pyherc Documentation

Release 0.16.0

Tuukka Turto

March 26, 2016

Contents

CHAPTER 1

Reference

1.1 Intro

pyherc is a roguelike engine written in combination of Python and Hy. This document has brief description of some major parts of the system.

1.2 Building blocks

Codebase is divided in two main pieces `pyherc` and `herculeum`. pyherc is a sort of platform or library for writing roguelike games. herculeum on the other hand is a sample game that has been written on top of pyherc.

1.3 Main components

1.3.1 Model

`pyherc.data.model.Model` is the main class representing current state of the playing world. It holds reference to important things like:

- Player character
- Dungeon
- Configuration
- Various tables

1.3.2 Character

`pyherc.data.character.Character` is used to represent both player character and monsters. It manages things like:

- Stats
- Inventory
- Location

1.3.3 Dungeon

`pyherc.data.dungeon.Dungeon` is currently very sparse and is only used to hold reference to first level in the dungeon.

1.3.4 Level

`pyherc.data.level.Level` is key component, as it is used to store layout and content of levels where player adventures. It manages:

- Shape of the level, including stairs leading to other levels
- Items
- Characters

1.3.5 Rules

`pyherc.rules` is what defines what kind of actions player and monsters are allowed to take and how they affect the world around them. Rules for things like moving, fighting and drinking potions are found here. Refer to Actions for more detailed description how actions are created and how to add more.

1.3.6 Ports

`pyherc.ports` is the interface that rest of the code uses to connect with actions subsystem. Instead of interfacing with ActionFactory and relatively complex logic, client code should use functions defined in this module.

1.4 Generating a level

This section will have a look at level generation, how different parts of the software work together to create a new level and how to add new levels into the game.

1.4.1 Overview of generating dungeon

Dungeon is used to represent playing area of the game. It contains levels which player can explore.

Dungeon is generated by `pyherc.generators.dungeon.DungeonGenerator`.

1.4.2 Adding a new type of level

Adding a new level is quite straightforward procedure, when you know what you are doing. Following section will give a rough idea how it can be accomplished.

Level generator

In order to add a new type of level into the game, a level generator needs to be written first. It has a simple interface:

```
(fn generate-level [self portal]
   ...)
```

Arguments supplied to this function are:

- portal - Portal at an existing level, where this level should be connected

Shape of the level

One of the first things for our level generator to do, is to create a new Level object:

```
(new-level model)
```

This call will instantiate a Level object. Note that the level initially has no dimensions at all. The datastructure used will allow level to grow to any direction, as much as there is memory in the computer (more or less anyway). Now the level generator code can start modifying layout of the level:

```
(for [y (range 1 39)]
  (for [x (range 1 79)]
    (floor-tile #t(x y) :stone)))
```

Adding monsters

No level is complete without some monsters. Next we will add a single rat:

```
(add-character level (.find-free-space level)
               (creature-generator "rat"))
```

Adding items

Our brave adventurer needs items to loot. Following piece of code will add a single random food item:

```
(add-item level (.find-free-space level)
          (self.item-generator :item-type "food"))
```

Linking to previous level

Our level is almost ready, we still need to link it to level above it. This is done using the Portal object, that was passed to this generator in the beginning:

```
(when portal
  (let [[another-portal (Portal)]]
    (setv another-portal.model model)
    (.add-portal level another-portal
                 (.find-free-space level)
                 portal)))
```

First we create a new Portal and link it to our Model. Then we add it to the new level at random location and link it to portal on a previous level.

Linking to further levels

If you want to this dungeon branch to continue further, you can create new Portal objects, place them on the level and repeat the process above to generate level.

Another option is to use proxy level generators, that will cause levels to be generated at the moment when somebody tries to walk through portal to enter them.

Adding level into the dungeon

Now you have a generator that can be used to generate new levels. Last step is to modify an existing level generator to place a portal and create a level using this new generator. If that step is skipped, new type of levels will never get generated.

1.5 Modular level generator

Now that we are aware how level generation works in general, we can have a look at more modular approach. `pyherc.generators.level.new_level_generator()` is a high order function used to construct new modular level generator functions.

```
(defn new-level-generator [model partitioners room-generators decorators
                           portal-adders item-adders creature-adders
                           trap-generator rng name description]
  ...)
```

Calling this function will return a function that can be used to generate level as configured. It has simple interface:

```
(fn [portal]
  ...)
```

1.5.1 Overview of level generator

Instead of performing all the steps by itself, level generator delegates most of its tasks to sub components.

First new level is created and sent to a partitioner. This component will divide level into sections and link them to each other randomly. Partitioners are required to ensure that all sections are reachable.

A room is generated within each section and corridors are used to link rooms to neighbouring sections. Linking is done according to links set up in the previous phase. This in turn ensures that each room is reachable.

Adding of creatures is done by creature adders. These contains information of the type of creatures to add and their placement.

Items are added in the same way as the portals, but item adders are used.

Portals are added by portal adders. These portals will lead deeper in the dungeon and cause new levels generated when player walks down to them. One special portal is also created, that links generated level to the higher level.

