Purpose
This document provides documentation both on the syntax and semantics of the board game model.

High Level Overview
We describe a game as being composed of *objects*, *actions, players* and *win conditions.* Players take actions manipulating objects until a win condition is reached.

Players
Terms:
- Active
- Inactive
- Owns

A player can be either *active* or *inactive*. Only active players can take actions. You can think of this as a way of specifying who's turn it is, without rigidly limiting action taking to a clockwise one-action-per-player system. A game in this model could, for instance, set all players as active for the game's duration (e.g. Dutch Blitz).

Players may *own* objects. An object cannot have multiple owners.

Objects
Terms:
- Type
- In
- Contains
- Owned

Every object has a *type*. Think of every object as a container. Any object can be *in* one other object. An object can *contain* any number of objects. The analogy of a container breaks down however, because "loops" of objects can be formed; for instance, A in B, B in C, and C in A. Objects can even contain themselves.

Objects can be *owned* by a player, but they don't have to be.

Actions
Terms:
Requirements
- In
- Recursive In
- Contains

- Recursive Contains
- Owns
- Timer

Effects

- Put
- Remove
- Create
- Delete
- Distribute
- Disown
- Players
- Set Timer

Modifiers

- Not
- .in
- .contains
- .type

Actions are applied to objects, and have prerequisites in the form of *require* statements. If objects exist meeting an action's requirements, an active player may take the action on those objects, thereby performing action's *effects*.

Here is an example action that takes an object of type "Car" out of an object of type "Road".

```
Action Park Car c Road r
      require in c r
      effect remove c
```

Declaration Syntax

The first word is always "Action" followed by the action name. Then objects the action is applied on are listed "ObjectType name".

Requirements

In

```
require in A:object B:object
require in A:object T:type
```

Require that A is in B. A must be an object. B can be an object or a type.

Contains

```
require contains A:object B:object
require contains A:object B:objects
require contains A:object T:type
require contains A:object T:types
require contains A:object T:type N:number
require contains A:object T:type < N:number
require contains A:object T:type > N:number
```

The inverse of In -- requires that A contains B. May require that A contains an object of type T, some N number of objects with type T, or even that A contains fewer (or more) than N objects of type T. To see how to make B refer to multiple objects (or T refer to multiple types), read about "*.contains*" and "*.type*".

Recursive In

```
require rin A:object B:object
require rin A:object T:type
```

The same as *In*, except that instead of just checking the object that A is in, we also check the object (A is in) is in, and ((A is in) is in) is in, and so on to see if any of them match the criteria. Any implementation of this functionality in code must be careful to check for infinite loops.

Recursive Contains

```
require rcontains A:object B:object
require rcontains A:object B:objects
require rcontains A:object T:type
require rcontains A:object T:types
require rcontains A:object T:type N:number
require rcontains A:object T:type < N:number
require rcontains A:object T:type > N:number
```

The same as *Contains*, except that we also recursively check if the the objects are recursively contained in the objects A contains. To see how to make B refer to multiple objects (or T refer to multiple types), read about "*.contains*" and "*.type*".

Owns

```
require owns A:object
require owns T:type
require owns T:type < N:Number
require owns T:type > N:Number
```

Requires that the player taking the action owns the object A or an object of type T.

Timer

```
require timer
```

Actions are allowed to set the game's timer, at which point it begins counting down. This is true when the timer still has seconds on the clock.

Note on Requirements

Software implementing these requirement options could implicitly add other requirements to ensure software safety.

Effects

Put

```
effect put A:object B:object
effect put T:type B:object
effect put T:type B:object N:Number
```

Puts an object A into B. If A is already in another object C, A is first removed from C. If only a type is specified, an object of type T is created and put in B.

Remove

```
effect remove A:object
effect remove T:type B:object N:number
effect remove T:type B:object all
```

When only an object is specified, that object is pulled out of whatever contains it. Otherwise, Remove is used to pull objects of a certain type out of a container.

Create

```
effect create T:type
```

Creates an object of type T.

Delete

```
effect delete A:Object
effect delete O:Objects
```

Deletes the object(s) specified. to see how to specify multiple objects read about "*.contains*".

