Project Capycity

Author: Markus Hülß

HOCHSCHULE
COBURG

Szenario

Capybaras sind unglaubliche Ingenieure, erst kürzlich haben sie einen Durchbruch bei der umweltfreundlichen Energie erlangt. Sie beschließen daher eine extra Stadt namens Capycity aufzubauen, welche ihren Fokus auf die Erzeugung von Energie legt und somit umgebende Städte vollständig versorgen kann.

Aktuell sind sie in der Lage mithilfe folgender Technologien Energie zu erzeugen:

- Wasserkraftwerk
- Windkraftwerk
- Solarpanele

Kapitel 1 - Der Aufbau (1 Punkte)

Die Landfläche

Um Capycity zu bauen, benötigt es zunächst bebaubarer Landfläche. Die Ingenieure wollen nicht ohne ordentlich Planung anfangen. Ein Capybara, auf dessen T-Shirt steht "!false - it's funny because it's true", blickt in den Raum und schlägt vor ein erstes Simulationstool zu entwickeln.

"Wir benötigen ein Tool, welchem wir eine Fläche der Form LxB (Länge mal Breite) mitgeben können, die dann virtuell im Speicher erstellt wird. Danach soll es uns die Möglichkeit geben verschiedene Gebäude an verschiedenen Plätzen zu platzieren. Um Arbeit zu sparen, sollte das Tool gleich nach der Größe, auch wieder in der Form LxB sowie der Position des Gebäudes fragen. Sollte es eine Kollision zwischen zwei Gebäuden geben, dann soll natürlich ein Fehler ausgegeben werden. Es muss daher für Korrekturen die Möglichkeit geben Bereiche wieder als bebaubar zu deklarieren. Und damit wir den Plan ordentlich betrachten können, muss dieser natürlich ausdruckbar sein"

Die anderen Capybaras stimmen nickend zu und machen sich gleich an die Arbeit.

Aufgaben

Schreibe ein erstes Simulationstool (Kommandozeilentool), welches folgende Features besitzt:

- ☐ Erhalt der Länge und Breite des Baubereichs als Argument über die Kommandozeile.
- ☐ Erstellung eines Arrays, welches den Baubereich repräsentiert. Elemente sollen dabei ein Enum sein, welcher den Gebäudetypen darstellt (bzw. ein Feld als leer darstellt, wenn sich auf einer Position kein Gebäude befindet)
- ☐ Anzeige eines Menüs mit folgenden Einträgen und dahinterstehender Funktionalität
    - ☐ Gebäude setzen (mit darauffolgender Nachfrage nach Art, Länge, Breite und Position)
    - ☐ Bereich löschen (Betroffene Gebäude sollen nicht gelöscht, sondern dadurch nur verkleinert werden)
    - ☐ Ausgeben des aktuellen Bauplans
    - ☐ Beenden des Programms
- ☐ Prüfung ob Teile eines zu bauenden Gebäudes mit anderen Gebäuden kollidiert oder außerhalb des Baubereichs liegt.

Kapitel 2 - Das Review (2 Punkte)

Die Anpassungen

Der Capybara mit dem nerdigen T-Shirt, Bob ist sein Name, ruft alle andere Capybaras zusammen und lässt sich den Zwischenstand zeigen. Er spielt mit dem Tool ein wenig herum und schaut sich auch den Code an. Kurz darauf nickt er zufrieden. “Ich denke, dass ist ausreichend für die Planung”.
Plötzlich tritt seine Kollegin Carla hervor und tippt mit ihren Finger auf einige Codestellen. “Ich denke, dass wir hier noch einiges machen können. Außerdem haben wir gar keinen Überblick über benötigte Materialien und resultierende Kosten. Erstmal sollte die Planung eine eigenständige Klasse sein. Dann wäre es auch gut die Gebäude als Klassen zu definieren, welche einen Grundpreis, ein Label und eine Auflistung benötigter Materialien enthält. Danach sollten wir das Ausdrucken des Plans erweitern, es sollten uns auch benötigte Materialien und eine Auflistung der Preis sowie der Gesamtpreis angezeigt werden.”
Bob denkt kurz über Carlas Worte nach, blickt zu den anderen Capybaras und sagt: “Carla hat Recht, wir sollten noch die von ihr genannten Anpassungen und Erweiterungen machen, bisher ist alles doch noch etwas rudimentär”
Erneut stimmen die anderen Capybaras nickend zu und machen sich wieder an die Arbeit

Aufgaben

Mache bei dem Simulationstool folgende Anpassungen:

- ☐ Kapsel die Verwaltung der Gebäude innerhalb einer Klasse namens **CapycitySim**
- ☐ Erstelle eine Klasse für Materialien.
  - ☐ Es soll **Holz**, **Metall**, **Kunststoff** geben
  - ☐ Alle leiten sich von der Basisklasse **Material** ab
  - ☐ Jedes Material soll einen eigenen Preis besitzen
- ☐ Ersetze die Enum der Gebäude durch eigenständige Klassen für die einzelnen Gebäude.
  - ☐ Diese sollen sich von einer Basisklasse **Building** ableiten.
  - ☐ Jedes Gebäude soll einen **Grundpreis** besitzen
  - ☐ Jedes Gebäude soll ein **Label** besitzen, welches beim Ausdrucken des Plans angezeigt wird
  - ☐ Jedes Gebäude soll eine **Liste in Form eines Arrays oder Vektors** haben, welche benötigte Materialien enthält. Mehr Materialien einer Art benötigen mehr Plätze (z.B. 2xHolz -> [Holz, Holz] bei einem Array)
- ☐ Erweitere das Ausdrucken des Plans um die Darstellung der Gebäude mit ihren Labels, einer Auflistung der Gebäude sowie deren benötigter Materialien, dem Einzelpreis eines Gebäudes sowie dem Gesamtpreis von allen Gebäuden.

