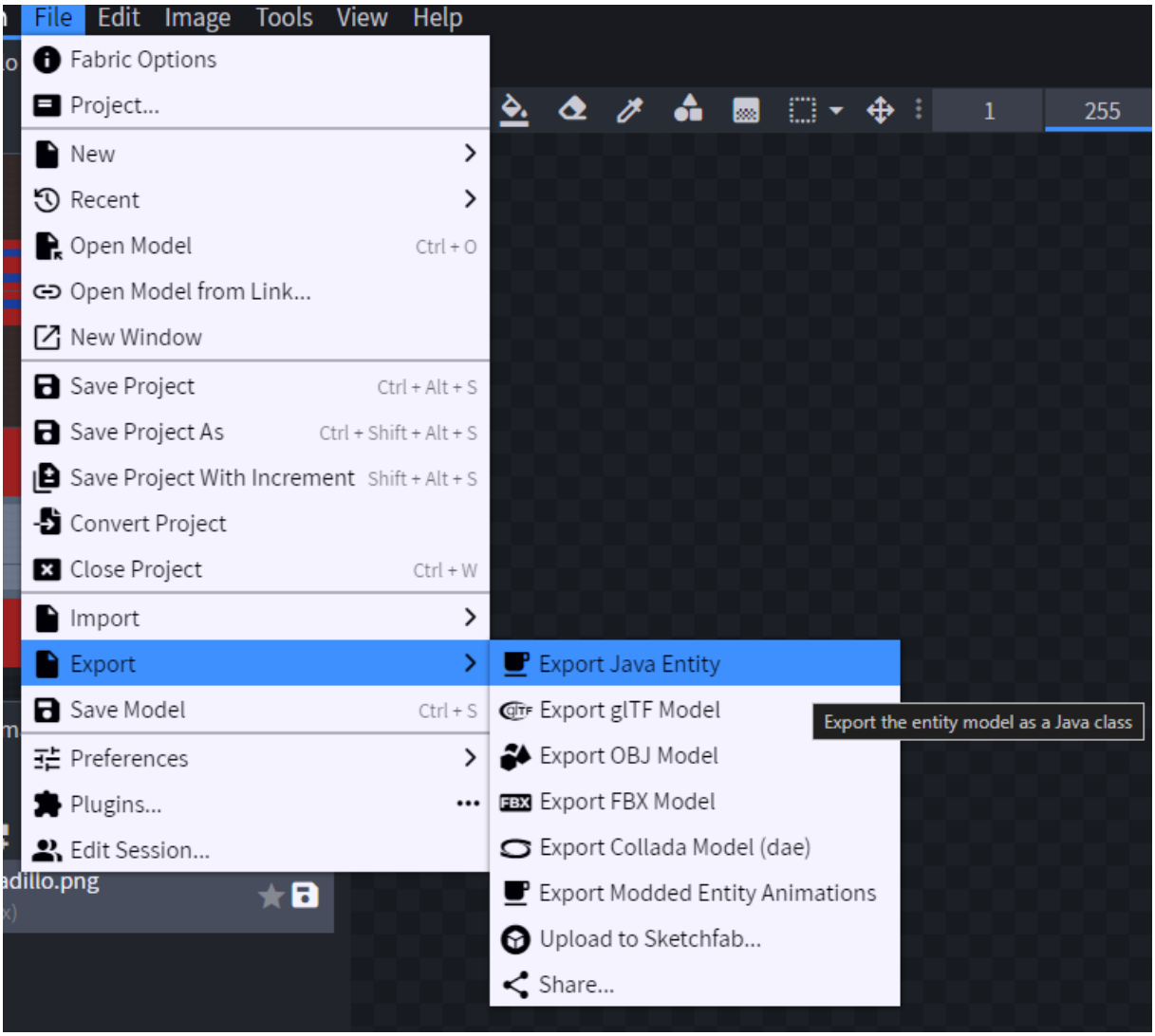
CAMP MINECRAFT AI

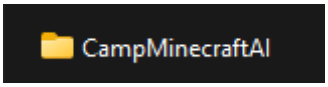
– Doku 2 –

Kapitel 3 – Export und Einbindung in den Code

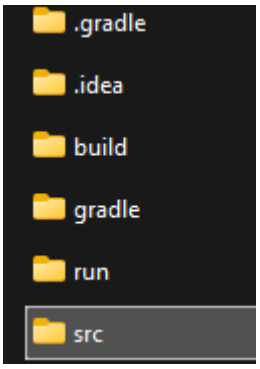
Damit wir unsere eben erstellte Entität jetzt auch im Spiel sehen können, müssen wir das Modell und die Textur zunächst in unser IntelliJ-Projekt einbinden. Dafür müssen wir die Datei, die wir am Anfang hoffentlich alle richtig erstellt haben, exportieren. Das machen wir unter „Datei“ (engl.: „File“), dann unter „Export“, und wählen dann „Export Java Entity“ aus:



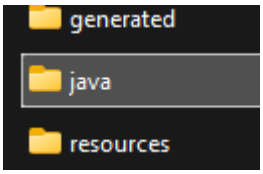
In dem folgenden Prompt wählen wir dann direkt den richtigen Speicherort aus, dieser wäre wie folgt:

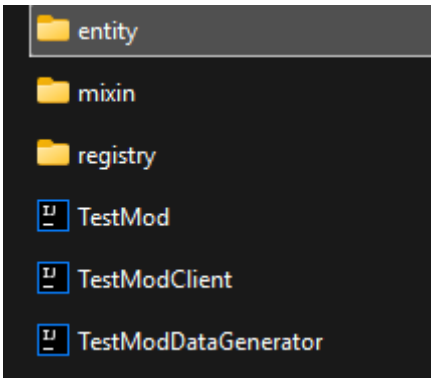


Unser Arbeitsordner für das Camp, zu finden auf dem Desktop →

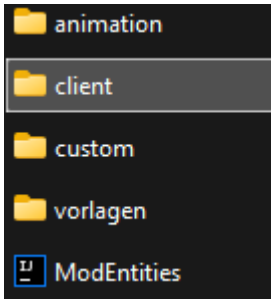


Der sogenannte „source“ Ordner, in dem sich der gesamte Code befindet, gefolgt von dem „main“ Ordner → danach suchen wir den „java“ Ordner, gefolgt von den Ordnern „net“, „forscherfreunde“ und „mod“ →





Dort angekommen suchen wir dann nach dem „entity“ Ordner, und wählen hier drinnen dann den Ordner „client“ aus.



In diesen Ordner speichern wir unsere Datei dann als:

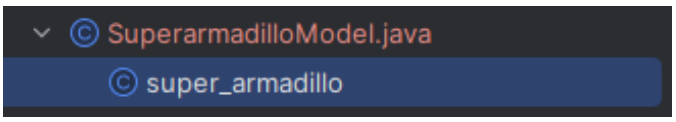
„NameModel“ ab.

WICHTIG: der Name wird zusammengeschrieben, mit einem großen Anfangsbuchstaben, und ohne Leerzeichen, gefolgt von dem groß geschriebenen Wort „Model“. Für das Beispiel meines Super-Amarillos wäre das dann also wie folgt:

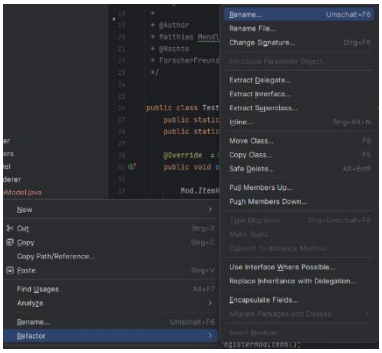
SuperarmadilloModel

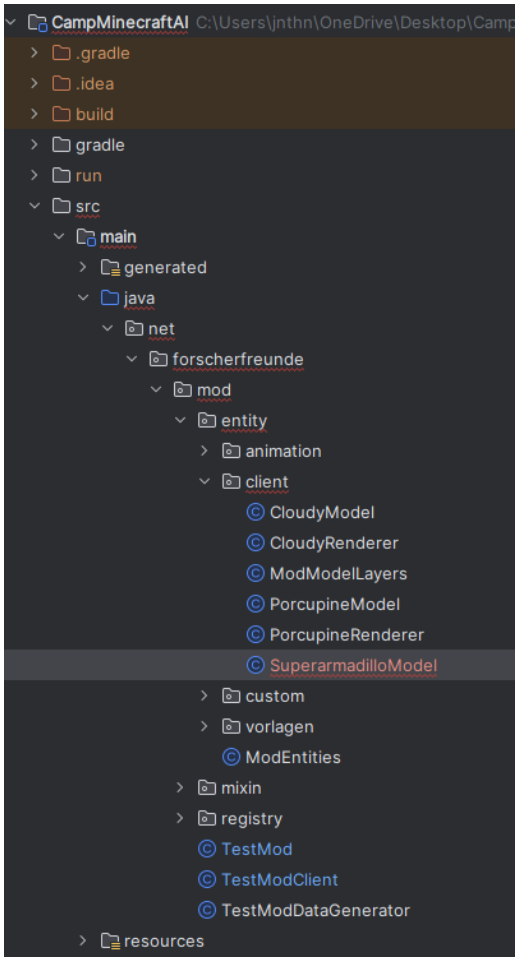
Diese Klasse finden wir dann auch direkt in IntelliJ wieder, wenn wir genau dieselben Ordner links in der Ordnerstruktur suchen und aufklappen →

Sollte es bei euch so aussehen:



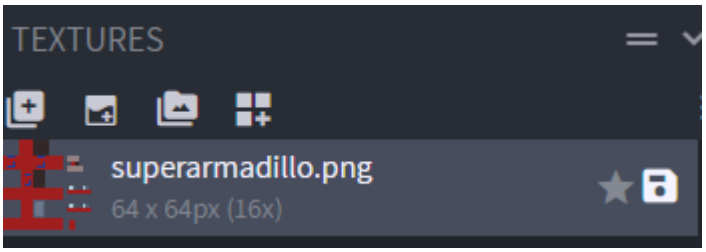
…dann müsst ihr einmal die untere der beiden Klassen auswählen, rechtsklicken darauf, und den Namen anpassen, so dass er mit der ersten der Beiden Klassen übereinstimmt. Das macht ihr über das Menu unter dem Punkt „refactor“ → „rename“:



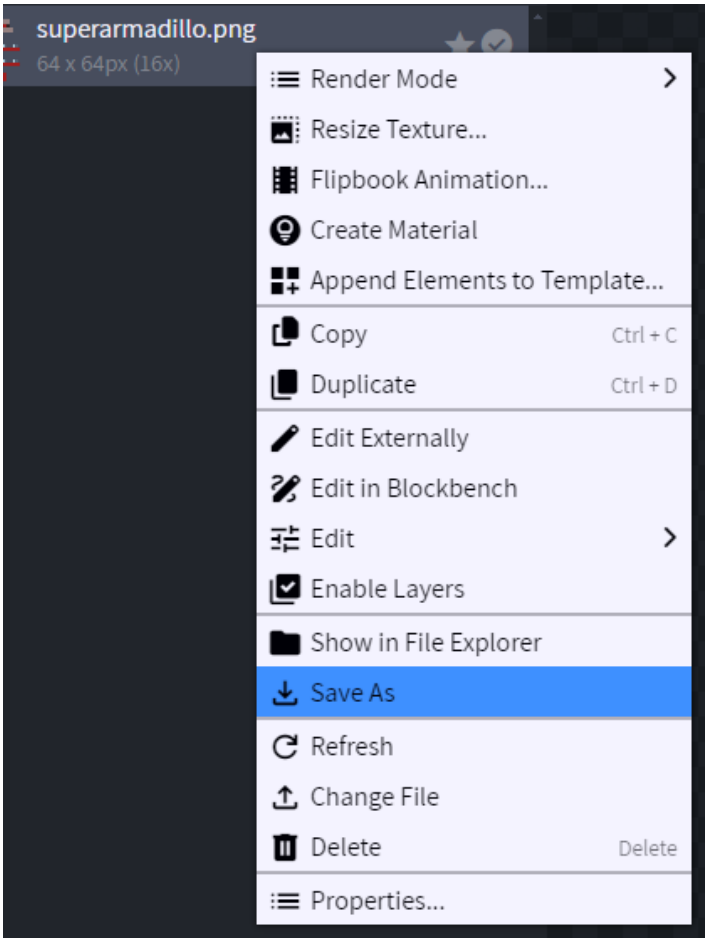


Als nächstes müssen wir noch die Textur Datei abspeichern, das Model allein sieht ohne die Textur im Spiel sonst ganz komisch aus….

Dafür speichern wir die Textur Datei, die wir in Blockbench auf der ganz linken Seite finden:



einfach im Texturen Ordner ab. Dafür gehen wir mit Rechtsklick auf die Textur Datei, und wählen im Menu dann „Speichern unter“ (engl.: „Save As“) aus:



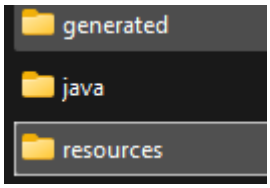
Dann speichern wir die Datei mit dem Namen im folgenden Format ab:

Alles kleingeschrieben, keine Leerzeichen, genau wie der Name der Entität.

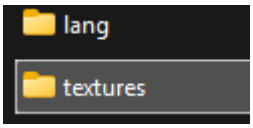
In meinem Beispiel also:

superarmadillo

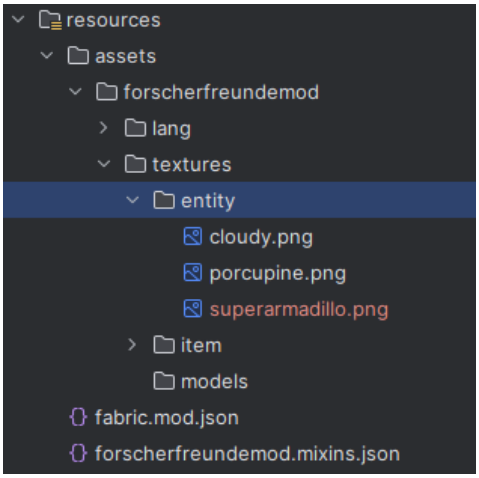
Diesmal suchen wir jedoch einen etwas anderen Ordner zum Abspeichern. Wir gehen denselben weg wie vorhin, bis wir bei dem Ordner „src“ ankommen, bei welchem wir diesmal jedoch den Ordner „resources“ auswählen:



In diesem Ordner gehen wir dann über die Ordner „assets“ und „forscherfreundemod“, zu dem Ordner „textures“:



Hier angekommen wählen wir zu guter Letzt dann den Ordner „entity“ aus, und speichern hier unsere Texturdatei mit dem richtigen Namen ab. Das sollte bei mir dann also so aussehen:



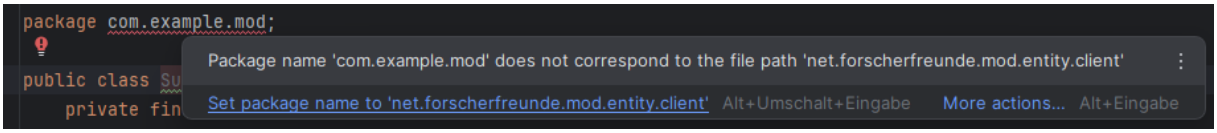
Ich kann nun also meine Texturdatei auch in IntelliJ ganz einfach wiederfinden, erneut in der linken Ordnerstruktur, wenn ich denselben Weg gehe, den ich beim Speichern gegangen bin, und die richtigen Ordner aufklappe.

