pyherc Documentation

Release 0.14.1

Tuukka Turto

January 30, 2016

Contents

1 Intro **3**

2 Building blocks **5**

3 Main components **9**
3.1 Model . . . 9
3.2 Character . . . 9
3.3 Dungeon . . . 9
3.4 Level . . . 9
3.5 Rules . . . 10

4 Generating a level **11**
4.1 Overview of generating dungeon . . . 11
4.2 Adding a new type of level . . . 11

5 Modular level generator **13**
5.1 Overview of LevelGenerator . . . 13
5.2 Partitioners . . . 14
5.3 Room generators . . . 14
5.4 Decorators . . . 14
5.5 Portal adders . . . 15
5.6 Creature adder . . . 15
5.7 Item adder . . . 15
5.8 Defining levels . . . 15

6 Generating an item **19**
6.1 Overview of generating item . . . 19
6.2 Defining items . . . 19

7 Actions **21**
7.1 Overview of Action system . . . 21
7.2 Action creation during play . . . 21
7.3 Adding a new type of action . . . 21

8 Events **25**
8.1 Overview of event system . . . 25
8.2 Events and UI . . . 25

9 Effects **27**

9.1 Overview of effects system . . . 27
9.2 Effect handles . . . 27
9.3 Effect . . . 27
9.4 Creating Effects . . . 27
9.5 Effects collection . . . 28

10 Configuration **31**
10.1 Configuration scripts . . . 31
10.2 Level configuration . . . 31
10.3 Item configuration . . . 31
10.4 Character configuration . . . 31
10.5 Player characters . . . 32
10.6 Effects configuration . . . 32
10.7 Handling icons . . . 32

11 Magic **33**
11.1 Overview of Magic system . . . 33
11.2 How spells are cast . . . 34
11.3 Spell creation during play . . . 34
11.4 Adding a new type of spell . . . 34

12 Error Handling **35**
12.1 General idea . . . 35
12.2 Specific cases . . . 35

13 Testing **37**
13.1 Overview of testing . . . 37
13.2 Running tests . . . 37
13.3 Writing tests . . . 38

14 Indices and tables **43**

Contents:

CHAPTER 1

Intro

This guide has been split to two parts. First of them is programmer's guide that shows how you can accomplish various things with the codebase and how things work.

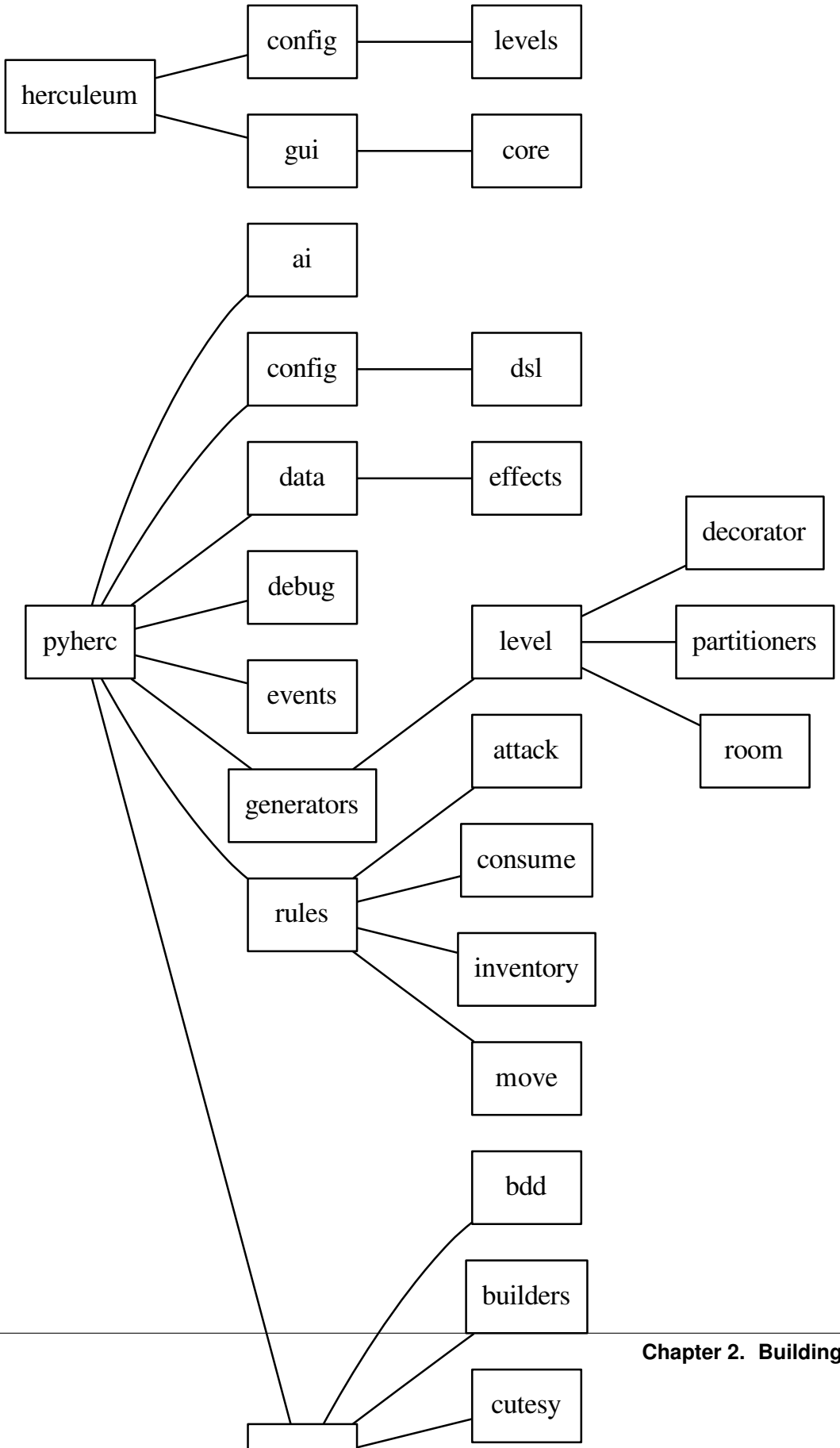
Second part is programmer's reference guide which lists packages, modules and classes that make up the pyherc codebase.

