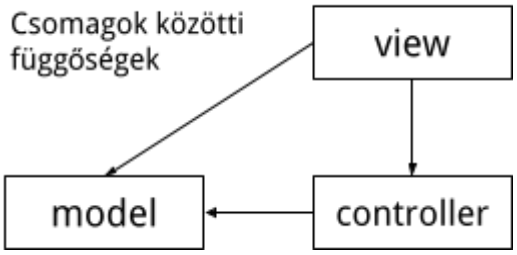
Programozói dokumentáció

Csomagok megválasztása



A programot három csomagba van szétválasztva: *model, view, controller*. Az osztályokat ezen szempontok alapján vannak besorolva a három csomag egyikébe:

- *view*: megjelenítéssel, felhasználói be- és kimenettel kapcsolatos osztályok. Általánosságban minden, amihez szükség volt a *swing* vagy *awt* csomagokra
- *controller*: algoritmusok, működés, viselkedés leírása a megjelenítéstől függetlenül
- *model*: adatszerkezetek, a programbeli objektumokat leíró osztályok, a viselkedéstől függetlenül

Labirintus generálása

Labirintus geometriája

Az absztrakt labirintus osztálynak két implementációja van a programban, egy négyszögletű és egy hatszög alapú labirintus. Az absztrakt osztályból azonban származtathatnánk komplexebb pl. voronoi alapú labirintusokat is.

- négyszögletes labirintus: labirintus csomópontjai egy rácshálón helyezkednek el, két csomópont csak akkor lehet szomszédos, ha befoglaló négyzeteiknek van közös oldaluk (tehát átlósan nem)
- hatszögletű labirintus: négyszögleteshez hasonló, azonban itt egy csomópontnak akár 6 szomszédja is lehet. Valójában ez is egy kétdimenziós tömbre képződik le (így hogy a labirintus hatszög alakú legyen, a tömbben vannak 'invalid' mezők), koordináta- és index-transzformációkkal éri el a program a kívánt eredményt. Az ehhez szükséges számításokat innen emeli át a program: https://www.redblobgames.com/grids/hexagons/. Az itt felsorolt megoldási utak közül a program az *axial coordinates*-t használja.

Folyosók

A folyosók mindig irányított fa struktúrájúak, ahol két csomópont akkor szomszédos akkor szomszédos ha a kettejük között nincs fal (a szobák nélkül). A gyökér a labirintus egy tetszőleges pontja, és ezen a csomóponton kívül minden más csomópontnak pontosan egy szülője van, ezt a programban a szülő irányába mutató vektorral tárolja. A labirintus kezdetben egy szabályos fából indul ki, és elemi lépések segítségével lesz randomizálva. Minden elemi lépés megtartja a fa tulajdonságot, a gyökeret azonban megváltoztathatja. Az algoritmus alapötlete ebből a videóból származik, itt részletesen el van magyarázva ez az elemi lépés is: https://www.youtube.com/watch?v=zbXKcDVV4G0

Szobák generálása

A szobák generálásakor az az alapvető gondolat, hogy csak egy kiválasztott pont részfájának pontjai vannak belevéve az adott szobába. Ez azért jó mert így nem jöhetnek létre túl nagy hurkok a fában (hiszen csak olyan csomópontok között keletkezik új átjárási lehetőség, akik eleve közel voltak egymáshoz). Nyilván ha ez a kiválasztott pont a gyökér, akkor ez minden pontot jelent, így nem lenne sok értelme ennek a kitételnek. Ezért a szobák generálása előtt a gyökér a bal alsó sarokba van állítva, majd a szobák generálása a jobb felső sarokból indul, és minden olyan pontból, ami még nem egy szoba része, új szobakeresés indul úgy, hogy a kiválasztott pont részfájának csak azon pontjait veszi figyelembe az algoritmus, ami még nem egy másik szoba része.

A szobák keresésére két különböző algoritmust is elérhető:

- négyszög alakú szobák keresése: az elérhető pontok közül megkeresi azon pontok halmazát, amelyek a legnagyobb területű négyzetet adják ki (amíg ez nem lépi túl a megadott felső korlátot)
- konkáv szobák keresése: az elérhető pontok közül kikeresi azt a legtöbb pontból álló részhalmazt, ami köré ha egy sokszöget rajzolnánk nem metszené önmagát a sokszög körvonala és az "átfogója" nem lépné át a megadott felső korlátot

Labirintus elemei

A szobákon és folyosókon kívül a labirintus tartalmazhat különböző *Storable* elemeket, illetve minden ilyen *Storable* elemhez tartozhat egy-egy fény is.

