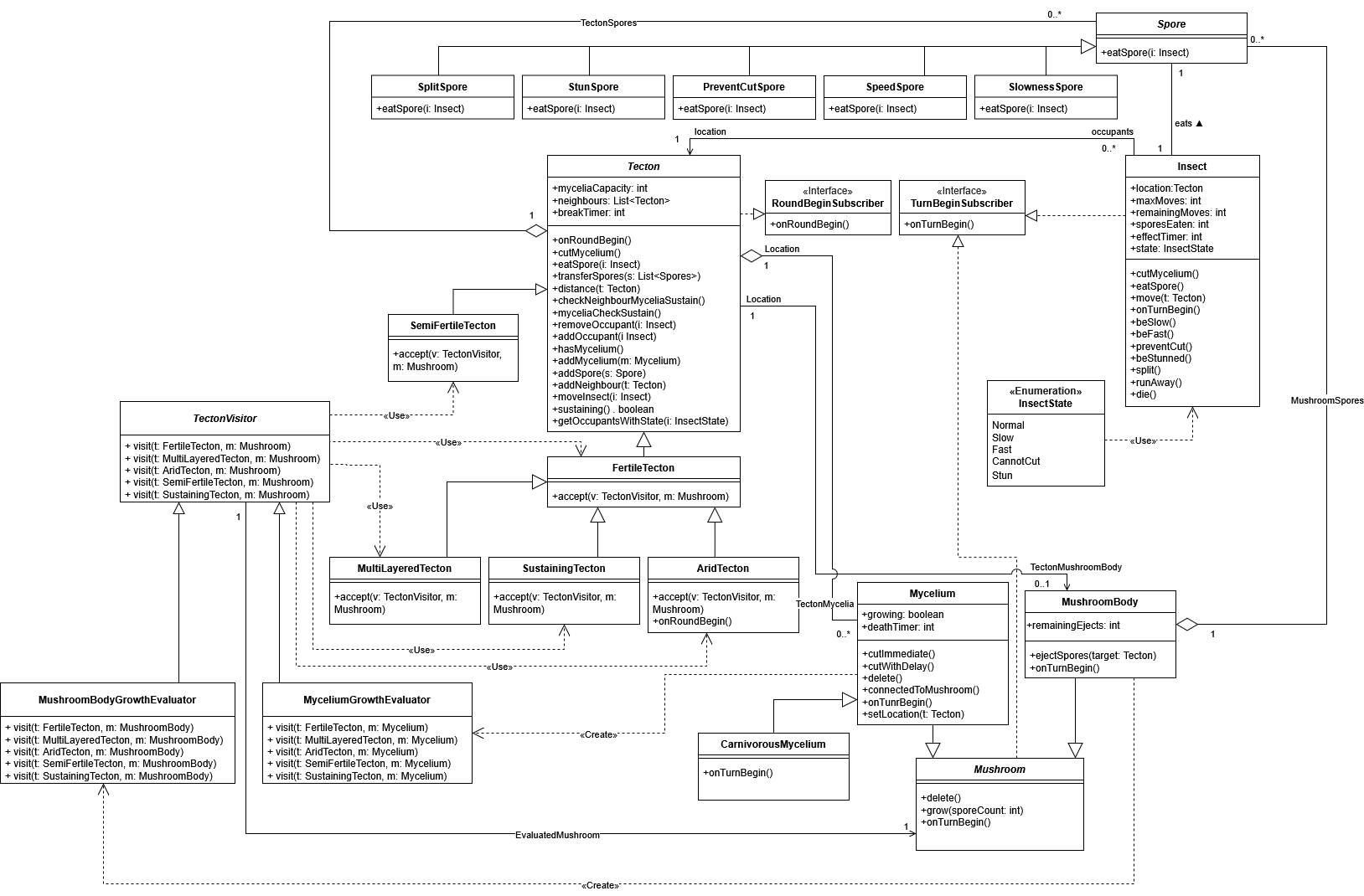
# Prototípus koncepciója

## Változás hatása a modellre

### Módosult osztálydiagram



### Új vagy megváltozó metódusok

#### Mycelium – cutImmidiate()

Azonnal elvágja a gombafonalat.

#### Mycelium – cutWithDelay()

Egy gombafonál típustól függő idő után elvágja a gombafonalat.

#### Tecton - getOccupantsWithState(i: InsectState)

Azon rovaroknak a listáját hozza létre, amelyek a tektonon helyezkednek el, és adott állapotban vannak.