CHAPTER 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.

On a high level, pyherc and herculeum codebases are divided as detailed below:


herculeum
config
levels
gui
core
pyherc
ai
config
dsl
data
effects
debug
events
generators
level
decorator
partitioners
room
attack
consume
rules
inventory
move
bdd
builders
cutesy

Convenient links to each of main components

pyherc:

- pyherc
- pyherc.ai
- pyherc.config
- pyherc.data
- pyherc.debug
- pyherc.events
- pyherc.generators
- pyherc.rules
- pyherc.test

herculeum:

- herculeum
- herculeum.config
- herculeum.gui

CHAPTER 3

Main components

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

3.2 Character

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

- Stats
- Inventory
- Location

3.3 Dungeon

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

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

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.

CHAPTER 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.

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`.

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.

4.2.1 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:

```
def 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 = Level((80, 40), tiles.FLOOR_ROCK, tiles.WALL_GROUND)
```

This call will instantiate a Level object, set it size to be 80 times 40, create floor of rock and fill the whole level will ground wall. After this the generator can create structure of the level as wanted.

```
for y_loc in range(1, 39):
    for x_loc in range(1, 79):
        new_level.walls[x_loc][y_loc] = tiles.WALL_EMPTY
```

Adding monsters

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

```
monster = self.creature_generator.generate_creature(
                                          model.tables, {'name':'rat'})
new_level.add_creature(monster, new_level.find_free_space())
```

This will instruct `pyherc.generators.creature.CreatureGenerator` to use supplied monster tables and create a monster called 'rat'. After this, the rat is added at a random free location.

Adding items

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

```
new_item = self.item_generator.generateItem(model.tables, {'type':'food'})
new_item.location = new_level.find_free_space()
new_level.items.append(new_item)
```

This will instruct `pyherc.generators.item.ItemGenerator` to use supplied item tables and create random food type item. After this the item is added to the level. This portion of the Level interface will most likely change in the future, to match better to the interface used to add monsters.

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:

```
if portal != None:
    new_portal = Portal()
    new_portal.model = model
    new_level.add_portal(new_portal, new_level.find_free_space(), 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.

CHAPTER 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.generator.LevelGenerator` is a high level generator that can be used to create different kinds of levels. Usually the system does not directly use it, but queries a fully setup generator from `pyherc.generators.level.generator.LevelGeneratorFactory`

5.1 Overview of LevelGenerator