Storable **elemek**

Minden *Storable* elem a labirintus egy adott csomópontjához tartozik, ezen belül a csomópont középpontjától el van tárolva a csomóponton belüli eltolása. A csomópontok közötti falakat úgy kezeli le a program, hogy a csomópontokat úgy kezeli, hogy valamennyire egymásba lógnak (*padding* érték) és ebbe a közös területbe csak akkor helyezkedhet el a *Storable*, ha az egymásba lógó csomópontok szomszédosak. A játékos karaktere is egy ilyen *Storable* elem.

A Storable elemek *sprite*-ját a *ModelSprite* enum segítségével tárolja el a program, hogy a megjelenítéstől független legyen a modell, és a *view* csomagban lehessen meghatározni a *sprite*-hoz tartozó képet.

A *Storable* elemek lehetnek *Item* elemek is, jelen állásában a programban ez annyit jelent, hogy a játékos össze tudja szedni ezeket az elemeket, de nyilván ez a viselkedés még bővíthető. Ilyen *Item* például egy kulcs vagy a labirintus kijárata is.

Fények

Minden Storable elemhez opcionálisan tartozhat egy fény is. A fény az adott elemből árad, de a falakon nem képes áthatolni. A fény színét a *ModelColor* enum segítségével tárolja a program.

Labirintus állapotának tárolása

A labirintus elemeit a *LabState* osztály tárolja el, illetve ez az osztály tárolja el a játék állapotát is pl. játék neve, játékos karakter kezdő pozíciója, láthatósági beállítások.

Felhasználó felület

A felhasználó felület két fő részből áll: a játék és szerkesztői mód felületeiből. Ezt a program *CardLayout*-al kezeli, jelen esetben mindkét mód számára egy-egy kártya van fenntartva, de a program bővíthető lenne, hogy több szerkesztő vagy játék lehessen egyszerre megnyitva.

A felhasználó lementheti és betöltheti a programba a labirintusokat mindkét módban. Az lementett labirintusokat *.laby* kiterjesztéssel látja el a program a könnyű megkülönböztetés érdekében. Jelen állapotában a program csak a munkakönyvtárba menti el a labirintusokat, illetve csak innen tudja betölteni őket.