Dabei stellt ihr fest, dass die „resources“ mit allen Ordnern die folgen, ganz unten in der Ordnerstruktur stehen normalerweise, also könnt ihr den Weg immer schnell finden.

Kapitel 3.1 – die letzten Schliffe an der Model-Klasse

Damit wir etwas mit der eben erstellten Model-Klasse von unserer Entität aber etwas anfangen können, müssen wir erst einmal die ganzen Fehler beheben, die sich da automatisch eingeschlichen haben in den Code. Dafür schauen wir uns die Klasse einmal gemeinsam an, und mit ein paar Klicks sollten keine rot unterstrichenen Codezeilen mehr vorhanden sein.

Dafür fangen wir ganz oben an, in Zeile 5 im Code, finden wir das sogenannte Package der Klasse. Dieses müssen wir einmal ändern, bzw. das macht IntelliJ netterweise für uns, wir müssen dem nur zustimmen. Indem wir über der roten Zeile Code mit der Maus schweben, können wir einfach direkt den Vorschlag von IntelliJ annehmen:



Als nächstes müssen wir einmal die Klasse etwas anpassen, denn da passen die Werte noch nicht so ganz. Wir wollen zunächst den Typenparameter der Klasse verändern, dafür schauen wir uns die folgende Zeile Code einmal an:

```
public class SuperarmadilloModel extends EntityModel<Entity> {
```

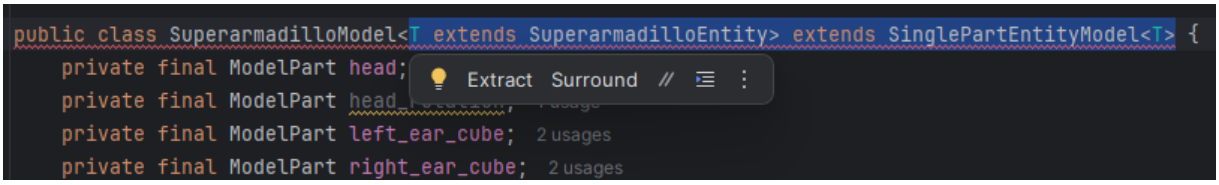
Und ändern eben diese Zeile Code etwas ab. Wir schreiben hinter unser SuperarmadilloModel einmal folgendes:

<T extends SuperarmadilloEntity>

und dann hinter das extends schreiben wir statt Entity folgendes:

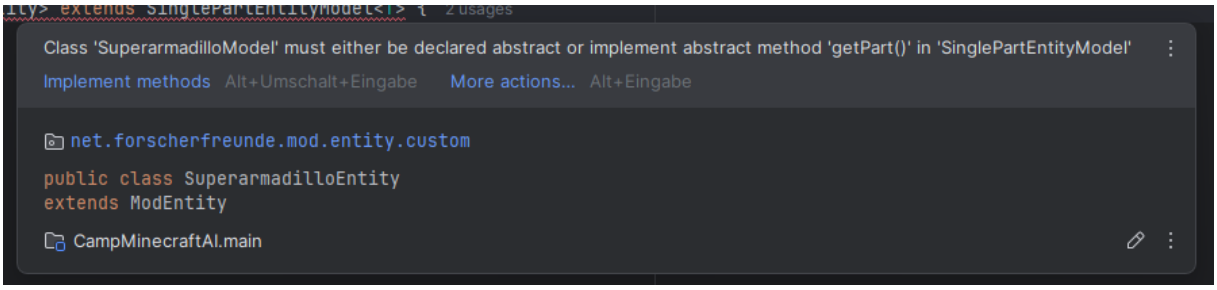
SinglePartEntityModel<T>

so dass es bei mir dann wie folgt aussehen sollte:

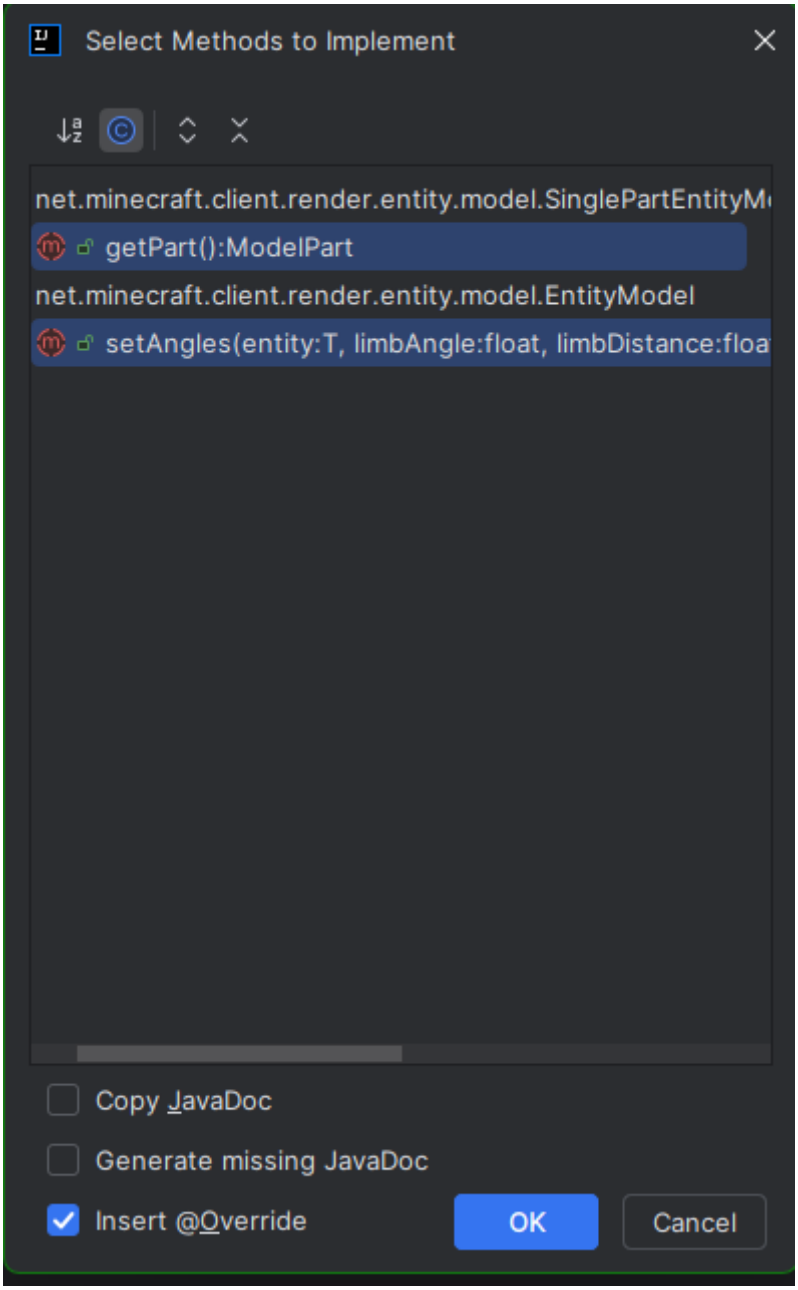


Dabei müsst ihr natürlich den Namen von eurer Entity- bzw. Modelklasse verwenden!

Im Anschluss müssen wir uns nun um die Einbindung der eben erfolgten Änderungen kümmern. Dafür gehen wir mit der Maus über unsere rot Unterstrichene Zeile Code, und fügen die Methoden ein die IntelliJ uns vorschlägt. Das sieht dann in etwa so aus:



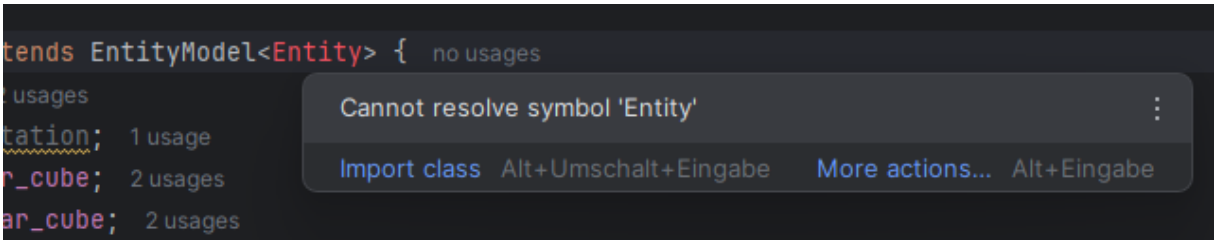
Nun klicken wir auf „Implement methods“, und klicken auf OK:



Im Anschluss müssen wir nur noch die alten Methoden, die wir nicht mehr brauchen löschen. Dafür markieren wir die Zeilen Code – achten darauf alle dazugehörigen Klammern und Semikolons zu markieren, und löschen den markierten Code Bereich:

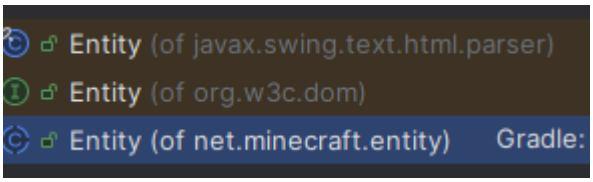
```
    ModelPartData tail = modelPartData.addChild("tail", ModelPartBuilder.create().uv(44, 53).cuboid(-0.5F, -0.0865F, 0.0933F, 1.0F, 6.0F, 1.0F, new Di
    return TexturedModelData.of(modelData, 64, 64);
}
@Override
public void setAngles(Entity entity, float limbSwing, float limbSwingAmount, float ageInTicks, float netHeadYaw, float headPitch) {
}
@Override
public void render(MatrixStack matrices, VertexConsumer vertexConsumer, int light, int overlay, float red, float green, float blue, float alpha) {
    head.render(matrices, vertexConsumer, light, overlay, red, green, blue, alpha);
```

Wenn wir das gemacht haben, nehmen wir uns den restlichen rot unterstrichenen Zeilen Code an – diese folgen aber alle demselben Prinzip von – mit der Maus drüber schweben, warten bis IntelliJ vorschlägt die Klasse zu importieren, das dann akzeptieren, und fertig ist die Klasse!



