine Simulationssoftware für eine Fabrik programmiert werden. In dieser Fabrik stehen verschiedene Machinen, die unterschiedliche Produkte herstellen. Die fertig gestellten Produkte werden in nach Produkttypen getrennten Lager einsortiert.

Das Herzstück der Simulationssoftware ist eine Schleife, die den zeitlichen Fortschritt implementiert (implementiert in der `run()` -Methode in der Klasse Factory). Den zeitlichen Verlauf modellieren wir dabei als eine Abfolge von diskreten Zeitschritten (die Iterationen der Zeitschleife, auch `Ticks` genannt).

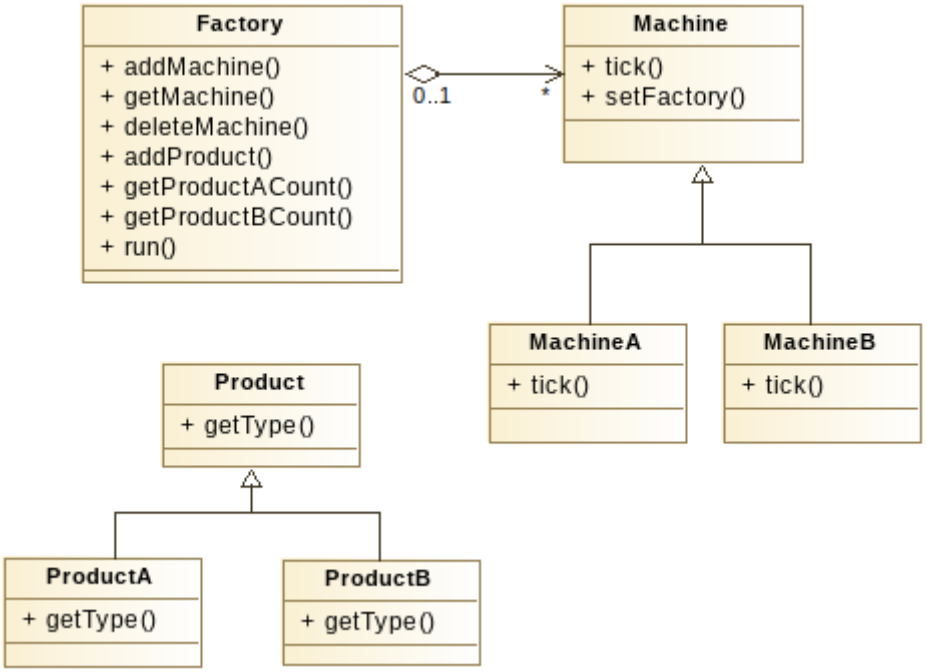
Ablauf Zeitschleife

In jeder Iteration der Zeitschleife wird für jede einzelne Maschine die `tick()` -Methode aufgerufen. In der `tick()` -Methode werden dann die entsprechenden Produkte erzeugt und der Fabrik mithilfe der `addProduct()` -Methode übergeben. Mit einer gewissen Wahrscheinlichkeit kann es passieren, dass es zu einem Fehler kommt (d.h. es werden keine Produkte erzeugt). Ein `Machine Failure` (repräsentiert durch eine `MachineFailureException`) bedeutet, dass die betreffende Maschine für die nächsten 3 Ticks keine Produkte produzieren kann. Eine `Machine Explosion` (repräsentiert durch eine `MachineExplosionException`) bedeutet, dass die entsprechende Machine permanent kaputt ist und aus der Fabrik entfernt werden muss.

Lassen Sie aus praktischen Gründen nach jeder Iteration die Zeitschleife mindestens eine Sekunde schlafen (mit `sleep()`).

Grundlegende Softwarearchitektur

Die grundlegende Software-Architektur schaut folgendermaßen aus:



Public Interface

Das im folgenden gegebene öffentliche Interface soll **genauso wie angegebenen** implementiert werden. Sie können dieses auch um weitere Funktionen erweitern, wenn Sie dies für nötig erachten.

Klasse Factory

Die Klasse `Factory` repräsentiert die Fabrik und implementiert die Simulation. Sie verwaltet die Maschinen, d.h. sie übernimmt für diese die Object Ownership. Da immer wieder neue Maschinen hinzugefügt und alte entfernt werden, soll die Maschinenverwaltung über einen dynamische Datencontainer erfolgen. Wählen Sie selbstständig einen passenden Datencontainer aus.

Des Weiteren verwaltet auch die Fabrik auch die Lager der Produkte, d.h. auch hier übernimmt sie für diese die Object Ownership. Für jeden Produkttypen soll es ein eigenes Lager geben, die durch separate dynamische Datencontainer repräsentiert werden. Wählen Sie selbstständig passende Datencontainer aus.