#### Tecton - sustaining()

Megadja, hogy a tekton, vagy a tektonon elhelyezkedő objektumok képesek-e egy gombafonalat életben tartani.

#### TectonVisitor - visit(t: SustainingTecton, m: Mushroom)

A visitor működésének kiterjesztése az új tekton fajtára.

#### Insect – die()

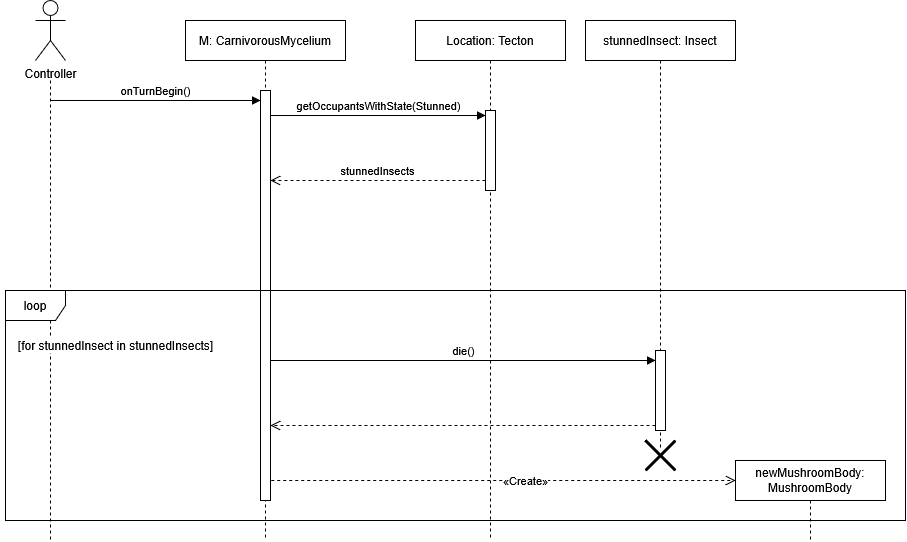
A rovar elpusztul, azaz eltűnik a játékmezőről.

#### Insect – split()

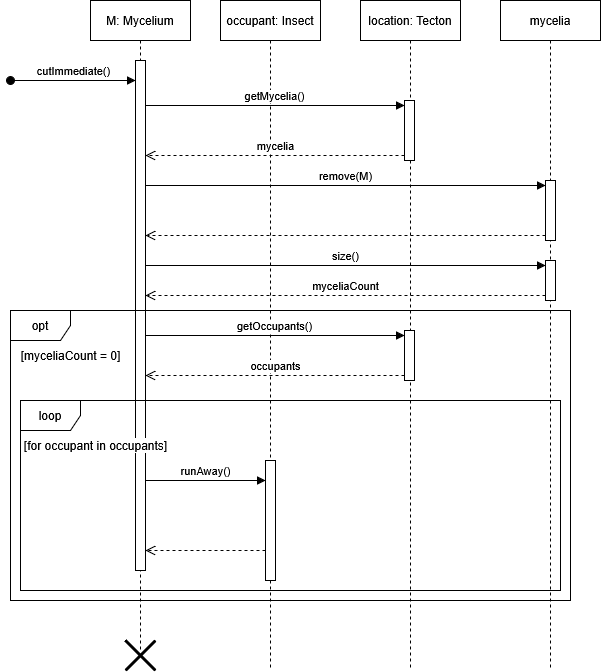
A rovar csinál egy másolatot saját magából.