Kapitel 3 - Der Streit (2 Punkte)

Die Entscheidung

Bob und Carla reviewen gerade die aktuelle Version mit der Versionsnummer **0.42-final-build-this-time-really-there-will-be-no-changes-before-monday-plz-let-us-sleep** und fragen sich, ob sie vielleicht doch zum Seminar über Work-Life-Balance hätten gehen sollen. Dabei fällt Carla ein Kommentar im Code auf. “Hey, warum packen wir jedes benötigte Material als extra Objekt in eine einfache Datenstruktur? Das erzeugt doch nur unnötige Redundanzen und Speicherverbrauch – Viele Grüße Anton”.

Carla und Bob denken kurz nach bis Carla meint: “Er hat recht, ich denke jedes Gebäude sollte lieber eine Map besitzen, mit der Klasse als **Key**, welcher das **Material identifiziert** und der **Menge an benötigtem Material** als **Value**. Damit lässt sich auch viel einfacher der Preis von einem Gebäude berechnen.” Bob antwortet leicht skeptisch: “Das würde funktionieren, aber ich denke es wäre besser, wenn die **Verwaltung** der **benötigten Materialien** sowie deren **Preis** in eine **separate Klasse** ausgelagert werden. Man kann dafür ja auch Maps verwenden, nur eben entsprechend gekapselt.”Das würde auch gehen, aber ich finde, dass die Erweiterbarkeit darunter leidet und man schnell irgendwann alle Informationen darin auslagern kann, dann ist die Kapselung auch dahin.” erwidert Carla. “Jaaaaa, aber Materialien sind jetzt keine komplexen Konstrukte, da sollte man nicht mehr rein packen als notwendig” antwortet Bob schon leicht patzig. Carla verdreht kurz die Augen bevor sie leicht entnervt antwortet: “Aber da wären wir doch wieder bei dem von mir angesprochen Problem.” Carla atmet kurz tief ein, um wieder in einem normaleren Tonfall sprechen zu können. “Aber weißt du was, wir haben hier viele fähige Capybaras, sie sollen entscheiden was die bessere Lösung für unsere Anforderungen ist. Was meinst du?” Bob denkt kurz nach und nickt: “Das ist eine gute Idee und dessen Idee sich durchsetzt erhält vom anderen eine große **Käsepizza** mit **extra Käse**, welcher vorher mit anderen **Käse überbacken** wurde”. “Ok, abgemacht” antwortet Carla mit einem leichten Lachen.

Daraufhin gehen die beiden direkt zu den anderen Capybaras und verkündigen die neuen Änderungen.

Aufgaben

Mache bei dem Simulationstool folgende Anpassungen:

- ☐ Überarbeite die Verwaltung der benötigten Materialien von Gebäuden
    - ☐ Realisiere dafür Carlas Idee (Verwaltung innerhalb der Gebäude) **oder** Bobs Idee (Verwaltung über zusätzliche Verwaltungsklasse)
        - ☐ Nutze **Maps** für die Verwaltung

- ☐ Zur eindeutigen und konsistenten Identifizierung muss die **Klasse** als **Key** verwendet werden

☐ Gib an für welche Idee du dich entschieden hast und warum.

Kapitel 4 - Der Release? (4 Punkte)

Die vergessenen Anforderungen

Bob und Carla schauen sich erneut die aktuelle Version an und wirken so, als würden sie keine größeren Probleme mehr im aktuellen Stand sehen. Wie aus dem Nichts tippt jemand beiden auf die Schulter. Sehr erschrocken, da beide sehr vertieft im Review der aktuellen Version waren, drehen sie sich um und sehen Holger vor sich stehen, der merkt, dass er die zwei etwas erschrocken hat und daher noch kurz wartet, bevor er anfängt sein Anliegen zu nennen. “Was meint ihr zu dem aktuellen Stand der Software?” fragt er. “Wir sind damit richtig zufrieden, ich denke, dass wir nur noch ein wenig Refactoring betreiben müssen und dann können wir die erste Version an die Stadtplanung schicken.” sagt Carla mit zufriedener Stimme. “Das freut mich zu hören, aber mir ist noch eine Sache eingefallen. Wir können aktuell noch gar nicht sagen wie effizient ein Plan ist. Am Ende wird eine extrem teure Stadt gebaut, welche aber nur wenig Energie erzeugen kann. Wie soll die Stadtplanung entscheiden können welcher Entwurf der beste ist? Und so richtig vergleichen können wir die Pläne auch nicht. Aktuell kann man immer nur einen erstellen. Sobald ein neuer erstellt wird, wird der vorherige gelöscht” sagt er. Bob und Carla verziehen für wenige Sekunden keine Miene, bevor sich ihr Gesichtsausdruck zu einem leicht panischen wandelt. “Du hast recht!” sagt Bob leicht nervös. Carla stimmt dem nickend zu.