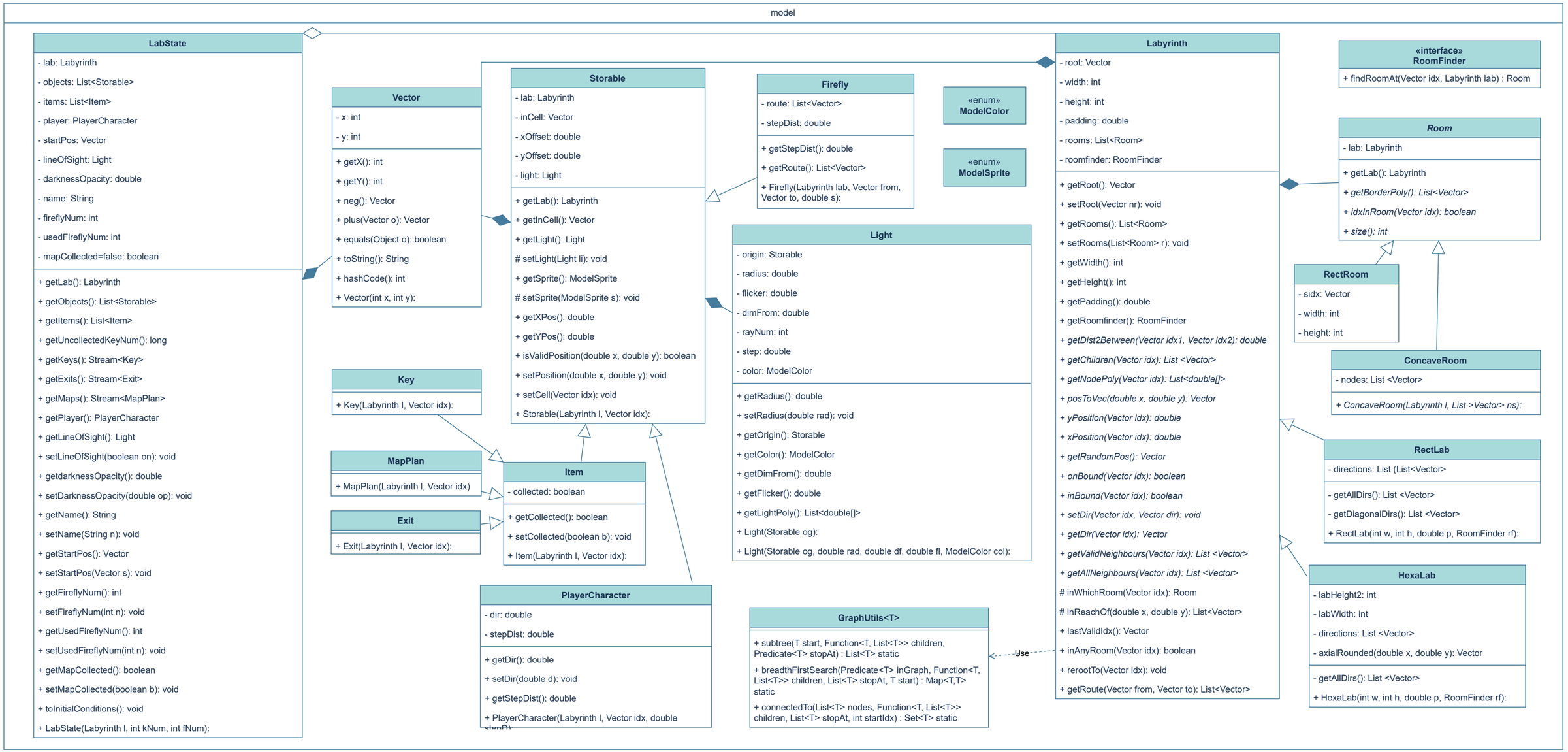
Fordítás és futtatás

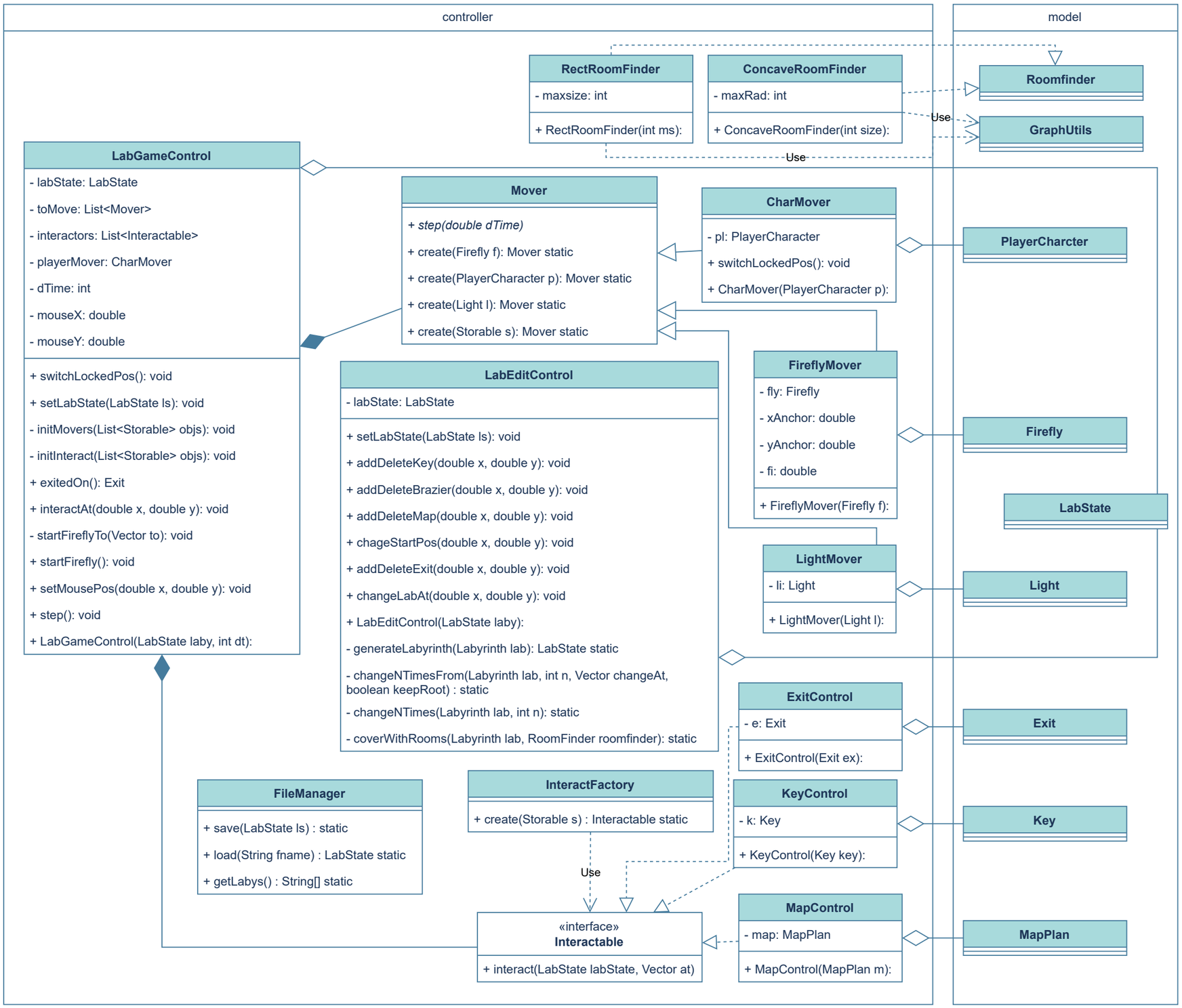
A program maven segítségével fordítható (*mvn compile*), futtatható (*mvn exec:java*) és tesztelhető (*mvn test*).