### Szekvencia-diagramok

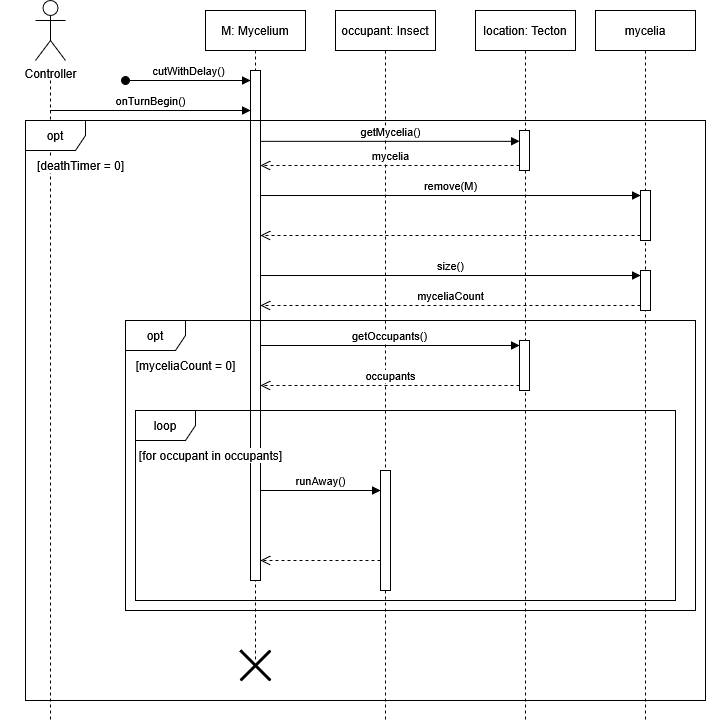
#### Húsevő gombafonál



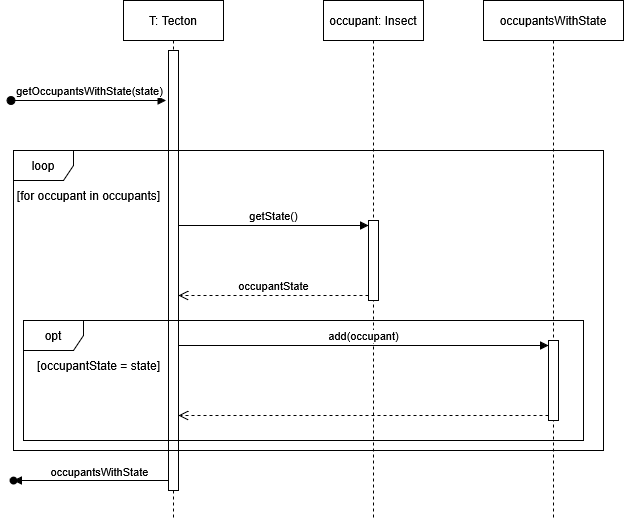
#### Azonnali fonálelvágás



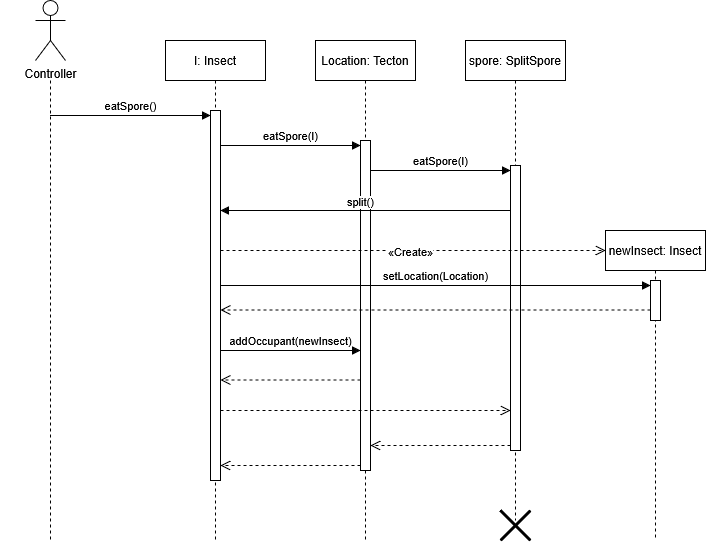
#### Késeltetett fonal elvágás



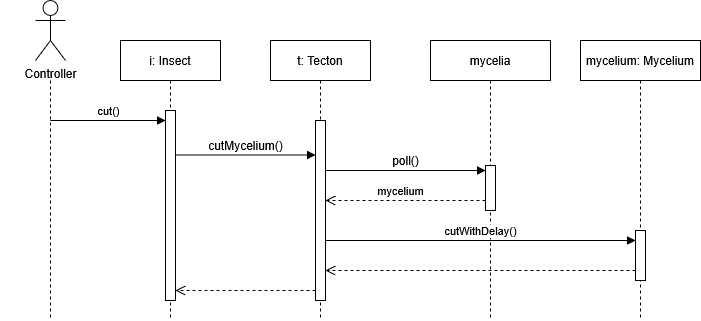
#### getOccupantsWithState szekvencia



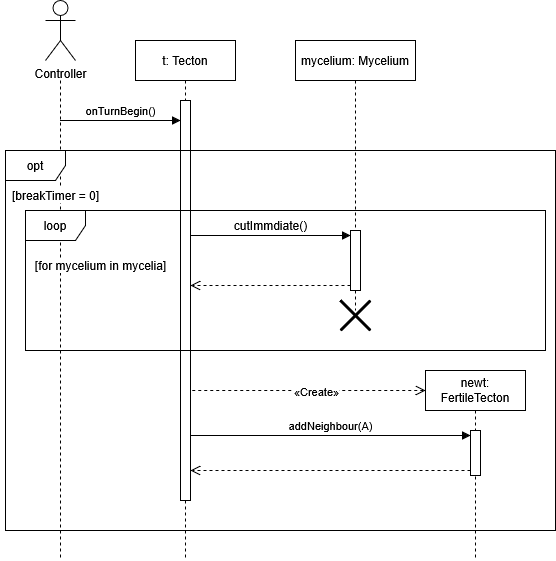
#### „Split” típusú spóra elfogyasztása



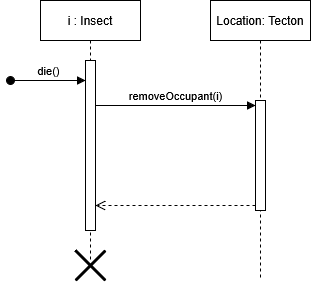
#### Rovar általi gombafonál elvágás



#### Tektontörés



#### Rovar elpusztulása



## Prototípus interface-definíciója

[Definiálni kell a teszteket leíró nyelvet. Külön figyelmet kell fordítani arra, hogy ha a rendszer véletlen elemeket is tartalmaz, akkor a véletlenszerűség ki-bekapcsolható legyen, és a program determinisztikusan is tesztelhető legyen.]

### Az interfész általános leírása

[A protó (karakteres) input és output felületeit úgy kell kialakítani, hogy az input fájlból is vehető legyen illetőleg az output fájlba menthető legyen, vagyis kommunikációra csak a szabványos be- és kimenet használható.]

A felhasználó a program irányítását parancsokkal végzi. Ezeknek a parancsoknak vannak argumentumai, melyeket a parancs után tud a játékos beleírni. A parancsok maguktól nem adnak outputot. Ha egy parancs egy részében felhasználói választásra van szükség, akkor a program ki tud kérdezni a felhasználóhoz. A program futása során minden objektumnak van egy saját, egyedi neve, ami alapján az objektum egyértelműen azonosítható.