In decoration step details are added into the level. Walls are built where empty space meets solid ground and floors are detailed.

1.5.2 Partitioners

`pyherc.generators.level.partitioners.grid.grid_partitioning()` creates a basic partitioner, which knows how to divide level into a grid with equal sized sections.

All partitioners have same interface:

```
(fn [level]
  ...)
```

Calling the function will partition level to sections, link sections to each other and return them in a list.

`pyherc.generators.level.partitioners.section.Section` is used to represent section. It defines a rectangular area in level, links to neighbouring areas and information how they should connect to each other. It also defines connections for rooms.

1.5.3 Room generators

Room generators are used to create rooms inside of sections created by partitioner. Each section has information how they link together and these connection points must be linked together by room generator.

Room generator is instantiated with `pyherc.generators.level.room.new_room_generator()` function. It will create a generator function with following signature:

```
(fn [section trap-generator]
  ...)
```

Calling this function should create a room inside section and connect all connection points together.

1.5.4 Decorators

Decorators can be used to add theme to level. Simple ones can be used to change appearance of the floor to something different than what was generated by room generator. More complex usage is to detect where walls are located and change their appearance.

Decorators have simple interface:

```
(fn [level]
  ...)
```

1.5.5 Portal adders

```
(fn [level]
  ...)
```

1.5.6 Creature adder

```
(fn [level]
  ...)
```

1.5.7 Item adder

```
(fn [level]
  ...)
```

1.5.8 Defining levels

Levels are defined in configuration scripts that are fed to `pyherc.config.config.Configuration` during system startup.

1.6 Generating an item

This section will have a look at item generation and how to add new items into the game.

1.6.1 Overview of generating item

`pyherc.generators.item.ItemGenerator` is used to generate items.

To generate item, following code can be used:

```
new_item = self.item_generator.generate_item(item_type = 'food')
```

This will generate a random item of type food. To generate item of specic name, following code can be used:

```
new_item = self.item_generator.generate_item(name = 'apple')
```

This will generate an apple.

1.6.2 Defining items

Items are defined in configuration scripts that are fed to `pyherc.config.config.Configuration` during system startup. Following example defines an apple and dagger for configuration.

```
from pyherc.generators import ItemConfigurations
from pyherc.generators import ItemConfiguration, WeaponConfiguration
from pyherc.data.effects import EffectHandle

def init_items():
    """
    Initialise common items
    """
    config = []

    config.append(
                  ItemConfiguration(name = 'apple',
                                    cost = 1,
                                    weight = 1,
                                    icons = [501],
                                    types = ['food'],
                                    rarity = 'common'))

    config.append(
                  ItemConfiguration(name = 'dagger',
                                    cost = 2,
                                    weight = 1,
                                    icons = [602, 603],
                                    types = ['weapon',
                                               'light weapon',
                                               'melee',
                                               'simple weapon'],
                                    rarity = 'common',
                                    weapon_configration = WeaponConfiguration(
                                            damage = [(2, 'piercing'),
                                                      (2, 'slashing')],
                                            critical_range = 11,
                                            critical_damage = 2,
                                            weapon_class = 'simple')))

    return config

config = init_items()

print(len(config))
print(config[0])
```

Example creates a list containing two ItemConfiguration objects.

```
2
<pyherc.generators.item.ItemConfiguration object at 0x...>
```

For more details regarding to configuration, refer to Configuration page.

1.7 Generating characters

This section will have a look at character generation and related actions.

1.7.1 Character generator

Characters can be created with `generate-creature` function:

```
(generate-creature config model item-generator rng "rat")
```

Supplying creature configuration, model instance, item generator and random number generator every time is tedious. For that reason, application configuration `pyherc.config.Configuration` has attribute `creature_generator` that holds reference to function with a simpler interface, that is configured when system starts:

```
(creature-generator "rat")
```

Only name is required, all other parameters are automatically using the values supplied when the system started. This is also the function that is usually passed around in the system to places where creatures might be generated (level generators mainly).

1.7.2 Character selector

When a specific part of the system requires ability to generate characters, there are two options. First option is to pass a full fledged creature generator and use that as explained in the previous paragraph. Another, much simpler option is to use character selector. This is just a function, that takes no parameters and will return a list of generated creatures. Advantage of using them over creature generator is simplified usage:

```
(defn skeletons [empty-pct character-generator rng]
  "create character selector for skeletons"
  (fn []
    (if (> (.randint rng 1 100) empty-pct)
      (character-generator "skeleton warrior")
      []))))

(setv character-selector (skeletons 50
                                    creature-generator
                                    random))

(setv monster (character-selector))
```

Usually character selector are given a descriptive name, like `skeletons` or `common-critters`. For example `pyherc.data.features.new_cache()` uses selectors to configure what kind of creatures or items might reside inside of the cache.

1.8 Actions

This section will have a look at actions, how they are created and handled during play and how to add new actions.

1.8.1 Overview of Action system

Actions are used to represent actions taken by characters. This include things like moving, fighting and drinking potions. Every time an action is taken by a character, new instance of Action class (or rather subclass of it) needs to be created.