Részletes dokumentáció és osztálydiagramok

Az osztályok és metódusaik részletes leírása javadoc kommentekből generált *html*-benolvasható.

Az program osztályait leíró osztálydiagramok ennek a dokumentumnak a végén találhatóak.


model
LabState
- lab: Labyrinth
- objects: List<Storable>
- items: List<Item>
- player: PlayerCharacter
- startPos: Vector
- lineOfSight: Light
- darknessOpacity: double
- name: String
- fireflyNum: int
- usedFireflyNum: int
- mapCollected=false: boolean
+ getLab(): Labyrinth
+ getObjects(): List<Storable>
+ getItems(): List<Item>
+ getUncollectedKeyNum(): long
+ getKeys(): Stream<Key>
+ getExits(): Stream<Exit>
+ getMaps(): Stream<MapPlan>
+ getPlayer(): PlayerCharacter
+ getLineOfSight(): Light
+ setLineOfSight(boolean on): void
+ getdarknessOpacity(): double
+ setDarknessOpacity(double op): void
+ getName(): String
+ setName(String n): void
+ getStartPos(): Vector
+ setStartPos(Vector s): void
+ getFireflyNum(): int
+ setFireflyNum(int n): void
+ getUsedFireflyNum(): int
+ setUsedFireflyNum(int n): void
+ getMapCollected(): boolean
+ setMapCollected(boolean b): void
+ toInitialConditions(): void
+ LabState(Labyrinth l, int kNum, int fNum):
Vector
- x: int
- y: int
+ getX(): int
+ getY(): int
+ neg(): Vector
+ plus(Vector o): Vector
+ equals(Object o): boolean
+ toString(): String
+ hashCode(): int
+ Vector(int x, int y):
Storable
- lab: Labyrinth
- inCell: Vector
- xOffset: double
- yOffset: double
- light: Light
+ getLab(): Labyrinth
+ getInCell(): Vector
+ getLight(): Light
# setLight(Light li): void
+ getSprite(): ModelSprite
# setSprite(ModelSprite s): void
+ getXPos(): double
+ getYPos(): double
+ isValidPosition(double x, double y): boolean
+ setPosition(double x, double y): void
+ setCell(Vector idx): void
+ Storable(Labyrinth l, Vector idx):
Firefly
- route: List<Vector>
- stepDist: double
+ getStepDist(): double
+ getRoute(): List<Vector>
+ Firefly(Labyrinth lab, Vector from, Vector to, double s):
«enum»
ModelColor
«enum»
ModelSprite
Light
- origin: Storable
- radius: double
- flicker: double
- dimFrom: double
- rayNum: int
- step: double
- color: ModelColor
+ getRadius(): double
+ setRadius(double rad): void
+ getOrigin(): Storable
+ getColor(): ModelColor
+ getDimFrom(): double
+ getFlicker(): double
+ getLightPoly(): List<double[]>
+ Light(Storable og):
+ Light(Storable og, double rad, double df, double fl, ModelColor col):
Key
+ Key(Labyrinth l, Vector idx):
MapPlan
+ MapPlan(Labyrinth l, Vector idx)
Exit
+ Exit(Labyrinth l, Vector idx):
Item
- collected: boolean
+ getCollected(): boolean
+ setCollected(boolean b): void
+ Item(Labyrinth l, Vector idx):
PlayerCharacter
- dir: double
- stepDist: double
+ getDir(): double
+ setDir(double d): void
+ getStepDist(): double
+ PlayerCharacter(Labyrinth l, Vector idx, double stepD):
GraphUtils<T>
+ subtree(T start, Function<T, List<T>> children, Predicate<T> stopAt) : List<T> static
+ breadthFirstSearch(Predicate<T> inGraph, Function<T, List<T>> children, List<T> stopAt, T start) : Map<T,T> static
+ connectedTo(List<T> nodes, Function<T, List<T>> children, List<T> stopAt, int startIdx) : Set<T> static
Use
Labyrinth
- root: Vector
- width: int
- height: int
- padding: double
- rooms: List<Room>
- roomfinder: RoomFinder
+ getRoot(): Vector
+ setRoot(Vector nr): void
+ getRooms(): List<Room>
+ setRooms(List<Room> r): void
+ getWidth(): int
+ getHeight(): int
+ getPadding(): double
+ getRoomfinder(): RoomFinder
+ getDist2Between(Vector idx1, Vector idx2): double
+ getChildren(Vector idx): List <Vector>
+ getNodePoly(Vector idx): List<double[]>
+ posToVec(double x, double y): Vector
+ yPosition(Vector idx): double
+ xPosition(Vector idx): double
+ getRandomPos(): Vector
+ onBound(Vector idx): boolean
+ inBound(Vector idx): boolean
+ setDir(Vector idx, Vector dir): void
+ getDir(Vector idx): Vector
+ getValidNeighbours(Vector idx): List <Vector>
+ getAllNeighbours(Vector idx): List <Vector>
# inWhichRoom(Vector idx): Room
# inReachOf(double x, double y): List<Vector>
+ lastValidIdx(): Vector
+ inAnyRoom(Vector idx): boolean
+ rerootTo(Vector idx): void
+ getRoute(Vector from, Vector to): List<Vector>
«interface»
RoomFinder
+ findRoomAt(Vector idx, Labyrinth lab) : Room
Room
- lab: Labyrinth
+ getLab(): Labyrinth
+ getBorderPoly(): List<Vector>
+ idxInRoom(Vector idx): boolean
+ size(): int
RectRoom
- sidx: Vector
- width: int
- height: int
ConcaveRoom
- nodes: List <Vector>
+ ConcaveRoom(Labyrinth l, List >Vector> ns):
RectLab
- directions: List (List<Vector>
- getAllDirs(): List <Vector>
- getDiagonalDirs(): List <Vector>
+ RectLab(int w, int h, double p, RoomFinder rf):
HexaLab
- labHeight2: int
- labWidth: int
- directions: List <Vector>
- axialRounded(double x, double y): Vector
- getAllDirs(): List <Vector>
+ HexaLab(int w, int h, double p, RoomFinder rf):


