**CubeIt – generování objektu Item**

# Raw code (Kotlin)

## Generování itemu

fun generateItem(playerG: Player, inQuality: Int? = null, itemType: ItemType? = null, itemSlot: Int? = null, itemLevel: Int? = null): Item? {  
  
 val tempArray: MutableList<Item?> = returnItem(playerG, itemType, itemSlot)  
 if(tempArray.size == 0){  
 return null  
 }  
  
 val itemReturned = tempArray[Random.nextInt(0, tempArray.size)]  
 val itemTemp: Item? = when (itemReturned?.type) {  
 ItemType.Weapon -> Weapon(  
 name = itemReturned.name,  
 type = itemReturned.type,  
 charClass = itemReturned.charClass,  
 description = itemReturned.description,  
 levelRq = itemReturned.levelRq,  
 drawableIn = *getKey*(*drawableStorage*, itemReturned.drawable)!!,  
 slot = itemReturned.slot  
 )  
 ItemType.Wearable -> Wearable(  
 name = itemReturned.name,  
 type = itemReturned.type,  
 charClass = itemReturned.charClass,  
 description = itemReturned.description,  
 levelRq = itemReturned.levelRq,  
 drawableIn = *getKey*(*drawableStorage*, itemReturned.drawable)!!,  
 slot = itemReturned.slot  
 )  
 ItemType.Runes -> Runes(  
 name = itemReturned.name,  
 type = itemReturned.type,  
 charClass = itemReturned.charClass,  
 description = itemReturned.description,  
 levelRq = itemReturned.levelRq,  
 drawableIn = *getKey*(*drawableStorage*, itemReturned.drawable)!!,  
 slot = itemReturned.slot  
 )  
 else -> Item(name = itemReturned!!.name, type = itemReturned.type, charClass = itemReturned.charClass, description = itemReturned.description, levelRq = itemReturned.levelRq, drawableIn = *getKey*(*drawableStorage*, itemReturned.drawable)!!, slot = itemReturned.slot)  
 }  
 itemTemp!!.levelRq = itemLevel ?: Random.nextInt(playerG.level - 2, playerG.level + 1)  
 if (inQuality == null) {  
 itemTemp.quality = when (Random.nextInt(0, (GenericDB.balance.itemQualityPerc["7"] ?: *error*("")) + 1)) { //quality of an item by percentage  
 in 0 *until* (GenericDB.balance.itemQualityPerc["0"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["0"] ?: *error*("") //39,03%  
 in (GenericDB.balance.itemQualityPerc["0"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["1"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["1"] ?: *error*("") //27%  
 in (GenericDB.balance.itemQualityPerc["1"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["2"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["2"] ?: *error*("") //20%  
 in (GenericDB.balance.itemQualityPerc["2"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["3"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["3"] ?: *error*("") //8,41%  
 in (GenericDB.balance.itemQualityPerc["3"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["4"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["4"] ?: *error*("") //5%  
 in (GenericDB.balance.itemQualityPerc["4"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["5"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["5"] ?: *error*("") //0,5%  
 in (GenericDB.balance.itemQualityPerc["5"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["6"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["6"] ?: *error*("") //0,08%  
 in (GenericDB.balance.itemQualityPerc["6"] ?: *error*("")) + 1 *until* (GenericDB.balance.itemQualityPerc["7"] ?: *error*("")) -> GenericDB.balance.itemQualityGenImpact["7"] ?: *error*("") //0,01%  
 else -> 0  
 }  
 } else {  
 itemTemp.quality = kotlin.math.*min*(7, inQuality)  
 }  
  