1.8.2 Action creation during play

Actions are instantiated via ActionFactory, by giving it correct parameter class. For example, for character to move around, it can do it by:

```
(.execute (action-factory (MoveParameters character
                                          Direction.west)))
```

This creates a WalkAction and executes it, causing the character to take a single step to given direction. Doing this all the time is rather cumbersome, so there are convenience functions at `pyherc.ports` that can be used:

```
(move character Direction.west)
```

For checking if an action can be performed, following ways are generally supported:

```
(.legal? (action-factory (MoveParameters character
                                         Direction.west)))

(move-legal? character Direction.west)
```

The first example will always be supported. The second example is generally supported, but not always.

1.8.3 Interface

Each function at `pyherc.ports` should return either `(Right character)` if the action was succesfull, or `(Left character)` if it couldn't be completed. First parameter of the function should be the character who is performing the action. Following these conventions allows us to define more complex actions as terms of simpler ones:

```
(defn lunge [character direction rng]
  (monad-> (move character direction)
           (attack direction rng)
           (add-cooldown)))
```

Character is threaded through consecutive calls. If any of the calls fail for any reason, calls after that one are automatically bypassed.

1.8.4 Extending

ActionFactory has been designed to allow easy adding of new actions. Each action has a respective factory function that can create it. These factory functions are registered at the startup of the system in `pyherc.config.Configuration` class. When an action is requested, each factory function is called in turn, until a correct one is found.

Factory function has general structure of:

```
(fn [parameters]
  (if (can-handle? parameters)
    (Just Action)
    (Nothing)))
```

If factory function can handle the request, new action is returned, wrapped inside `Just`. In case function can not handle this request `Nothing` is returned.

1.9 Events

Events, in the context of this article, are used in relaying information of what is happening in the game world. They should not be confused with UI events that are created when buttons of UI are pressed.

1.9.1 Overview of event system

Events are represented by classes found at `pyherc.events` and they all inherit from `pyherc.events.event.Event`.

Events are usually created as a result of an action, but nothing prevents them from being raised from somewhere else too.

Events are relayed by `pyherc.data.model.Model.raise_event()` and there exists convenient `pyherc.data.character.Character.raise_event()` too.

`pyherc.data.character.Character.receive_event()` method receives an event that has been raised somewhere else in the system. The default implementation is to store it in internal array and process when it is character's turn to act. The character can use this list of events to remember what happened between this and his last turn and react accordingly.

1.10 Effects

This section will have a look at effects, how they are created and handled during the play and how to add new effects.

1.10.1 Overview of effects system

Effects can be understood as on-going statuses that have an effect to an character. Good example would be poisoning. When character has poison effect active, he periodically takes small amount of damage, until the effect is removed or it expires.

Both items and characters can cause effects. Spider can cause poisoning and healing potion can grant healing.

1.10.2 Effect handles

`pyherc.data.effects.effect.EffectHandle` are sort of prototypes for effects. They contain information on when to trigger the effect, name of the effect, possible overriding parameters and amount of charges.

1.10.3 Effect

`pyherc.data.effects.effect.Effect` is a baseclass for all effects. All effects have duration, frequency and tick. Duration tells how long it takes until effect naturally expires. Frequency tells how often effect is triggered and tick is internal counter which keeps track when effect should trigger.

When creating a new effect, subclass Effect class and define method:

```
def do_trigger(self):
```

Do trigger method is automatically triggered when effect's internal counter reaches zero. After the method has been executed, counter will be reset if the effect has not been expired.

1.10.4 Creating Effects

Effects are cread by `pyherc.generators.effects.create_effect()`. It takes configuration that defines effects and named arguments that are effect specific to create an effect.

EffectsFactory is configured during the start up of the system with information that links names of effects to concrete Effect subclasses and their parameters.

```
from pyherc.generators import create_effect, get_effect_creator
from pyherc.data.effects import Poison
from pyherc.test.cutesy import Adventurer
from pyherc.rules import Dying

effect_creator = get_effect_creator({'minor poison': {'type': Poison,
                                                      'duration': 240,
                                                      'frequency': 60,
                                                      'tick': 60,
                                                      'damage': 1,
                                                      'icon': 101,
                                                      'title': 'Minor poison',
                                                      'description': 'Causes minor amount of damage'

Pete = Adventurer()
print('Hit points before poisoning: {0}'.format(Pete.hit_points))

poisoning = effect_creator('minor poison', target = Pete)
poisoning.trigger(Dying())

print('Hit points after poisoning: {0}'.format(Pete.hit_points))
```

Pete the adventurer gets affected by minor poison and as a result loses 1 hit point.

```
Hit points before poisoning: 10
Hit points after poisoning: 9
```

Note how the effect factory has been supplied by a dictionary of parameters. These are matched to the constructor of class specified by 'type' key. All parameters that are present in the constructor, but are not present in the dictionary needs to be supplied when effect factory creates a new effect instance. In our example there was only single parameter like this, the target of poisoning.