### Bemeneti nyelv

[Definiálni kell a teszteket leíró nyelvet (szintakszis és szemantika). Külön figyelmet kell fordítani arra, hogy ha a rendszer véletlen elemeket is tartalmaz, akkor a véletlenszerűség ki-bekapcsolható legyen, és a program determinisztikusan is futtatható legyen. A szálkezelést is tesztelhető, irányítható módon kell megoldani. A programot egy adott konfigurációból is el kell tudni indítani, vagyis kell olyan parancs, amivel konkrét előre megadott állapotból indul a rendszer (pl. load).]

Egy parancs általános szintaxisa:

<UTASÍTÁS> <PARAMÉTER> {<LISTA\_ELEM\_1> <LISTA\_ELEM\_2> <LISTA\_ELEM\_N>}

Tehát a parancsnak megadjuk a nevét, (utasítás), majd utána, ha szükséges, szóközzel elválasztva a parancs paramétereit megadhatjuk. Ha a parancs paramétere egy gyűjtemény lenne, akkor a gyűjtemény kezdetét és végét kapcsos zárójellel jelöljük. Az egyes elemek szóközzel vannak elválasztva.

Ha egy parancsnak felhasználói inputra van szüksége, akkor azt a program az alábbi módon jelzi:

<< 1:<OPCIÓ\_1> 2:<OPCIÓ\_2> N:<OPCIÓ\_N>

Azaz, a „<<” karakterekkel jelöli, hogy éppen bemenetre van szükség, az opciókat pedig a sorszámukkal együtt kiírja a program. A felhasználó választani úgy tud, hogyha a választott opció sorszámot megadja.

Minden a MACRO\_CASE elnevezési konvenciót használja, azaz minden betű nagybetű, és a szavak alsókötőjel ( \_ ) karakterrel vannak elválasztva.