- Objektfunktionen:
  - `unsigned addMachine(Machine* m)` : Fügt eine neuen Maschine hinzu. Der Rückgabewert ist eine ID, die die jeweilige Maschine **eindeutig** identifiziert.
  - `Machine* getMachine(unsigned id)` : Gibt die Maschine mit der angegebenen ID zurück.
  - `void deleteMachine(unsigned id)` : Entfernt die Maschine mit der angegebenen ID und gibt alle damit verbundenen Ressourcen wieder frei.
  - `void addProduct(Product* p)` : Übergibt ein neues Produkt der Fabrik. Die Fabrik muss dann den Typ des Produkts bestimmen und in das entsprechende Lager einsortieren. Wenn ein unbekanntes Produkt übergeben wird, dann soll eine `MachineFailureException` geworfen werden.
  - `unsigned getProductACount()` : Gibt die Anzahl der im Lager vorhandenen Produkte A zurück.
  - `unsigned getProductBCount()` : Gibt die Anzahl der im Lager vorhandenen Produkte B zurück.
  - `void run(unsigned iterations)` : Diese Methode implementiert die Zeitschleife. Der Eingabeparameter `iterations` gibt an, nach wievielen Iterationen die Zeitschleife abgebrochen werden soll (0 bedeutet, dass die Schleife nie abgebrochen wird).

Klasse Machine

Die Klasse `Machine` ist die Oberklasse aller Maschinen und definiert deren öffentliches Interface.

- Objektfunktionen:
  - `void tick()` : Simuliert das Produzieren von Produkten (siehe Ablauf Zeitschleife für genauere Informationen).
  - `void setFactory(Factory* f)` : Damit eine Maschine ein neu erstelltes Produkt der Fabrik mithilfe der Methode `addProdukt()` übergeben kann, benötigt die Maschine eine Referenz auf die Fabrik. Mithilfe dieser Methode kann die entsprechende Referenz der Maschine in der `addMachine()` -Methode übergeben werden.

Konkrete Maschinen

Es gibt zwei konkrete Maschinen:

- MachineProductA:
  - Produziert pro Zeitschritt 2 Einheiten von Produkt A.
  - Hat eine 15% Wahrscheinlichkeit eine `MachineFailureException` zu werfen.
  - Hat eine 2% Wahrscheinlichkeit eine `MachineExplosionException` zu werfen.
- MachineProductB
  - Produziert pro Zeitschritt 3 Einheiten von Produkt A.
  - Hat eine 20% Wahrscheinlichkeit eine `MachineFailureException` zu werfen.
  - Hat eine 5% Wahrscheinlichkeit eine `MachineExplosionException` zu werfen.

Klasse Product

Die Klasse `Product` ist die Oberklasse aller Produkte und definiert deren öffentliches Interface.

- Objektfunktionen:
  - `int getType()` : Identifiziert den Produkttypen (Damit die automatischen Tests korrekt funktionieren, verwenden Sie bitte hier dynamische Bindung).

Konkrete Produkte

Es gibt zwei konkrete Produkte:

- ProductA:
  - `getType()` gibt 1 zurück.
- ProductB
  - `getType()` gibt 2 zurück.

Exceptions

Die Basisklasse aller Exceptions soll `FactoryException` heißen, die wiederum von `std::exception` erbt. Überschreiben Sie die Funktion `const char* what()` der Klasse `std::exception` , sodass eine aussagekräftige Fehlermeldung zurückgegeben wird (alternativ können Sie auch von `std::runtime_error` erben und die Fehlermeldung dessen Konstruktor übergeben).

Wenn Sie es für notwendig erachten, können Sie auch weitere Exception-Klassen neben den schon oben genannten hinzufügen.

Sorgen Sie dafür, dass es für alle möglichen Exceptions einen passenden Exception-Handler gibt.

Interne Implementierung

Wie Sie die gewünschte Funktionalität intern implementieren, bleibt Ihnen überlassen. Wählen Sie passende Sichtbarkeiten und vergessen Sie nicht, wenn möglich das `const` Keyword bei Eingabeparametern und wenn passend Call-by-Reference zu verwenden.

Wählen Sie selbstständig passende Datencontainer aus. Es darf zu keinem Zeitpunkt zu einem Speicher- oder sonstigen Ressourcenleck kommen. Sorgen Sie dafür, dass, wenn die Fabrik-Instanz zerstört wird, alle Ressourcen wieder sauber freigegeben werden.

Überlegen Sie sich außerdem, wo sie dynamische Bindung verwenden.