It is also possible to supply parameters during call that have been specified in the dictionary. These parameters are then used to override the default ones.

1.10.5 Effects collection

`pyherc.data.effects.effectscollection.EffectsCollection` is tasked to keep track of effects and effect handles for particular object. Both Item and Character objects use it to interact with effects sub system.

Following example creates an EffectHandle and adds it to the collection.

```python
from pyherc.data.effects import EffectsCollection,EffectHandle

collection = EffectsCollection()
handle = EffectHandle(trigger = 'on kick',
                      effect = 'explosion',
                      parameters = None,
                      charges = 1)
collection.add_effect_handle(handle)

print(collection.get_effect_handles())
```

The collection now contains a single EffectHandle object.

```
[<pyherc.data.effects.effect.EffectHandle object at 0x...>]
```

Following example creates an Effect and adds it to the collection.

```python
from pyherc.data.effects import EffectsCollection, Poison

collection = EffectsCollection()
effect = Poison(duration = 200,
                frequency = 10,
                tick = 0,
                damage = 1,
                target = None,
                icon = 101,
                title = 'minor poison',
                description = 'Causes small amount of damage')
collection.add_effect(effect)

print(collection.get_effects())
```

The collection now contains a single Poison object.

```
[<pyherc.data.effects.poison.Poison object at 0x...>]
```

1.11 Configuration

Configuration of pyherc is driven by external files and internal scripts. External files are located in resources directory and internal scripts in package `pyherc.config`.

1.11.1 Configuration scripts

pyherc supports dynamic detection of configuration scripts. The system can be configured by placing all scripts containing configuration in a single package and supplying that package to `pyherc.config.config.Config` class during system start:

```
self.config = Configuration(self.base_path, self.world)
self.config.initialise(herculeum.config.levels)
```

1.11.2 Level configuration

The file containing level configuration should contain following function to perform configuration.

```
def init_level(rng, item_generator, creature_generator, level_size)
```

This function should create `pyherc.generators.level.config.LevelGeneratorFactoryConfig` with appropriate values and return it. This configuration is eventually fed to `pyherc.generators.level.generator.LevelGeneratorFactory` when new level is requested.

1.11.3 Item configuration

The file containing item configuration should contain following function to perform configuration

```
def init_items(context):
```

This function should return a list of `pyherc.generators.item.ItemConfiguration` objects.

1.11.4 Character configuration

The file containing character configuration should contain following function to perform configuration:

```
def init_creatures(context):
```

This function should return a list of `pyherc.generators.creature.CreatureConfiguration` objects.

1.11.5 Player characters

Player characters are configured almost identically to all the other character. The only difference is the function used:

```
def init_players(context):
```

1.11.6 Effects configuration

The file containing effects configuration should contain following function to perform configuration

```
def init_effects(context):
```

This function should return a list of effect specifications.

1.11.7 Handling icons

Each of the configurators shown above take single parameter, context. This context is set by client application and can be used to relay information that is needed in configuration process. One such an example is loading icons.

Example of context can be found at `herculeum.config.config.ConfigurationContext`.

1.12 Magic

This section will outline how spells are implemented.

1.12.1 Overview of Magic system

SpellCastingAction created by SpellCastingFactory SpellCastingAction has

- caster
- spell
- effects_factory
- dying_rules

Spell has

- targets []
- EffectsCollection
- spirit

Spell is created by SpellGenerator by using SpellSpecification

SpellSpecification has

- effect_handles
- targeter
- spirit

1.12.2 How spells are cast

1.12.3 Spell creation during play

1.12.4 Adding a new type of spell

Overview

Whole code

1.13 Finite-state machines

Finite-state machine is often used for artificial intelligence routines in games. They can model different states character can be: patrolling, searching for food, investigating noise and fighting. There is a small DSL for defining finite-state machines supplied with pyherc.

1.13.1 Sample configuration

Following code is a sample definition for a very simple finite-state machine. It has two states `addition` and `subtraction`.

```
(defstatemachine SimpleAdder [message]
  "finite-state machine for demonstration purposes"

  "add 1 to message, 0 to switch state"
  (addition initial-state
            (active (+ message 1))

            "message 0 will change state"
            (transitions [(= message 0) subtraction]))

  "substract 1 from message, 0 to switch state"
  (subtraction (active (- message 1))

               "message 0 will change state"
               (transitions [(= message 0) addition])))
```

In order to use the finite-state machine, one needs to create an instance of it and call it like a function:

```
=> (setv fsm (SimpleAdder))
=> (fsm 1)
2
=> (fsm 2)
3
=> (fsm 0)
-1
=> (fsm 1)
0
=> (fsm 2)
1
```

As you can see, `fsm` will first return the argument passed to it plus 1. As soon as `0` is passed in, finite-state machine switches to subtraction state and starts returning the argument passed to it minus 1. Passing a `0` again will change the state back to addition.