Die Vorgehensweise ist hier jetzt nur einmal beispielhaft für alle der Fehler in der Klasse. Einfach auf „Import class“ klicken, und fertig ist es.

Bei der „Entity“ Klasse ist nur zu beachten, dass es 3 Auswahlmöglichkeiten gibt. Die Richtige hier ist die Klasse, bei der „of net.minecraft.entity“ dahinter steht in Klammern:



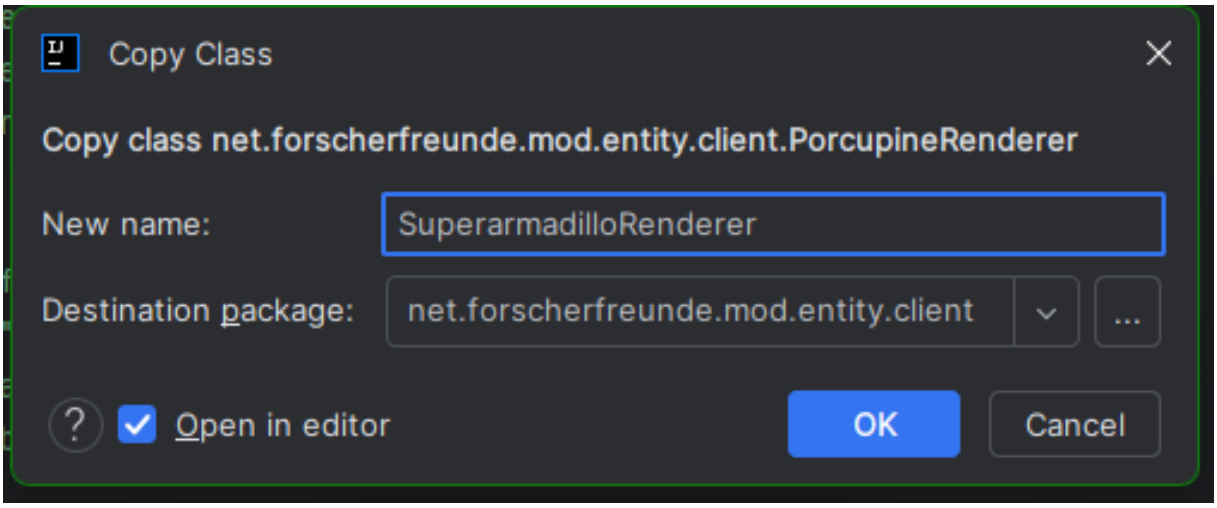
Wenn wir das alles gemacht haben, sollten keine Fehler mehr in der Klasse vorhanden sein, und wir können fortfahren!

Kapitel 3.2 – die Renderer-Klasse

Wir benötigen für unsere Entität nun auch noch eine Klasse, welche die Textur richtig auf das Modell lädt. Wir nennen das „Rendern“, und dafür erstellen wir uns nun eine eigene neue Klasse. Dabei sparen wir uns etwas Arbeit, indem wir uns einfach eine vorhandene Renderer-Klasse kopieren. Dafür halten wir die strg (oder engl.: ctrl) Taste auf der Tastatur gedrückt, und ziehen die „PorcupineRenderer“ Klasse einfach auf den Ordner mit dem Namen „client“. In dem sich öffnenden Fenster ändern wir dann den Namen auf unseren Namen mit „Renderer“ hinten dran, also in meinem Fall:

SuperarmadilloRenderer

Und klicken dann auf „OK“:



In der sich dann öffnenden Klasse müssen wir nun nur noch ein paar wenige Stellen anpassen, um unsere Renderer Klasse fertig zu haben, das machen wir aber in Ruhe, sobald wir die Entity Klasse erstellt haben.

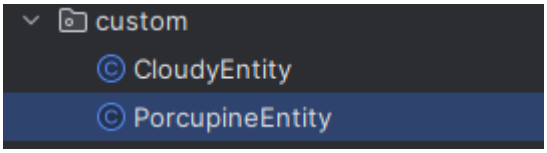
Und damit kommen wir zum vorletzten Schritt…

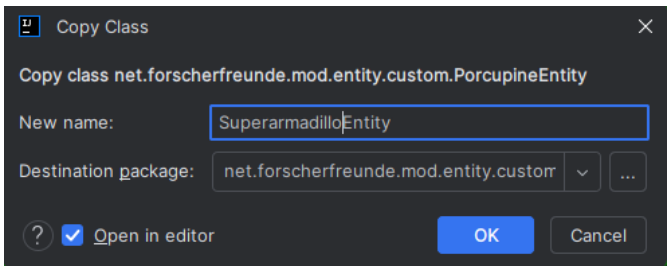
Kapitel 3.3 – die Entity-Klasse

Diese Klasse beinhaltet die KI der Entität – ist also aus Sicht des Spieles die wichtigste Klasse. Ohne Textur oder Renderer könnte die Entität zwar im Spiel existieren – nicht aber ohne die eigene Entity-Klasse.

