ViCE Documentation

Release 0.0.1

Edwin Marshall

May 19, 2012

CONTENTS

I Preface **3**

1 User Roles **7**

2 How This Manual is Organized **9**

3 Design Principles **11**

4 Openness **13**

5 Portability **15**

6 Modularity **17**

II Player Guide **19**

III Designer Guide **23**

IV Developer Guide **27**

7 Architecture **31**
- 7.1 Plugin Framework 31
- 7.2 Database Layer 31

V API Reference **33**

8 vice – Main package, also provides PropertyDict **35**

9 vice.database – Database abstraction and integration **37**

10 vice.plugins – Infrastructure for implementing plugins **39**
- 10.1 vice.plugins.actions – Built-in action plugins 39
- 10.2 vice.plugins.items – Built-in item plugins 40

Index **41**

Welcome to ViCE's official user manual!

Part I

Preface

ViCE (Virtual Card-game Engine) is an open source, portable, modular framework for playing and creating TCGs (Trading Card Games). This manual outlines ViCE's design principles, describes its various features from three perspectives (which we refer to as roles), and even includes reference documentation for the public API.

Note: While there are subtle differences between the Trading Card Games and Collectible Card Games, in the interest of simplicity, this manual refers to them collectively (no pun intended) as TCGs.

CHAPTER

ONE

USER ROLES

ViCE is a framework suitable for use by users of varying interests and technical backgrounds. In an effort to help facilitate learning, three key user roles have been identified, as follows:

Players These are the individuals who aren't concerned with how ViCE's internals work, and could care less about creating new or porting existing :abbr:TCGs to the framework. There sole reason for using ViCE is to play whatever games *do* exist for ViCE.

Designers These are the individuals who are not interested in the implementation details of ViCE's internals, but would like to learn enough to enable them to either create new TCGs or port existing ones to ViCE.

Developers These are the individuals who are interested in *how* ViCE works, for the sake of implementing new features not possible through the currently available API or fixing bugs.

While we have defined three distinct roles, these roles are not mutually exclusive, and can be viewed as somewhat hierarchical. That is, an individual who categorizes himself as a developer may also categorize himself as a designer and player.

CHAPTER

TWO

HOW THIS MANUAL IS ORGANIZED

This book is organized into four parts, the first three of which being guides modeled after their respective roles:

1. Player Guide
2. Designer Guide
3. Developer Guide
4. API Reference

CHAPTER

THREE

DESIGN PRINCIPLES

In this section, we'll talk about the core principles that we adhere to when developing, designing, and playing ViCE.

CHAPTER

FOUR

OPENNESS

ViCE is open source software licensed under the Affero General Public License, which means that you can use it, redistribute it, and even modify it without any legal ramifications, so long as you abide by the terms of the license. This also guarantees that as improvements are made, whether it be to ViCE's upstream code base or to a fork, that if those improvements are made public, they are made without any monetary obligation. That is, you should never have to pay for new features or corrected defects.

CHAPTER

FIVE

PORTABILITY

ViCE is written using a variety of cross-platform technologies, and as such, every effort is made to ensure that ViCE runs *natively* on at least Linux, Mac OS, and Windows.

CHAPTER

SIX

MODULARITY

Most of ViCE's features are implemented on top of its plugin architecture to allow for maximum extensibility. Not only does this imply that new features can be added rather easily, but also that features may be used a la carte: if a feature isn't required for you to finish a plugin, it doesn't need to be used.

Part II

Player Guide

Welcome to ViCe's official player guide. It describes ViCE's UI so that you can quickly get started playing your favorite TCGs on ViCE, find other players, and even run your own ViCE server so that you can host tournaments or keep track of stats.

Part III

Designer Guide

Welcome to ViCe's official designer guide. It describes ViCE's high-level components so that you can port your favorite TCG's to ViCE, improve the quality of existing ports, or create a new TCG from scratch.

Part IV

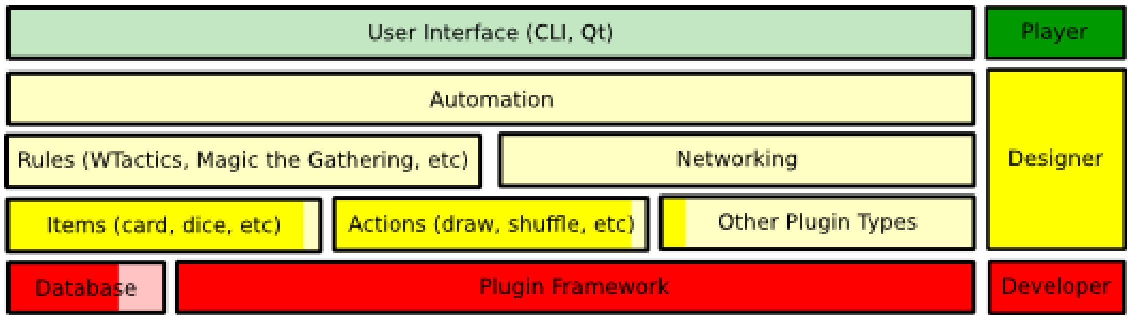
Developer Guide

This guide discusses the underlying patterns and idioms used when developing ViCE.