Sometimes there's need to perform extra initialisation when finite-state machine is created or store data across different states. Following example highlights how `--init--` and `state` forms can be used to achieve this.

```
(defstatemachine Minimal [message]
  "default initializer"
  (--init-- [bonus] (state bonus bonus))
  "handle message"
  (process initial-state
           (active (* message (state bonus)))))
```

Following example shows how the finite-state machine defined in previous example can be used:

```
=> (setv fsm (Minimal 3))
=> (fsm 1)
3
=> (fsm 5)
15
```

As you can see, the parameter supplied during initialization of finite-state machine is stored under symbol `bonus` and used when finite-state machine is activated.

1.13.2 Syntax of finite-state machine definition

Finite-state machine is defined with `(defstatemachine <name> <parameters>)` form. `<name>` defines name of the class that will encapsulate finite-state machine definition. `<parameters>` is a list of zero or more symbols that define function interface that the finite-state machine will have. Keyword only, optional or other special parameter types are not supported.

Inside of `defstatemachine` form, there are one or more state definitions. Strings are allowed and they're treated as comments (ie. ignored). Format of state definition is `(<name> [initial-state] [(on-activate ...)] [(active ...)] [(on-deactivate ...)] [(transitions ...)])`. `<name>` is name of the state, it should be unique within a finite-state machine as transitions refer to them. One and only one of the states should be marked as an `initial-state`. This is the state the finite-state machine will enter when first activated. Rest three forms are all optional. Order of the forms is not significant. Symbols defined in `<parameters>` block of `defstatemachine` are available to all of these three functions. Strings are allowed and they are treated as comments (ie. ignored). Special form `--init--` can be used to create initializer method for finite-state machine. It has syntax of `(--init-- <parameters> <body>)`. `<parameters>` is a list of symbols that are to be added in `--init--` method of the finite-state machine and `<body>` is one or more s-expressions that are to be executed when finite-state machine is initialized.

First one is `on-activate`, which defines code that is executed when the given state is activated. Second one is `active` which defines code that is executed every time for the active state when finite-state machine is activated. `on-activate` is mirrored by `on-deactivate`, which gets executed every time a state deactivates. The last one is `transitions`. It defines one or more two element lists, where the first element is test and second element is symbol of a state to switch if the test returns true. `transitions` are checked for the active state every time finite-state machine is activated and it is performed before `active` code is executed.

In order to store data and pass it between states, `state` macro can be used. It has syntax of: `(state <symbol> [value])`. `<symbol>` is the stored data being accessed. If optional `value` is supplied, stored data is updated. In any case `state` returns the current value of the data.

1.14 Error Handling

Pyherc, like any other software contains errors and bugs. Some of them are so fatal that they could potentially crash the program. This chapter gives an overview on how runtime errors are handled.

1.14.1 General idea

The general idea is to avoid littering the code with error handling and only place it where it actually makes difference. Another goal is to keep the game running as long as possible and avoid error dialogs. Instead of displaying an error dialog, errors are masked as magical or mystical events. There should be enough logs though to be able to investigate the situation later.

1.14.2 Specific cases

Character

`pyherc.data.character.Character` is a central location in code. Majority actions performed by the characters flow through there after they have been initiated either by a user interface or artificial intelligence routine.

`pyherc.data.character.guarded_action()` is a decorator that should only be used in Character class. In the following example a move method has been decorated with both logged and guarded_action decorators:

```
@guarded_action
@logged
def move(self, direction, action_factory):
    ...
```

In case an exception is thrown, guarded_action will catch and handle it. The game might be in inconsistent state after this, but at least it did not crash. The decorator will set tick of the character to suitable value, so that other characters have a chance to act before this one is given another try. It will also emit `pyherc.events.error.ErrorEvent` that can be processed to inform the player that there is something wrong in the game.

Since the decorator is emitting an event, it should not be used for methods that are integral to event handling. This might cause an infinite recursion that ultimately will crash the program. It is best suited for those methods that are used to execute actions, like `pyherc.data.character.Character.move()` and `pyherc.data.character.Character.pick_up()`

1.15 Testing

This section will have a look at various testing approaches utilised in the writing of the game and how to add more tests.

1.15.1 Overview of testing

Tools currently in use are:

- nose
- doctest
- behave
- mockito-python
- pyhamcrest

Nosetests are mainly used to help the design and development of the software. They form nice safety net that catches bugs that might otherwise go unnoticed for periods of time.

Doctest is used to ensure that code examples and snippets in documentation are up to date.

Behave is used to write tests that are as close as possible to natural language.

Additional tool called nosy can be used to run nosetests automatically as soon as any file change is detected. This is very useful when doing test driven development.

1.15.2 Running tests

Nose

Nose tests can be run by issuing following command in pyherc directory:

```
nosetests
```

It should output series of dots as tests are executed and summary in the end:

```
......................................................................
......................................................................
........................................
----------------------------------------------------------------------
Ran 180 tests in 3.992s
```

If there are any problems with the tests (or the code they are testing), error will be shown along with stack trace.