```
def generate_level(self, portal, model, new_portals = 0, level=1, room_min_size = (2, 2)):
```

LevelGenerator has same generate_level - method as other level generators and calling it functions in the same way. Difference is in the internals of the generator. Instead of performing all the level generation by itself, LevelGenerator breaks it into steps and delegates each step to a sub component.

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.

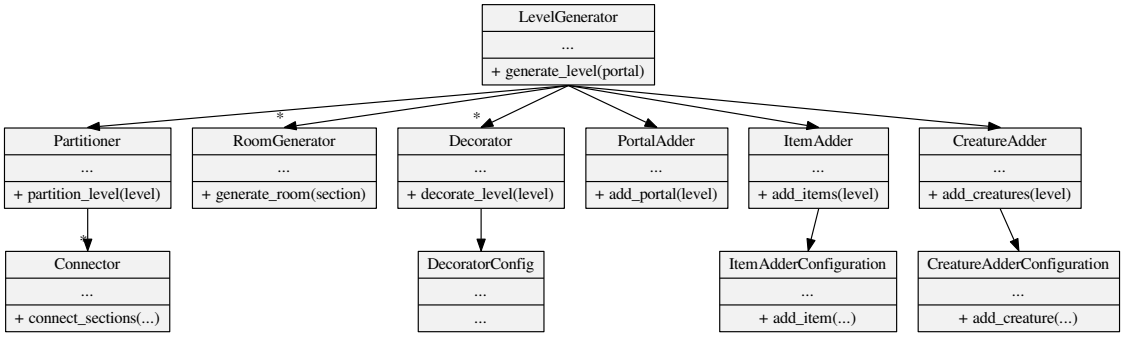
A room is generated within each section and corridors are used to link rooms to neighbouring sections. Linking is done according to links set up in the previous phase. This in turn ensures that each room is reachable.

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

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.

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.



5.2 Partitioners

`pyherc.generators.level.partitioners.grid.GridPartitioner` is basic partitioner, which knows how to divide level into a grid with equal sized sections.

GridPartitioner has method:

```
def partition_level(self, level,  x_sections = 3,  y_sections = 3):
```

Calling this method 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.

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 only needs a single method:

```
def generate_room(self, section):
```

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

Simple example can be found from `pyherc.generators.level.room.squareroom.SquareRoomGenerator`

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.

New decorator can be created by subclassing `pyherc.generators.level.decorator.basic.Decorator` and overriding method:

```
def decorate_level(self, level):
```

5.5 Portal adders

`pyherc.generators.level.portals.PortalAdder` is responsible class for generating portals. the class itself is pretty simple. It contains information of what kind of icons to use, where to place the portal (room, corridor, treasure chamber and so on) and what kind of level it will lead to.

Method:

```
def add_portal(self, level):
```

Will create a proxy portal at a random location. This portal will contain name of the level, instead of direct link. When player enters the portal, a new level generator is created and used to generate the new level.

5.6 Creature adder

Creatures are added with `pyherc.generators.level.creatures.CreatureAdder`. Usually there is no reason to subclass this class, but simple configuration is enough.

5.7 Item adder

Items are added with `pyherc.generators.level.items.ItemAdder`. Usually there is no reason to subclass this class, but simple configuration is enough.

5.8 Defining levels

Levels are defined in configuration scripts that are fed to `pyherc.config.config.Configuration` during system startup. Following example defines an simple level:

```
from random import Random
from pyherc.generators.level.partitioners import GridPartitioner
from pyherc.generators.level.room import SquareRoomGenerator

from pyherc.generators.level.decorator import ReplacingDecorator
from pyherc.generators.level.decorator import ReplacingDecoratorConfig
from pyherc.generators.level.decorator import WallBuilderDecorator
from pyherc.generators.level.decorator import WallBuilderDecoratorConfig
from pyherc.generators.level.decorator import AggregateDecorator
from pyherc.generators.level.decorator import AggregateDecoratorConfig

from pyherc.generators.level.items import ItemAdderConfiguration, ItemAdder
from pyherc.generators.level.creatures import CreatureAdderConfiguration
from pyherc.generators.level.creatures import CreatureAdder

from pyherc.generators.level.portals import PortalAdderConfiguration

from pyherc.config.dsl import LevelConfiguration, LevelContext
```

```
def init_level(rng, item_generator, creature_generator, level_size, context):
    room_generators = [SquareRoomGenerator('FLOOR_NATURAL',
                                           'WALL_EMPTY',
                                           'FLOOR_NATURAL',
                                           ['upper crypt']),
                       SquareRoomGenerator('FLOOR_CONSTRUCTED',
                                           'WALL_EMPTY',
                                           'FLOOR_CONSTRUCTED',
                                           ['upper crypt'])]
    level_partitioners = [GridPartitioner(['upper crypt'],
                                          4,
                                          3,
                                          rng)]

    replacer_config = ReplacingDecoratorConfig(['upper crypt'],
                                       {'FLOOR_NATURAL': 'FLOOR_ROCK',
                                        'FLOOR_CONSTRUCTED': 'FLOOR_BRICK'},
                                       {'WALL_NATURAL': 'WALL_GROUND',
                                        'WALL_CONSTRUCTED': 'WALL_ROCK'})
    replacer = ReplacingDecorator(replacer_config)