A parancsok listája

**STATE**

**Leírás:** A kiválasztott objektum állapotát a megadott formátumban kiírja.

**Opciók:** A kiírandó objektum.

**Parancs2**

**Leírás:**

**Opciók:**

[Ha szükséges, meg kell adni a konfigurációs (pl. pályaképet megadó) fájlok nyelvtanát is.]

### Kimeneti nyelv

[Egyértelműen definiálni kell, hogy az egyes bemeneti parancsok végrehajtása után előálló állapot milyen formában jelenik meg a szabványos kimeneten. A program képes legyen olyan kimenetet előállítani, amellyel az objektumok állapota ellenőrizhető (pl. save). Ebben az alfejezetben is precízen definiálni kell, hogy a kimenet nyelve milyen elemekből és milyen szintakszissal áll elő.]

Egy objektum állapotát a STATE paranccsal lehet elérni. A STATE parancsnak és kimenetelének szintaxisa az alábbi:

STATE <OBJEKTUM>

<OBJEKTUM> <OBJETUM TÍPUSA>

<tagváltozó neve> <TAGVÁLTOZÓ TÍPUSA> = <TAGVÁLTOZÓ ÉRTÉKE>

<lista NEVE> <LISTA TÍPUSA> = {

elem1

elem2

elemN

}

Azaz először az objektum nevét, majd típusát írja ki. Majd az adott objektum alá tartozó tagváltozók egy tabulátorral beljebb kerülnek.

Az egyszerű tagváltozókat úgy írjuk ki, hogy először a nevét, majd típusát írjuk ki, majd egy egyenlőségjel után az értékét.

A gyűjtemény típusú tagváltozókat a gyűjtemény neve, utána a gyűjtemény típusa, majd egy egyenlőségjellel elválasztva az értéke, ahol a lista elejét és végét kapcsos zárjelek jelölik, és minden elem neve fel van sorolva, úgy, hogy mindegyik elem külön sorban van.

## Összes részletes use-case

[A use-case-eknek a részletezettsége feleljen meg a kezelői felületnek, azaz a felület elemeire kell hivatkozniuk a bemeneti nyelv parancsai alapján.

Alábbi táblázat minden use-case-hez külön-külön.]

|  |  |
| --- | --- |
| **Use-case neve** | Állapot kiírása |
| **Rövid leírás** | A felhasználó a kiválasztott objektum állapotát kiíratja. |
| **Aktorok** | User |
| **Forgatókönyv** | 1. Az objektum kiválasztása 2. Az objektum állapota kiíródik. |

## Tesztelési terv

[A tesztelési tervben definiálni kell, hogy a be- és kimeneti fájlok egybevetésével miként végezhető el a program tesztelése. Meg kell adni magas szintű teszt forgatókönyveket. Az egyes teszteket elég informálisan, szabad szövegként leírni, tesztesetenként egy-öt mondatban. Minden teszthez meg kell adni, hogy mi a célja, a proto mely funkcionalitását, osztályait stb. teszteli. Az alábbi táblázat minden teszt-esethez külön-külön elkészítendő.]

|  |  |
| --- | --- |
| **Teszt-eset neve** |  |
| **Rövid leírás** |  |
| **Teszt célja** |  |

## Tesztelést támogató segéd- és fordítóprogramok specifikálása

[Rövid bemutatással (elvárt funkcionalitás) specifikálni kell a tesztelést támogató segédprogramokat.]

A teszteléshez JUnit 5[[1]](#footnote-1) test framework-öt fogunk használni.

## Napló

|  |  |  |  |
| --- | --- | --- | --- |
| **Kezdet** | **Időtartam** | **Résztvevők** | **Leírás** |
| 2025.03.27. 12:00 | 2 óra | Kohár | Tevékenység:   * Az új funkciók implementálása az osztály diagrammba * Az új funkciók szekvencia diagrammjának elkészítése |
| 2025.03.27. 22:00 | 1 óra | Kohár | Tevékenység:   * Osztálydiagramm és szekvencia diagrammok javítása. |
| 2025.03.28. 20:00 | 1 óra | Kohár | Tevekénység:   * Kimeneti nyelv definiálása * Osztálydiagrammok exportálása |
| 2025. 03.29. 19:00 | 30 perc | Kohár | Tevékenység:   * Osztálykatalógus átnézése |
| 2025. 03.30. 20:00 | 5 perc | Kohár | Tevékenység:   * Teszt támogató segédprogram definiálása. |

1. https://junit.org/junit5/ [↑](#footnote-ref-1)