Hier trennen sich nun die Wege von den Monstern und den friedlichen Tieren – Alle die ein Monster erstellt haben, können die Schritte genau gleich mitgehen, jedoch nehmen als Vorlage bitte die „CloudyEntity“-Klasse.

Auch hier sparen wir uns gewaltig Zeit, indem wir einfach eine bestehende Entity Klasse abändern. Dafür bedienen wir uns an der Vorlage der „PorcupineEntity“-Klasse, welche wir in dem Ordner „custom“ finden. Diese kopieren wir dann genau wie die Renderer-Klasse, jedoch ziehen wir sie diesmal in den Ordner „custom“ hinein während wir die strg-Taste gedrückt halten:





Nachdem wir hier erneut auf „OK“ geklickt haben, müssen wir in der Klasse Ein oder Zwei Kleinigkeiten anpassen.

Zuerst schauen wir uns dafür die Methode „createChild()“ an. In dieser müssen wir das „pocupine“ in Grün hinterlegt, gegen den Namen unserer eigenen Entität austauschen:

```
@Override  new *
public PassiveEntity createChild(ServerWorld world, PassiveEntity entity) {
    return (PassiveEntity) ModEntities.ModEntitiesMap.get("porcupine").create(world);
}
```

Ich schreibe bei mir also:

```
@Override  new *
public PassiveEntity createChild(ServerWorld world, PassiveEntity entity) {
    return (PassiveEntity) ModEntities.ModEntitiesMap.get("superarmadillo").create(world);
}
```

Außerdem passen wir hier jetzt auch weitere Faktoren unserer Entität an – so wie das Paarungs- und Verführungsitem und natürlich das Verhalten der KI durch die Ziele.

Wenn ihr die Grundattribute anpassen wollt, wie die Gesundheit, Geschwindigkeit, Rüstung und Schaden, so könnt ihr das hier auch tun. Das schauen wir uns aber in einer eigenen Doku separat an, die den passenden Namen „Entity_Attribute“ trägt.

Nun da wir die Entity-Klasse erstellt haben, gehen wir einmal zurück, um die Renderer-Klasse zu beenden.

Kapitel 3.4 – der letzte Schliff

Zurück in unserer Renderer-Klasse, wollen wir nun einmal die angesprochenen Änderungen vornehmen. Starten tuen wir dabei in der Codezeile, welche die Klasse an sich definiert. Das wäre folgende Zeile:

```
public class SuperarmadilloRenderer extends MobEntityRenderer<PorcupineEntity, PorcupineModel<PorcupineEntity>> {  no usages  new *
```

In dieser Zeile Code ändern wir den oben markierten Teil einmal ab, nämlich zu unserer eigenen Entity Klasse als Referenz. Das gleiche machen wir bei allen folgenden Stellen in der Klasse, immer darauf achtend, ob wir unsere erstellte Entity-Klasse oder die Model-Klasse referenzieren sollen. Dabei schauen wir einfach, welche der Klassen von dem Porcupine hier referenziert wird. In meinem Fall schreibe ich hier an der ersten markierten Stelle also

ArmadilloEntity

hin.

Die folgenden Stellen müssen wir noch anpassen in der Klasse:

```
public class SuperarmadilloRenderer extends MobEntityRenderer<PorcupineEntity, PorcupineModel<PorcupineEntity>> {  no usages  new *
```

```
public class SuperarmadilloRenderer extends MobEntityRenderer<PorcupineEntity, PorcupineModel<PorcupineEntity>> {  no usages  new *
```

```
private static final Identifier TEXTURE = new Identifier(TestMod.MOD_ID,  path: "textures/entity/superarmadillo.png");  1 usa
public SuperarmadilloRenderer(EntityRendererFactory.Context context) {  no usages  new *
    super(context, new PorcupineModel<>(context.getPart(ModModelLayers.EntityModels.get("porcupine_model"))),  f: 0.6f);
}
```

```
@Override  no usages  new *
public Identifier getTexture(PorcupineEntity entity) { return TEXTURE; }
```

```java
@Override  no usages  new *
public void render(PorcupineEntity mobEntity, float f, float g, MatrixStack matrixStack,
                   VertexConsumerProvider vertexConsumerProvider, int i) {

    if (mobEntity.isBaby()) {
        matrixStack.scale( x: 0.5f,  y: 0.5f,  z: 0.5f);
    }else {
        matrixStack.scale( x: 1f,  y: 1f,  z: 1f);
    }

    super.render(mobEntity, f, g, matrixStack, vertexConsumerProvider, i);
}
```

Die einzigen Veränderungen, die sich etwas anders verhalten, sind folgende Zeilen:

```java
private static final Identifier TEXTURE = new Identifier(TestMod.MOD_ID,  path: "textures/entity/superarmadillo.png");  1 usage
public SuperarmadilloRenderer(EntityRendererFactory.Context context) {  no usages  new *
    super(context, new PorcupineModel<>(context.getPart(ModModelLayers.EntityModels.get("porcupine_model"))),  f: 0.6f);
}
```