Distribute

```
effect distribute A:object P:players T:type N:number
```

Takes a random selection of N objects (type T) from A and evenly distributes them to the specified players.

Disown

```
effect disown A:object
effect disown A:objects
effect disown T:type P:players
effect disown T:type P:players all
effect disown T:type P:players N:number
effect disown T:type P:players rand
effect disown T:type P:players rand N:number
```

In the first case, whichever player used to own A no longer does. When removing objects of a type, the “rand” determines whether or not the player losing the objects gets to choose what they are.

Players

```
effect players all
effect players player
effect players player.left
effect players player.right
effect players players.left
effect players players.right
effect players former
effect players former.left
effect players former.right
```

This effect sets the active players. “All” sets all players to active. “Player” sets the person taking the action to active (Note that they can only take this action if already active. Useful primarily for moving from all players as active to one specific player.) “Player.left” and “Player.right” refer to the players sitting next to the person taking the action. “Former” is similar, but refers to the player who previously took an action.

Set Timer

```
effect settimer N:number
```

Initializes the game’s timer to N seconds. The timer immediately begins counting down.

Modifiers

Not

```
require not SomeCondition
```

Not is simply a prefix for any requirement. It flips the truth-value of the condition.

.in

Consider the following action:

```
Action DigDeeper Mole m DirtLayer d1 DirtLayer d2
```

```
      require in m d1
      require in d1 d2
      effect put m d2
```

It performs the following:

```
m ------ [___]
d1 --- [______]      ⇒   d1 --- [___]  [___] --- m
d2 - [__________]        d2 - [___________]
```

Assuming we have prior knowledge concerning the structure of DirtLayer objects, we can express the same action in the following way.

```
Action DigDeeper Mole m
      effect put m m.in.in
```

The effects are the same, though the second action does not specify the types of the objects that m is nested in.

```
m ----------- [___]
m.in ------ [______]      ⇒   previously m.in --- [___]  [___] --- m
m.in.in - [__________]        new m.in --------- [___________]
```

.contains

.contains is very similar to .in, but since a.contains refers to everything in a, .contains often references multiple objects.

Once .contains is used, it may not be followed by another .contains nor a .type.

Example:

```
Action EmptyContainer Container c
      effect remove c.contains
```

.type

Switches from talking about an object to the type of an object.

Example:

```
require contains store shoppingList.contains.type
```

Initial Game State

Once a list of valid object types is declared, you can create a representation of the initial game state.

For instance, to represent a “circular” track of 10 cells, you could create the following:

```
TrackCell t0
TrackCell t1
      contains t0
TrackCell t2
      contains t1
TrackCell t3
      contains t2
TrackCell t4
      contains t3
TrackCell t5
      contains t4
TrackCell t6
      contains t5
TrackCell t7
      contains t6
TrackCell t8
      contains t7
TrackCell t9
      contains t8
TrackCell t0
      contains t9
```

Note that `t0` appears to be declared twice. The first time is its actual declaration; the second simply adds to `t0`’s properties.

The track example could be performed slightly differently so as to remove the directly stated modification to `t0`.

```
TrackCell t0
TrackCell t1
      contains t0
TrackCell t2
      contains t1
...
TrackCell t8
      contains t7
TrackCell t9
      in t0
      contains t8
```

In keeping with the model, an object can contain multiple objects but can only be in one. Consider the following example:

```
Cow A
Cow B
Farm C
      contains A
Farm D
      contains A
      contains B
```

In this case, the meaning is unclear. Does `C` or `D` contain `A`? This object instantiation is malformed.

When declaring an object instance, you don’t need to give it a name. Of course, if no name is given, it cannot be referenced by subsequently declared objects.

Win/Lose Conditions

Defining when a win or loss occurs is done very similarly to defining when an action is available.

A win condition (or lose condition) is relevant to all players who are currently active. They are only checked between actions. That is, actions are thought of as atomic, even though they can have multiple effects. A win condition simply means the active players win when the conditions are met. A lose condition means the active players are eliminated from the game. A lose condition may also have effects; this allows “clean-up” to be performed, including changing the active player set.

The syntax for win and lose conditions is exactly the same as that for actions, except that instead of “`Action`” you have either “`WinCondition`” or “`LoseCondition`”.