controller
model
RectRoomFinder
- maxsize: int
+ RectRoomFinder(int ms):
ConcaveRoomFinder
- maxRad: int
+ ConcaveRoomFinder(int size):
Roomfinder
GraphUtils
Use
Use
LabGameControl
- labState: LabState
- toMove: List<Mover>
- interactors: List<Interactable>
- playerMover: CharMover
- dTime: int
- mouseX: double
- mouseY: double
+ switchLockedPos(): void
+ setLabState(LabState ls): void
- initMovers(List<Storable> objs): void
- initInteract(List<Storable> objs): void
+ exitedOn(): Exit
+ interactAt(double x, double y): void
- startFireflyTo(Vector to): void
+ startFirefly(): void
+ setMousePos(double x, double y): void
+ step(): void
+ LabGameControl(LabState laby, int dt):
Mover
+ step(double dTime)
+ create(Firefly f): Mover static
+ create(PlayerCharacter p): Mover static
+ create(Light l): Mover static
+ create(Storable s): Mover static
CharMover
- pl: PlayerCharacter
+ switchLockedPos(): void
+ CharMover(PlayerCharacter p):
PlayerCharcter
LabEditControl
- labState: LabState
+ setLabState(LabState ls): void
+ addDeleteKey(double x, double y): void
+ addDeleteBrazier(double x, double y): void
+ addDeleteMap(double x, double y): void
+ chageStartPos(double x, double y): void
+ addDeleteExit(double x, double y): void
+ changeLabAt(double x, double y): void
+ LabEditControl(LabState laby):
- generateLabyrinth(Labyrinth lab): LabState static
- changeNTimesFrom(Labyrinth lab, int n, Vector changeAt, boolean keepRoot) : static
- changeNTimes(Labyrinth lab, int n): static
- coverWithRooms(Labyrinth lab, RoomFinder roomfinder): static
FireflyMover
- fly: Firefly
- xAnchor: double
- yAnchor: double
- fi: double
+ FireflyMover(Firefly f):
Firefly
LabState
LightMover
- li: Light
+ LightMover(Light l):
Light
ExitControl
- e: Exit
+ ExitControl(Exit ex):
Exit
FileManager
+ save(LabState ls) : static
+ load(String fname) : LabState static
+ getLabys() : String[] static
InteractFactory
+ create(Storable s) : Interactable static
KeyControl
- k: Key
+ KeyControl(Key key):
Key
Use
«interface»
Interactable
+ interact(LabState labState, Vector at)
MapControl
- map: MapPlan
+ MapControl(MapPlan m):
MapPlan