 if (itemTemp.levelRq < 1) itemTemp.levelRq = 1  
 var points = Random.nextInt(itemTemp.levelRq \* 3 \* (itemTemp.quality + 1), itemTemp.levelRq \* 3 \* ((itemTemp.quality \* 1.25).toInt() + 2))  
 var pointsTemp: Int  
 itemTemp.priceCubeCoins = points  
 val numberOfStats = Random.nextInt(1, 9)  
 for (i in 0..numberOfStats) {  
 pointsTemp = Random.nextInt(points / (numberOfStats \* 2), points / numberOfStats + 1)  
 when (itemTemp) {  
 is Weapon -> {  
 when (Random.nextInt(0, if (playerG.charClass.lifeSteal) 4 else 3)) {  
 0 -> {  
 itemTemp.power += (pointsTemp \* (GenericDB.balance.itemGenRatio["Power"] ?: 1.0)).toInt()  
 }  
 1 -> {  
 itemTemp.block += (pointsTemp \* (GenericDB.balance.itemGenRatio["Block"] ?: 1.0)).toInt()  
 }  
 2 -> {  
 itemTemp.dmgOverTime += (pointsTemp \* (GenericDB.balance.itemGenRatio["DamageOverTime"] ?: 1.0)).toInt()  
 }  
 3 -> {  
 itemTemp.lifeSteal += (pointsTemp \* (GenericDB.balance.itemGenRatio["LifeSteal"] ?: 1.0)).toInt()  
 }  
 }  
 }  
 is Wearable -> {  
 when (Random.nextInt(0, 4)) {  
 0 -> {  
 itemTemp.armor += (pointsTemp \* (GenericDB.balance.itemGenRatio["Armor"] ?: 1.0)).toInt()  
 }  
 1 -> {  
 itemTemp.block += (pointsTemp \* (GenericDB.balance.itemGenRatio["Block"] ?: 1.0)).toInt()  
 }  
 2 -> {  
 itemTemp.health += (pointsTemp \* (GenericDB.balance.itemGenRatio["Health"] ?: 1.0)).toInt()  
 }  
 3 -> {  
 itemTemp.energy += (pointsTemp \* (GenericDB.balance.itemGenRatio["Energy"] ?: 1.0)).toInt()  
 }  
 }  
 }  
 is Runes -> {  
 when (Random.nextInt(0, 4)) {  
 0 -> {  
 itemTemp.armor += (pointsTemp \* (GenericDB.balance.itemGenRatio["Armor"] ?: 1.0)).toInt()  
 }  
 1 -> {  
 itemTemp.health += (pointsTemp \* (GenericDB.balance.itemGenRatio["Health"] ?: 1.0)).toInt()  
 }  
 2 -> {  
 itemTemp.adventureSpeed += (pointsTemp \* (GenericDB.balance.itemGenRatio["AdventureSpeed"] ?: 1.0)).toInt()  
 }  
 3 -> {  
 if((itemTemp.inventorySlots + (pointsTemp \* (GenericDB.balance.itemGenRatio["InventorySlots"] ?: 1.0)).toInt()) > playerG.level / 2){ //limit the inventory slots by player's level  
 points += pointsTemp - kotlin.math.*abs*(((itemTemp.inventorySlots - (playerG.level / 2)) / (GenericDB.balance.itemGenRatio["InventorySlots"] ?: 1.0)).toInt())  
 itemTemp.inventorySlots += kotlin.math.*abs*(itemTemp.inventorySlots - (playerG.level / 2))  
 }else {  
 itemTemp.inventorySlots += (pointsTemp \* (GenericDB.balance.itemGenRatio["InventorySlots"] ?: 1.0)).toInt()  
 }  
 }  
 }  
 }  
 }  
 points -= pointsTemp  
 }  
 return itemTemp  
}

## Filtr všech existujících itemů na ty, které jsou povoleny pro daného hráče

fun returnItem(playerX: Player, itemType: ItemType? = null, itemSlot: Int? = null): MutableList<Item?> {  
 val allItems: MutableList<Item?> = (playerX.charClass.itemList.*asSequence*().*plus*(playerX.charClass.itemListUniversal.*asSequence*())).*toMutableList*()  
 //val allowedItems: MutableList<Item?> = mutableListOf()  
  
 //allItems.sortWith(compareBy { it!!.levelRq })  
 allItems.*retainAll* **{** playerX.level > **it**!!.levelRq - GenericDB.balance.itemLvlGenBottom && playerX.level < it.levelRq + GenericDB.balance.itemLvlGenTop**}** if(itemType != null){  
 allItems.*retainAll* **{ it**!!.type == itemType**}** }  
 if(itemSlot != null){  
 allItems.*retainAll* **{ it**!!.slot == itemSlot**}** }  
  
 return allItems  
}

# Logický postup generování itemů

**Vyfiltrovat** list všech itemů tak, aby odpovídal pravidlům hry (balance – GenericDB/Balance) – tedy aktuálně jen podle levelu.

**Ošetřit** případné chybné situace – například kdyby filtr vrátil prázdný list. Aktuálně to znamená, že se nevrátí žádný předmět, nicméně bych v budoucnu udělal nějaký „default item“.

Z vyfiltrovaného listu náhodně vybereme 1.

**Otypovat** – zjištění (Wearable, Weapon, Runes) a následné otypování vráceného předmětu, to v serverové funkci teoreticky není potřeba, nicméně se stále musíme řídit dle typů itemů pro následné úpravy.

**Generování klíčových parametrů**:

„levelRq“ – v kódu se k tomuto parametru přistupuje jako k levelu předmětu, tedy se podle něj generuje počet bodového ohodnocení

„quality“ – obdoba „levelRq“ s rozdílem toho, že je tento parametr staticky generován – dle pravidel generování (balance – GenericDB/Balance) a aplikace ho nikdy nebude dynamicky měnit („levelRq“ prozatím také ne, ale počítáme s tím, že se může v budoucnu změnit), tudíž level hráče nehraje roli, pouze centralizované pravděpodobnosti (generovaní obtížností – difficulty je všude stejné, prostě jen použijte mapu pravděpodobností z herních pravidel, stejně jako je v raw kódu).

Points / počet bodového ohodnocení – počet bodů, kterých se následně rozdělí do specifických atributů itemu (například: 500 bodů celkově, 400 se dalo do „power“ a 100 do „block“)

Počet atributů itemu, mezi kterými se budou rozdělovat points / body.

**Přiřazení bodů** – některé atributy jsou limitovány podle typu charakteru hráče (aktuálně pouze vampire – charclass/lifeSteal – boolean) – tedy zdali se některé z atributů smí, či nesmí generovat. Body přidělené k danému atributu jsou ještě násobeny číslem získaným z pravidel generování (balance – GenericDB/Balance) viz. Raw code: pointsTemp \* (GenericDB.balance.itemGenRatio["Block"]