“Wie konnten wir nur nicht daran denken, das ist für die Planung essentiell. Ich denke wir werden nicht um eine weitere Version herumkommen. Holger, du hast daran gedacht, daher denke ich, dass du am ehesten an alle noch notwendigen Anforderungen denkst, was sollte noch alles gemacht werden?” fragt Carla. Holger ist kurz etwas verdutzt bevor er antwortet, da er bisher noch nie nach Anforderungen gefragt wurde, sondern diese bisher immer nur umgesetzt hat. “Ich würde ich vorschlagen, dass wir noch folgendes ergänzen. Man sollte sich zunächst den **vollen Preis**, den ein **Gebäude kostet**, von diesem **einholen** können. Dann müssen **Gebäude** noch zusätzlich eine **Leistung in Megawatt** besitzen. Aktuell hält die **Klasse CapycitySim** immer nur einen Plan. Dieser Plan muss in eine **Blueprint Klasse** ausgelagert werden. Die **Blueprint Klasse** soll dabei neben der Ausgabe des Plans noch eine **Kennzahl berechnen** können, welche **einzeln abgerufen** werden kann, aber auch bei der **Ausgabe des Plans** steht. Die **Kennzahl** berechnet sich dabei wie folgt:”

$$K = \frac{(Leistung\ Gebäude)}{(Preis\ Gebäude) * (Menge\ an\ Flächeneinheiten\ des\ Plans)}$$

“Weiterhin müssen wir in einer **passenden Datenstruktur** alle bisher erstellten **Pläne speichern** können. Wenn ein **neuer Plan** erstellt wird, wird dieser gleich **darin gespeichert** und gilt als der **aktuell zu bearbeitende Plan**. Wir werden dadurch wohl noch einen **neuen Menüpunkt** erstellen müssen, der einen **neuen Plan** erstellt. Da wir nicht ausschließen können, dass Pläne doppelt vorhanden sind, sollten wir beim Erstellen eines neuen Plans prüfen, ob für den letzten Plan bereits ein identischer

vorhanden ist, um diesen in dem Fall zu löschen, ich denke, dass man das gut mit einem **Funktor** in der **Blueprint Klasse** lösen kann. Zuletzt müssen wir auch **alle Pläne ausdrucken** können, dafür können wir ja problemlos auf die **entsprechende Funktion** in der **Blueprint Klasse** zurückgreifen, während wir durch unsere Pläne iterieren. Aber vor jedem Ausdrucken sollten die Pläne vielleicht immer in absteigender Reihenfolge sortiert werden, dann sieht man gleich welcher Plan am besten ist. Am besten nutzen wir dafür eine passende **Lambda-Funktion**. Ich denke, damit sollten wir die fehlenden Anforderungen erfüllen können, was meint ihr?"

Bob und Carla stehen für ein paar Sekunden wortlos da, da sie die Menge an Informationen noch verarbeiten. Kurz darauf sagen beide gleichzeitig "Ich denke, dass sollte so klappen" und schreiben sie noch ein paar Notizen auf, bevor sie zu den anderen Capybaras gehen, um ihnen die letzten notwendigen Anpassungen zu nennen.

- ☐ Ergänze die **Building Klassen** um folgendes:
  - ☐ Methode zur Berechnung und Rückgabe des Preises
  - ☐ Erweiterung um die Eigenschaft **Leistung**, welche auch über eine Methode zurückgegeben werden kann
- ☐ Erstelle eine **Klasse Blueprint**
  - ☐ Die Klasse enthält die Daten und Funktionen eines Plans, welche sich zuvor in der Klasse **CapycitySim** befanden.
  - ☐ Erstelle eine Methode, um die Kennzahl des Plans zu berechnen und zurückzugeben.
  - ☐ Erweitere die Klasse um einen **Funktor**, der prüft, ob der Plan mit einem anderen Plan identisch ist
- ☐ Erweitere die Klasse **CapycitySim**
  - ☐ um eine geeignete Datenstruktur, zur Speicherung mehrerer Pläne.
  - ☐ um einen Menüeintrag, zur Erstellung eines neuen Plans
  - ☐ um eine **Prüfung des aktuellen Plans**, ob es bereits einen **identischen gibt**, um den Plan in dem Fall zu **löschen**. Diese **Prüfung** soll bei der Erstellung eines **neuen Plans** auf den **noch aktuellen Plan angewendet** werden.
- ☐ um ein Ausdrucken aller Pläne, absteigend sortiert nach der Kennzahl. Die Sortieren soll mithilfe einer Lambda-Funktion immer unmittelbar vor dem Ausdrucken durchgeführt werden.