CHAPTER

SEVEN

ARCHITECTURE



As you can see from the architecture stack diagram above, ViCE's core is composed of the plugin framework and database layer [1]. Item, Action, and other, yet to be developed plugin types sit directly on these components. While not yet started, other features and abstractions suitable for designers and players will then be built on top of these.

Note: The opacity of each box represents an approximation of how complete the represented component is.

7.1 Plugin Framework

ViCE's plugin framework is based on the simple fact that all new-style classes [2] in python know about their subclasses. All plugins [3] must be given a name by assigning the class attribute NAME. It is through this name that plugins are identified, and without it, a plugin won't be discoverable.

For information on the different plugin types and their roles, please refer to the *Designer Guide*.

7.2 Database Layer

Note: Not all of SQLAlchemy's API has been abstracted, so some things are not yet possible. For more information, consult the *API Reference*.

The database layer is an abstraction on top of the already excellent abstraction layer SQLAlchemy. While this might seem excessive, it is necessary for three reasons:

[1] Custom data types such as the PropertyDict are also part of ViCE's core, but they were left out of the chart for the sake of simplicity.

[2] http://www.python.org/doc/newstyle/

[3] Only plugins which are meant to be instantiated need assign the NAME class attribute. That is, plugin base classes should *not* assign this attribute.

1. Flexibility: Tucking the implementation details behind a simple API allows us to not only change the underlying module used if we ever decide to, but also selectively reimplement features that we feel aren't covered well by the underlying module.
2. Brevity: ViCE's database API is much more concise than SQLAlchemy alone, serves to simplify code, as well as the learning of the API itself.
3. Seamless Integration: Since the database layer sits next to the plugin framework and beneath all other components, it's tightly integrated [4] with the rest of the framework.

Currently, the database layer is not implemented as a plugin because SQLAlchemy provides a unified API on top of many RDBMSs (Relational Database Management Systems).

See Also:

- *vice – Main package, also provides PropertyDict*
- *vice.database – Database abstraction and integration*
- *vice.plugins – Infrastructure for implementing plugins*

[4] Note that we did not say "tightly coupled". As such, it is possible for alternate implementations to be used.

Part V

API Reference

CHAPTER

EIGHT

VICE – MAIN PACKAGE, ALSO PROVIDES PROPERTYDICT

class `vice.`**`PropertyDict`**

A dict subclass that allows values to be retrieved by accessing their keys as properties.

PropertyDict provides a dictionary whose key:value pairs may be assigned in the conventional brace notation (foo['bar'] = 'baz'), or in the more elegant object:property notation (foo.bar = 'baz'). Beyond this subtle addition, they behave identically to regular dictionaries.

CHAPTER

NINE

VICE.DATABASE – DATABASE ABSTRACTION AND INTEGRATION

class `vice.database.`**`Database`**`(`*URI=None*, *echo=False*`)`

Abstraction layer on top of SQLAlchemy's interface.

While SQLAlchemy abstracts the particulars of different database backends and the subtle ways SQL may differ within them, this Database class abstracts the SQLAlchemy API into something more simple, meanwhile adding facilities that make it integrate better with ViCE's plugin architecture.

`connect``(`*URI*, *echo=False*`)`

Connects to an existing database, or creates a new one if one isn't found.

URI may be any URI recognized by SQLAlchemy, and generally follows the form:

```
"<protocol>:///<location>"
```

For example:

```
"sqlite:///wtactics.db"
```

echo determines whether or not SQL statements are echoed to stdout after each operation.

`create_record``(`*table_name*, ***parameters*`)`

Creates a record in table_name, using parameters.

Example:

```
db.create_record('cards',
    # note that id is auto-incremented, so isn't specified
    name = 'Imp',
    atk = 2,
    def_ = 2
)
```

`create_table``(`*table_name*, ***column_attrs*`)`

Creates a table in the database named table_name, with columns whose attributes match column_attrs.

table_name may be anything you wish, but if it so happens to be a reserved word in Python (eg. 'def'), then you must suffix it with an underscore ('_'). You needn't worry, however, since the trailing underscore is removed inside the actual database.