Doctest

Running doctest is as simple. Navigate to the directory containing make.bat for documentation containing tests (doc/api/) and issue command:

```
make doctest
```

This will start sphinx and run the test. Results from each document are displayed separately and finally summary will be shown:

```
Doctest summary
===============
    4 tests
    0 failures in tests
    0 failures in setup code
    0 failiuers in cleanup code
build succeeded.

Testing of doctests in the sources finished, look at the results in build/doctest/output.txt.
```

Results are also saved into a file that is placed to build/doctest/ directory

There is handy shortcut in main directory that will execute both and also gather test coverage metrics from nosetests:

```
suite.py
```

Coverage report is placed in cover - directory.

Behave

Navigate to directory containing tests written with behave (behave) and issue command:

```
behave
```

This will start behave and run all tests. Results for each feature are displayed on screen and finally a summary is shown:

```
2 features passed, 0 failed, 0 skipped
3 scenarios passed, 0 failed, 0 skipped
21 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.0s
```

1.15.3 Writing tests

Unit tests

Unit tests are placed in package `pyherc.test.unit` Any module that is named as "test_*" will be inspected automatically by Nose when it is gathering tests to run. It will search for classes named "Test*" and methods named "test_*".

Following code is simple test that creates EffectHandle object and tries to add it into EffectsCollection object. Then it verifies that it actually was added there.

```python
from pyherc.data.effects import EffectsCollection
from pyherc.test.builders import EffectHandleBuilder
from hamcrest import *
from pyherc.test.matchers import has_effect_handle

class TestEffectsCollection(object):

    def __init__(self):
        super(TestEffectsCollection, self).__init__()
        self.collection = None

    def setup(self):
        """
        Setup test case
        """
        self.collection = EffectsCollection()

    def test_adding_effect_handle(self):
        """
        Test that effect handle can be added and retrieved
        """
        handle = EffectHandleBuilder().build()

        self.collection.add_effect_handle(handle)

        assert_that(self.collection, has_effect_handle(handle))

test_class = TestEffectsCollection()
test_class.setup()
test_class.test_adding_effect_handle()
```

Interesting parts of the test are especially the usage of EffectHandleBuilder to create the EffectHandle object and the customer has_effect_handle matcher.

Builders are used because they make setting up objects easy, especially when dealing with very complex objects (Character for example). They are placed at `pyherc.test.builders` module.

Custom matchers are used because they make dealing with verification somewhat cleaner. If the internal implementation of class changes, we need to only change how builders construct it and how matchers match it and tests should not need any modifications. Custom matchers can be found at `pyherc.test.matchers` module.

Three macros are provided to help reduce boilerplate from tests: `background`, `fact` and `with-background`. Background is used to create setup function. It can return one or more symbols for tests:

```
(require archimedes)

(background weapons
        [item (-> (ItemBuilder)
                  (.with-damage 2 "piercing")
                  (.with-name "club")
                  (.build))]
        [character (-> (CharacterBuilder)
                       (.build))]
        [_ (set-action-factory (-> (ActionFactoryBuilder)
                                   (.with-inventory-factory)
                                   (.build)))])
```

The example code creates background called `weapons` and initializes it with `item` and `character` symbols. In addition, `set-action-factory` is called for side effect.

Facts are executable tests, that can be standalone, or use previously defined background. When using a background, a list of symbols to retrieved is given to `with-background` macro. This will generate a call to background and retrieve specified symbols to current scope:

```
(fact "character can wield weapon"
  (with-background weapons [item character]
    (equip character item)
    (assert-that character.inventory.weapon (is- (equal-to item)))))
```

Each fact should have unique description, since it is used to generate name for test function.

Cutesy

Cutesy is an internal domain specific language. Basically, it's just a collection of functions that can be used to contruct nice looking tests. Theory is that these easy to read tests can be used to communicate what the system is supposed to be doing on a high level, without making things complicated with all the technical details.

Here's an example, how to test that getting hit will cause hit points to go down.

```python
from pyherc.test.cutesy import strong, Adventurer
from pyherc.test.cutesy import weak, Goblin
from pyherc.test.cutesy import Level

from pyherc.test.cutesy import place, middle_of
from pyherc.test.cutesy import right_of
from pyherc.test.cutesy import make,  hit

from hamcrest import assert_that
from pyherc.test.cutesy import has_less_hit_points

class TestCombatBehaviour():

    def test_hitting_reduces_hit_points(self):
        Pete = strong(Adventurer())
        Uglak = weak(Goblin())

        place(Uglak, middle_of(Level()))
        place(Pete, right_of(Uglak))
        make(Uglak, hit(Pete))

        assert_that(Pete, has_less_hit_points())

test = TestCombatBehaviour()
test.test_hitting_reduces_hit_points()
```

Tests written with Cutesy follow same guidelines as regular unit tests. However they are placed in package `pyherc.test.bdd`

Doctest

Doctest tests are written inside of .rst documents that are used to generate documentation (including this one you are currently reading). These documents are placed in doc/api/source folder and folders inside it.

`.. testcode::` Starts test code block. Code example is placed inside this one.

`.. testoutput::` Is optional block. It can be omitted if it is enough to see that the code example can be executed. If output of the example needs to be verified, expected output is placed here.