Separieren Sie bitte die `main()` Funktion in einer eigenen Datei, die Sie `main.cpp` nennen. Der Hintergrund ist, dass die automatischen Tests ihre eigene `main()` Funktion verwenden, daher muss diese einfach austauschbar sein. Der Inhalt ihrer `main()` -Funktion ist für die Prüfung nicht relevant.

Sorgen Sie für ausreichend Debug-Ausgaben bei der Programmausführung, damit man nachvollziehen kann, was die Fabrik gerade tut.

Die restlichen Dateien können Sie benennen, wie Sie wollen.

Geben Sie mit der Abgabe auch ein Makefile ab, welches Ihr Programm kompiliert.

Unverbindliche Schritt-für-Schritt Anleitung

1. Beginnen Sie mit der Implementierung der Klasse `Product` und deren Subklassen. Gehen Sie dabei den Vererbungsbaum von oben nach unten durch. Testen Sie diese ausführlich.
2. Beginnen Sie mit der Implementierung der Klasse `Machine` und deren Subklassen. Gehen Sie dabei den Vererbungsbaum von oben nach unten durch. Testen Sie diese ausführlich.
3. Implementieren Sie anschließend die Klasse `Factory` und testen diese ausführlich.
4. Bevor Sie beginnen Detailfunktionalitäten zu implementieren, stellen Sie zuerst sicher, dass die Grundfunktionalität korrekt und vollständig implementiert ist (z.B. bevor Sie die 3 Ticks Wartepause nach einer MachineFailureException implementieren, sollten Sie zuerst das grundlegende Maschinenmanagement in der Factory Klasse vollständig implementieren).

In [ ]:
```
# new test suite
jagl.testsuite_begin("Übungszettel 6")

# delete all build artifacts
_ = jagl.remove_paths(files=["factory", "*.o"])
```

In [ ]:
```
%%bash
#
# Kopieren Sie vor der Abgabe ihr Makefile und Ihre Quellcodedatein in dassel
# in dem dieser Übungszettel zu finden ist.
#
# Der Name der ausführbaren Datei sollte 'factory' lauten.
#

make
```

In [ ]:
```
# Weisen Sie der untenstehenden Variable den Wert True zu, sobald Sie die Auf
```

```
# Datentyp: bool
exercise_6_1_solved = False

# DEINE ANTWORT HIER
```

In [ ]:

```
@jagl.testcase("1", desc="Exercise Solved")
def testcase_1_1(result, suite, case):
    varname = "exercise_6_1_solved"
    result.setSucceeded()
    jagl.check_variable_exists_and_has_type(result, globals(), varname, bool)
    if result.isSucceeded():
        if eval(varname):
            result.setSucceeded("Exercise solved.")
        else:
            result.setFailed("Exercise not solved.")
```

In [ ]:

```
@jagl.testcase("2", desc="Check Makefile", deps=["1"])
def testcase_1_2(result, suite, case):
    result.setSucceeded()
    jagl.check_path_exists(result, files=["Makefile"], message = "Are you sur
    if result.isSucceeded():
        result.addMessage("Makefile was found.")
```

In [ ]:

```
@jagl.testcase("3", desc="Building program", deps=["2"])
def testcase_1_3(result, suite, case):
    progname = "robot"
    if os.path.isfile(progname):
        result.setSucceeded("The program could be build successfully.")

    else:
        res = jagl.exec_bin("make")
        if res.state == jagl.ExecutionResult.SUCCESS:
            stdout = jagl.bytes_decode(res.stdout).strip()
            stderr = jagl.bytes_decode(res.stderr).strip()
            if res.exitCode == 0:
                if os.path.isfile(progname):
                    result.setSucceeded("The program could be build successfu
                else:
                    result.setFailed("make was executed successfully, but I c
                if len(stderr) > 0:
                    result.setFailed(["However, there were errors."] + stderr
            else:
                result.setFailed(["The program could not be build:"] + stderr
        elif res.state == jagl.ExecutionResult.NOTFOUND:
            result.setFailed("make could not be found. Please inform your cou
        elif res.state == jagl.ExecutionResult.TIMEOUT:
            result.setFailed("Timeout while calling make. Please inform your
        elif res.state == jagl.ExecutionResult.ERROR:
            result.setFailed('Unknown error during execution of make: ' + str
```

In [ ]:

```
@jagl.testcase("4", desc="Executing program", deps=["3"])
def testcase_1_4(result, suite, case):
    progname = "robot"
    execresult = jagl.exec_bin("./" + progname, [], timeout = 30)
    jagl.check_execution_result(result, execresult, checkExitCode = True)
    if execresult.exitCode is not None:
        if execresult.exitCode == -11:
            result.setFailed("Program caused a segmentation fault.")
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```
#try:
#    shutil.move("main.cpp.bak", "main.cpp")
#except:
#    pass
```