view

MapView

- labState: LabState
- xoffset: int
- yoffset: int
- scale: double
- w: int
- h: int

+ getWidth(): int
+ getHeight(): int
- setSize(): void
+ setLabState(LabState ls): void
- fuzz(): int
- drawRoom(Graphics2D g, Room r): void
- drawCorridors(Graphics2D g): void
- xlabPosToPx(double p): int
- ylabPosToPx(double p): int
- paint(Graphics2D g): void
+ getImage(boolean rotate): BufferedImage
+ MapView(LabState laby, int sc):

LabView

- labState: LabState
- xoffset: int
- yoffset: int
- scale: double
- visiblityOverride: double
- drawMap: boolean

+ setVisiblityOverride(double vo): void
+ xlabPosToPx(double p): int
+ ylabPosToPx(double p): int
+ xpxToLabPos(int px): double
+ ypxToLabPos(int px): double
+ setLabState(LabState ls): void
- center(): void
- calculateCorridorWidth(): double
- decodeColor(ModelColor c): Color
- withMaxOpacity(Color c, int opacity): Color
- drawCell(Graphics2D g, Vector idx): void
- drawRoom(Graphics2D g, Room r): void
- drawCorridors(Graphics2D g): void
- drawImageAt(Graphics2D g, Image img, int x, int y, double sizeModifier): void
- drawObject(Graphics2D g, Storable obj): void
- getLightArea(Light l, double rad): Area
- circleInBound(int w, int h, int cx, int cy, double r): boolean
- drawLightColors(Graphics2D g, List<Light> lights): void
- darknessImage(List<Light> lights, int w, int h): BufferedImage
- drawStatus(Graphics2D g): void
+ paintComponent(Graphics g1): void
+ LabView(LabState laby, int sc, double vo):
+ LabView(LabState laby, int sc):

SimplePopup

- input: Componenet
- inpLabel: JLabel
- action: Runnable
- doneButton: Srting
- errorMsg: String
- cancellable: boolean

+ save(LabState lab): SimplePopup
+ load(MainDisplay disp, ModePanel pan): SimplePopup
+ message(String msg): SimplePopup
+ ask(String msg, Runnable a): SimplePopup
+ from(Compenent inp, Runnable a, String done): SimplePopup
+ startPopup(Component parent): void

MainDisplay

- mainPage: JPanel
- editPanel: EditPanel
- gamePanel: GamePanel
- current: ModePanel

+ switchTo(ModePanel m): void
- menuSetup(): JMenuBar
+ MainDisplay()
+ main(String[] args): void

Use

ModePanel

- labState: LabState
- labView: LabView
- timer: Timer

+ getLabState(): LabState
+ setLabState(LabState ls): void
+ init(): void
+ startMode(): void
+ exitMode(): void

StructSettings

- lWidth: int
- lHeight: int
- lType: String
- rSize: int
- rType: String
- pan: ModePanel
- disp: MainDisplay

+ generate(): LabState
+ generateAndSet(): void
+ StructSettings()
+ StructSettings(ModePanel mp):
+ StructSettings(MainDisplay d, ModePanel mp):

Use

CustomTimerTask

- task: Runnable

+ run(): void
+ CustomTimerTask(Runnable t):

EditPanel

- labEditControl: LabControl
- currMode: Mode

- createOptionPanel(): JPanel
- createGameplayPanel(): JPanel
- createSettingsPanel(): JPanel
+ EditPanel(LabState ls, int dt):

Use

«enum»
Mode

GamePanel

-- labControl: LabGameControl

+ GamePanel(LabState ls, int dt):

controller

FileManager

LabEditControl

Use

LabGameControl

model

LabState