Nosetest example earlier in this document is also a doctest example. If you view source of this page, you can see how it has been constructed.

More information can be found at Sphinx documentation.

Behave

Tests with behave are placed under directory behave/features. They consists of two parts: feature-file specifying one or more test scenarios and python implementation of steps in feature-files.

The earlier Cutesy example can be translated to behave as follows:

```
Feature: Combat
  as an character
  in order to kill enemies
  I want to damage my enemies

  Scenario: hit in unarmed combat
     Given Pete is Adventurer
       And Uglak is Goblin
       And Uglak is standing in room
       And Pete is standing next to Uglak
      When Uglak hits Pete
      Then Pete should have less hitpoints
```

Each of the steps need to be defined as Python code:

```
@given(u'{character_name} is Adventurer')
def impl(context, character_name):
    if not hasattr(context, 'characters'):
        context.characters = []
    new_character = Adventurer()
    new_character.name = character_name
    context.characters.append(new_character)
```

It is advisable not to reimplement all the logic in behave tests, but reuse existing functionality from Cutesy. This makes tests both faster to write and easier to maintain. For more information on using behave, have a look at their online tutorial.

Release notes

2.1 Release 0.1

2.1.1 New features

- initial release

2.1.2 Fixed bugs

- None

2.1.3 Other notes

- None

2.2 Release 0.2

2.2.1 New features

- New area, crypt
- Debug server, point your browser to http://localhost:8080/ to see it

2.2.2 Fixed bugs

- Monsters can no longer enter same location as the player

2.2.3 Other notes

- pyDoubles switched to mockito
- logging is done via aspects

2.3 Release 0.3

2.3.1 New features

- Potions now affect characters for multiple turns

2.3.2 Fixed bugs

- None

2.3.3 Other notes

- various builders can now be used in testing
- more hamcrest matchers were added

2.4 Release 0.4

2.4.1 New features

- Certain creatures can make poisoned attacks
- First version of Cutesy testing language included

2.4.2 Fixed bugs

- None

2.4.3 Other notes

- get_next_creature does not produce debug log anymore
- very rudimentary monster spawning added to debug server
- very rudimentary item spawning added to debug server
- documentation regarding to testing added
- internals of inventory handling improved
- improved internals of user interface
- tests are grouped by function (unit, integration, acceptance)
- IntegrationTest class has been removed

2.5 Release 0.5

2.5.1 New features

New features that are readily visible to players:

- User interface rewrite with PyQt
- 16 inventory window
- Message is shown for missed attack
- Message is shown for dying monster
- Message is shown for picked up item
- Message is shown for dropped item
- Player character can be given a name

Following new features are more technical in nature and not visible during gameplay:

- _at function added to Cutesy
- is_dead matcher added
- other components can register to receive updates from domain objects
- pyherc.rules.items.drop replaced with DropAction

2.5.2 Fixed bugs

- 17 Taking stairs do not update display correctly

2.5.3 Other notes

- Services are no longer injected to domain objects
- pyherc.rules.effects moved to pyherc.data.effects
- EffectsCollection moved to pyherc.data.effects
- qc added for testing
- poisoning and dying from poison tests moved to BDD side
- is_at and is_not_at changed to is_in and is_not_in
- herculeum.gui.core removed
- PGU and pygame removed as dependencies

2.6 Release 0.6

2.6.1 New features

- Support for Qt style sheets
- Splash screen at start up

- icons can be specified in level specific configuration scripts
- new weapons added
- new inventory screen
- player can drink potions
- on-screen counters to show damage, healing and status effects
- player can wield and unwield weapons

2.6.2 Fixed bugs

- 22 python path is not modified before first imports
- 19 mdi user interface is clumsy to use

2.6.3 Known bugs

- 26 spider poisons in combat even when it misses
- 25 dying should make game to return to main screen
- 21 PyQt user interface does not support line of sight
- 18 Entities created by debug server are not shown on map
- 5 Raised events are not filtered, but delivered to all creatures

2.6.4 Other notes

- behave taken into use for BDD
- testing guidelines updated
- "{character_name} is almost dead" added to behave
- pyherc.rules.magic package removed

2.7 Release 0.7

2.7.1 New features

- damage is shown negative in counters
- weapons deal different types of damage
- split damage is supported
- more streamlined user interface
- status effects are shown on main screen
- 32 view to show player character
- 31 better ai for skeleton warrior
- 30 showing hit points of player

- 29 being weak against damage
- 28 damage resistance
- 24 skeleton warrior

2.7.2 Fixed bugs

- 34 Split damage weapons do not show full damage on screen
- 33 using stairs while there is damage counter on screen crashes game
- 27 dropping a weapon in use retains the weapon in use
- 18 bug: Entities created by debug server are not shown on map

2.7.3 Known bugs

- 26 bug: spider poisons in combat even when it misses
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 10 bug: Player character creation has hard coded values
- 9 bug: Attacks use hard coded time
- 5 bug: Raised events are not filtered, but delivered to all creatures
- 3 bug: FlockingHerbivore has no memory