    wallbuilder_config = WallBuilderDecoratorConfig(['upper crypt'],
                                         {'WALL_NATURAL': 'WALL_CONSTRUCTED'},
                                          'WALL_EMPTY')
    wallbuilder = WallBuilderDecorator(wallbuilder_config)

    aggregate_decorator_config = AggregateDecoratorConfig(['upper crypt'],
                                                          [wallbuilder,
                                                          replacer])

    decorators = [AggregateDecorator(aggregate_decorator_config)]

    item_adder_config = ItemAdderConfiguration(['upper crypt'])
    item_adder_config.add_item(min_amount = 2,
                               max_amount = 4,
                               type = 'weapon',
                               location = 'room')
    item_adder_config.add_item(min_amount = 2,
                               max_amount = 4,
                               type = 'potion',
                               location = 'room')
    item_adder_config.add_item(min_amount = 0,
                               max_amount = 5,
                               type = 'food',
                               location = 'room')
    item_adders = [ItemAdder(item_generator,
                             item_adder_config,
                             rng)]

    creature_adder_config = CreatureAdderConfiguration(['upper crypt'])

    creature_adder_config.add_creature(min_amount = 4,
                                       max_amount = 8,
                                       name = 'spider')

    creature_adders = [CreatureAdder(creature_generator,
                                     creature_adder_config,
                                     rng)]
```

```
    portal_adder_configurations = [PortalAdderConfiguration(
                                        icons = ('PORTAL_STAIRS_DOWN',
                                                 'PORTAL_STAIRS_UP'),
                                        level_type = 'upper catacombs',
                                        location_type = 'room',
                                        chance = 25,
                                        new_level = 'upper crypt',
                                        unique = True)]

    level_context = LevelContext(size = level_size,
                                 floor_type = 'FLOOR_NATURAL',
                                 wall_type = 'WALL_NATURAL',
                                 level_types = ['upper crypt'])

    config = (LevelConfiguration()
                    .with_rooms(room_generators)
                    .with_partitioners(level_partitioners)
                    .with_decorators(decorators)
                    .with_items(item_adders)
                    .with_creatures(creature_adders)
                    .with_portals(portal_adder_configurations)
                    .with_contexts([level_context])
                    .build())
    return config

rng = Random()
item_generator = None
creature_generator = None
level_size = (80, 60)
config_context = object()

config = init_level(rng, item_generator, creature_generator, level_size, config_context)

print(config)
```

The example defines function to initialise a level configuration and executes it. In real life scenarion, item_generator and creature_generator objects would have been initialised before supplying them to configuration function, but it was omitted from the brevity's sake in this example.

Parameters config_context is an extension hook that can be used to deliver application specific information that needs to be transfered between the application and configuration.

```
<pyherc.generators.level.config.LevelGeneratorFactoryConfig object at 0x...>
```

CHAPTER 6

Generating an item

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

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.

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.

CHAPTER 7

Actions

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

7.1 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.

7.2 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:

```
action = self.action_factory.get_action(MoveParameters(self,
                                                        direction,
                                                        'walk'))
action.execute()
```

This creates a WalkAction and executes it, causing the character to take a single step to given direction.

7.3 Adding a new type of action

Lets say we want to create an action that allows characters to wait for specific amount of ticks.

7.3.1 Overview

Actions are located in `pyherc.rules`. Custom is to have own package for each type of action. For example, code related to moving is placed in `pyherc.rules.move` while code for various attacks is in `pyherc.rules.attack`. Parameter classes are placed to `pyherc.rules.public` and imported to top level for ease of use.

Last important piece is factory that knows how to read parameter class and construct action class based on it. Factories are placed in the same location as the actions.

7.3.2 Creating a sub action factory

Sub action factory is used to create specific types of actions. Start by inheriting it and overriding type of action it can be used to construct:

```
from pyherc.rules.factory import SubActionFactory

class WaitFactory(SubActionFactory):

    def __init__(self):
        self.action_type = 'wait'
```

Next we need to define factory method that can actually create our new action:

```
def get_action(self, parameters):
    wait_time = parameters.wait_time
    target = parameters.target
    return WaitAction(target, wait_time)
```

7.3.3 Defining new parameter class

WaitParameters class is very simple, almost could do without it even:

```
class WaitParameters(ActionParameters):

    def __init__(self, target, wait_time):
        self.action_type = 'wait'
        self.target = target
        self.wait_time = wait_time
```

Constructor takes two parameters: target who is character doing the waiting and wait_time, which is amount of ticks to wait. action_type is used by the factory system to determine which factory should be used to create action based on parameter class. It should match to the action_type we defined in WaitFactory constructor.

7.3.4 Creating the new action

WaitAction is not much more complex:

```
class WaitAction(object):

    def __init__(self, target, wait_time):
        self.target = target
        self.wait_time = wait_time

    def is_legal(self):
        return True

    def execute(self):
        self.target.tick = self.target.tick + self.wait_time
```

Constructor is used to create a new instance of WaitAction, with given Character and wait time.

is_legal can be called by system before trying to execute the action, in order to see if it can be safely done. We did not place any validation logic there this time, but one could check for example if the character is too excited to wait.

Calling execute will trigger the action and in our case increment internal timer of the character. This will effectively move his turn further in the future.

7.3.5 Configuring ActionFactory

`pyherc.rules.public.ActionFactory` needs to be configured in order it to be able to create our new WaitAction. This is done in `pyherc.config.Configuration`:

```
wait_factory = WaitFactory()

self.action_factory = ActionFactory(
                                      self.model,
                                      [move_factory,
                                      attack_factory,
                                      drink_factory,
                                      wait_factory])
```

7.3.6 Adding easy to use interface

Last finishing step is to add easy to use method to Character class:

```
def wait(self, ticks, action_factory):
    action = action_factory.get_action(WaitParameters(self,
                                                      ticks))
    action.execute()
```

Now we can have our character to wait for a bit, just by calling:

```
player_character.wait(5, action_factory)
```

Notice how we are passing ActionFactory from outside of Character objects, instead of defining it as an attribute of Character. We do not want to inject service objects into domain objects, because it would complicate saving and loading later on.

7.3.7 Whole code

Below is shown the whole example of wait action and demonstration how it changes value in character's internal clock.

```
from pyherc.data import Character, Model
from pyherc.rules import ActionFactory, ActionParameters
from pyherc.rules.factory import SubActionFactory
from random import Random

class WaitParameters(ActionParameters):

    def __init__(self, target, wait_time):
        self.action_type = 'wait'
        self.target = target
        self.wait_time = wait_time

class WaitAction(object):

    def __init__(self, target, wait_time):
        self.target = target
        self.wait_time = wait_time

    def is_legal(self):
        return True
```

```
    def execute(self):
        self.target.tick = self.target.tick + self.wait_time

class WaitFactory(SubActionFactory):

    def __init__(self):
        self.action_type = 'wait'

    def get_action(self, parameters):
        wait_time = parameters.wait_time
        target = parameters.target
        return WaitAction(target, wait_time)

model = Model()
wait_factory = WaitFactory()
action_factory = ActionFactory(model = model,
                               factories = [wait_factory])
character = Character(model)
action = character.create_action(WaitParameters(character, 5),
                                 action_factory)

print('Ticks {0}'.format(character.tick))
action.execute()
print('Ticks after waiting {0}'.format(character.tick))
```

The output shows how it is character's turn to move (tick is 0), but after executing wait action, the tick is 5.

```
Ticks 0
Ticks after waiting 5
```

CHAPTER 8

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.

8.1 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.

8.2 Events and UI

One important factor of events are their ability to tell UI what pieces of map were affected and should be redrawn next time the screen is updated. This is done by passing a list of coordinates (int, int) in affected_tiles parameter that each and every event has in constructor.

Events that modify location of characters, shape of the map or otherwise affect to the world visible to the player should list the affected tiles for fast UI update. For example, move action will usually have two tiles in the list: the original location of character and the new location. If character were carrying a lightsource when moving, appropriate squares should be listed too (those that were lighted, but are now in darkness and vice versa).

CHAPTER 9

Effects

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

9.1 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.

9.2 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.

9.3 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.

9.4 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.

9.5 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.

```
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.

```
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...>]
```

CHAPTER 10

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`.

10.1 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)
```

10.2 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.

10.3 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.

10.4 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.

10.5 Player characters

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

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

10.6 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.

10.7 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`.

CHAPTER 11

Magic

This section will outline how spells are implemented.

11.1 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

11.2 How spells are cast

11.3 Spell creation during play

11.4 Adding a new type of spell

11.4.1 Overview

11.4.2 Whole code

CHAPTER 12

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.

12.1 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.

12.2 Specific cases

12.2.1 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()`

CHAPTER 13

Testing

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

13.1 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.

13.2 Running tests

13.2.1 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.

13.2.2 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 failuers 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.

13.2.3 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
```

13.3 Writing tests

13.3.1 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.

```
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.

13.3.2 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.

```
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`

13.3.3 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.

13.3.4 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.

CHAPTER 14

Indices and tables

- genindex
- search