In dieser Zeile schreiben wir einfach den Namen unserer Entity, gefolgt von „_model“ hin, also in meinem Beispiel:

Superarmadillo_model

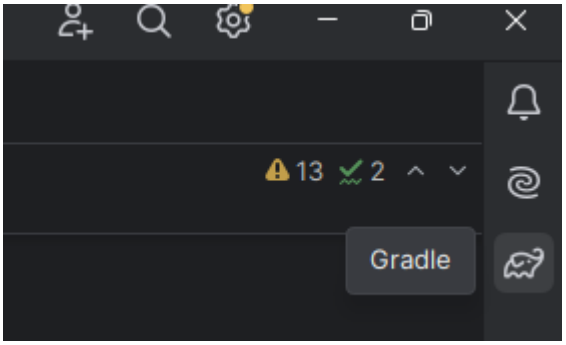
```java
private static final Identifier TEXTURE = new Identifier(TestMod.MOD_ID,  path: "textures/entity/superarmadillo.png");  1 usage
public SuperarmadilloRenderer(EntityRendererFactory.Context context) {  no usages  new *
    super(context, new PorcupineModel<>(context.getPart(ModModelLayers.EntityModels.get("porcupine_model"))),  f: 0.6f);
}
```

In dieser Zeile schreiben wir den Namen der Texturdatei hin, die wir ja in Kapitel 3 ganz am Anfang benannt haben. Dabei habt ihr hoffentlich alle darauf geachtet, den Namen der Entity zu verwenden, alles kleingeschrieben, ohne Leerzeichen oder sonstige Sonderzeichen. Dann sieht das bei mir wie folgt aus:

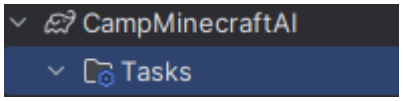
superarmadillo

gefolgt von der Endung .png so wie sie bereits dort steht.

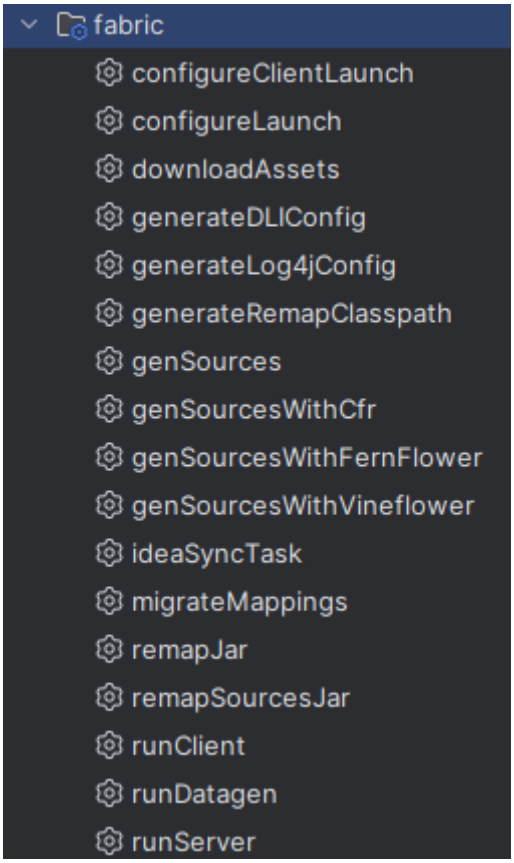
Wenn wir das alles gemacht haben, können wir uns unsere Entity einmal im Spiel anschauen, ob alles richtig funktioniert hat. Dafür starten wir einmal Minecraft über unsere Schnittstelle. Das machen wir über das Menü auf der rechten Seite mit dem Namen „gradle“



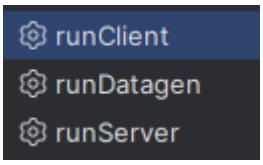
wählen hier dann einmal den Menüpunkt „Tasks“ aus



Dort dann einmal „fabric“



Und wählen hier per Doppelklick den Menüpunkt „runClient“ aus.



Dabei ist es wichtig, dass ihr mit dem Internet verbunden seid.

Sollte es bei dem Starten von Minecraft zu Fehlern kommen, das Spiel abstürzen oder gar nicht erst starten, dann sprecht bitte eure Teamer an, und schaut vor allem, ob alle Klassen richtig benannt sind, alle Referenzen auf die Klassen korrekt geschrieben sind, ihr auf die Groß bzw. Kleinschreibung an den relevanten Stellen geachtet habt, und ihr die Dateien, wie Textur, auch richtig abgespeichert habt.

Wenn ihr alle Schritte sorgfältig abgearbeitet habt, dann müsste sich Minecraft öffnen.

In dem aktuellen Zustand können wir zwar Minecraft öffnen, und das bedeutet, dass unsere Klassen funktionieren fürs erste, aber unsere Entity ist immer noch nicht im Spiel registriert. Dafür schauen wir uns in der nächsten Doku an, wie das funktioniert!