2.7.4 Other notes

- web.py is not required unless using debug server

2.8 Release 0.8

2.8.1 New features

- amount of damage done is reported more clearly
- new area: Crimson Lair
- weapons may have special effects that are triggered in combat
- 45 feature: ranged combat
- 44 feature: armours
- 43 feature: support for vi and cursor keys
- 40 feature: executable for Windows
- 39 feature: the Tome of Um'bano
- 37 feature: creating a new character
- 36 feature: escaping the dungeon

- 35 feature: crimson jaw
- equiping and unequiping raise events

2.8.2 Fixed bugs

- 26 bug: spider poisons in combat even when it misses
- 10 bug: Player character creation has hard coded values

2.8.3 Known bugs

- 42 bug: character generator generates incorrect amount of items in inventory
- 38 bug: damage effect does not take damage modifiers into account
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 9 bug: Attacks use hard coded time
- 5 bug: Raised events are not filtered, but delivered to all creatures

2.8.4 Other notes

- 41 player character configuration
- Aspyct is no longer needed to run the game
- behave tests moved under src/pyherc/test/BDD
- parts of the manual are generated directly from game data

2.9 Release 0.9

2.9.1 New features

- 46 curses interface

2.9.2 Fixed bugs

- 48 bug: Effects with None as duration or frequency cause crash when triggered

2.9.3 Known bugs

- 42 bug: character generator generates incorrect amount of items in inventory
- 38 bug: damage effect does not take damage modifiers into account
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 9 bug: Attacks use hard coded time

- 5 bug: Raised events are not filtered, but delivered to all creatures

2.9.4 Other notes

- 47 switch to Python 3

2.10 Release 0.10

2.10.1 New features

- new set of graphics and animations
- regular movement and attack can be done only to cardinal directions
- characters can wait for a bit without doing anything
- new player character, mage
- 68 feature: change direction of character when walking

2.10.2 Fixed bugs

- 72 bug: moving does not take armour into account
- 69 bug: layering of icons
- 54 bug: weapons with multiple damage types cause attacker to move
- 9 bug: Attacks use hard coded time

2.10.3 Known bugs

- 42 bug: character generator generates incorrect amount of items in inventory
- 38 bug: damage effect does not take damage modifiers into account
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 5 bug: Raised events are not filtered, but delivered to all creatures

2.10.4 Other notes

- 53: moved many actions (moving, combat, etc) from Character class to separate functions

2.11 Release 0.11

2.11.1 New features

- 77 feature: swapping places
- 73 update to latest version of Hy

- 65 feature: cleaner AI
- 62 feature: pits
- ability to specify starting level on command line

2.11.2 Fixed bugs

- 76 bug: it is impossible use stairs, if there is a creature standing on the other end

2.11.3 Known bugs

- 42 bug: character generator generates incorrect amount of items in inventory
- 38 bug: damage effect does not take damage modifiers into account
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 5 bug: Raised events are not filtered, but delivered to all creatures

2.11.4 Other notes

2.12 Release 0.12

2.12.1 New features

- 81 restructure dungeon layout
- 66 feature: animation system

2.12.2 Fixed bugs

- 86 bug: patrol AI sometimes gets very confused
- 82 bug: bug: if player is the last character in level, dying will put game into an infinite loop
- 42 bug: character generator generates incorrect amount of items in inventory
- 38 bug: damage effect does not take damage modifiers into account

2.12.3 Known bugs

- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight
- 5 bug: Raised events are not filtered, but delivered to all creatures

2.12.4 Other notes

2.13 Release 0.13

2.13.1 New features

- 84 feature: dragon de platino
- 83 prototype using dictionary instead of list of lists for level structure
- 15 feature: fungus

2.13.2 Fixed bugs

- 5 bug: Raised events are not filtered, but delivered to all creatures

2.13.3 Known bugs

- 89 bug: CharacterBuilder does not add character to given level
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight

2.13.4 Other notes

2.14 Release 0.14

2.14.1 New features

- 112 better installation instructions

2.14.2 Fixed bugs

- 114 fix setup.py
- 111 bug: mitosis can create creatures on traps, without triggering them
- 110 bug: creatures and stairs are sometimes placed on top of traps
- 96 bug: two characters switching places cause level to be in inconsistent state

2.14.3 Known bugs

- 113 bug: items dropped in pit are floating
- 97 bug: level generation sometimes places multiple characters in same location
- 89 bug: CharacterBuilder does not add character to given level
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight

2.14.4 Other notes

- 109 event system restructure

2.15 Release 0.15

2.15.1 New features

- None

2.15.2 Fixed bugs

- None

2.15.3 Known bugs

- 113 bug: items dropped in pit are floating
- 97 bug: level generation sometimes places multiple characters in same location
- 89 bug: CharacterBuilder does not add character to given level
- 25 bug: dying should make game to return to main screen
- 21 bug: PyQt user interface does not support line of sight

2.15.4 Other notes

- None

CHAPTER 3

Indices and tables

- genindex
- search