All remaining keyword arguments will be processed as column attributes where the keys should be valid column names, and the values should be arguments to column types.

Valid column types are currently:

- string() – Represents an string column.
- integer() – Represents an integer column .

Arguments may be passed to these column types to further specify restrictions on data or relationships.

Valid arguments are currently: primary_key=True – Marks the column as the primary key.

Example:

```
db.create_table('cards',
    id = integer(primary_key=True),
    name = string(),
    atk = integer(),
    def_ = integer() # notice the trailing underscore
)
```

Note: Since column_attrs is a dictionary, definition order is arbitrary, and thus the order in which the columns is specified may differ from what you expect when examining the resulting database.

select (*tables*, ***kwargs*)
Selects all records of the given tables.

tables is a list of table names.

Warning: The interface is mostly ported directly from SQLAlchemy. In the future, a simpler interface will be implemented, most probably named "find", and this one will be deprecated for immediate removal.

tables
Returns a list of table names for the current database object.

vice.database.**integer** (***kwargs*)
Returns a kwargs dictionary suitable for creating an SQLAlchemy Integer column.

vice.database.**string** (***kwargs*)
Returns a kwargs dictionary suitable for creating an SQLAlchemy String column.

CHAPTER

TEN

VICE.PLUGINS – INFRASTRUCTURE FOR IMPLEMENTING PLUGINS

class `vice.plugins.`**`Plugin`**

Base class for all new plugin types.

All new plugin types are created by subclassing Plugin. In most cases, you will want to inherit from a Plugin subclass rather than directly from Plugin itself.

All new plugins must have a NAME class attribute, which is used mainly for plugin discovery.

classmethod **`plugins`**`()`

Returns a plugin's subclasses as a vice.PropertyDict.

This PropertyDict's keys are plugin names and values are the plugin classes that correspond with those names. A common idiom is to assign the return value to a variable to ease the access to available plugins:

```
foo_plugins = Foo.plugins()
baz = foo_plugins.Baz(x, y)
```

10.1 vice.plugins.actions – Built-in action plugins

class `vice.plugins.actions.`**`Action`**

Callable plugin that provides general operations for Item plugins.

An action is a class which acts like a generic function that operates on Item plugins. This approach is more flexible, extensible, and less repetitious than implementing methods directly within a subclass.

To create a new action, define an Action subclass, override NAME (by convention, lowercase for actions), and finally override __call__:

```
class Foo(Action):
    NAME = 'foo'

    def __call__(cls):
        return 'bar'
```

Alternatively, you may define a simple function and pass that to Action.new:

```
def foo(cls):
    return 'bar'

Action.new(foo)
```

classmethod `new` (*function*)

Convenience method used to help simply creation of new actions.

The function's name is converted to title case and used as the name of the class, and it's original form is used as the Plugin's NAME.

When defining the function, make sure the first argument is cls, since this will be used as the class's __call__ special method

classmethod `plugins` (**args*, ***kwargs*)

Acts similarly to Plugin.plugins(), except that it returns instances of the plugin classes, rather than the classes themselves.

10.2 vice.plugins.items – Built-in item plugins

class `vice.plugins.items.`**`Item`**

Plugin that represents a games tangible objects.

An item is any object within a card game that can be interacted with. The most obvious example of this would be the game's cards, but things such as tokens and dice would be implemented as items as well.

To create a new item, define a new Item subclass, override the NAME attribute (by convention, uppercase for items), and finally override ATTRIBUTES with a sequence of strings:

```
class Card(Item):
    NAME = 'Card'
    ATTRIBUTES = 'name', 'atk', 'def'
```

Alternatively, you may pass an appropriate name and attributes to Item.new:

```
Dice = Item.new('Dice', ('name', 'atk', 'def'))
```

As another alternative, you may pass an appropriate name, valid database table and an optional exclude sequence to Item.fromTable:

```
db = vice.database.Database('sqlite:///wtactics.sqlite')
Card = Item.fromTable('Card', db.cards, exclude=['id'])
```

On instantiation, the values of ATTRIBUTES are converted to properties of the plugin instance. These properties are semi-immutable. That is, on instantiating of an item plugin, you may change the value of existing attributes, but you may not create new ones. If you wish to do so, you should add the new attribute to ATTRIBUTES when defining the class.

classmethod `fromTable` (*name*, *table*, *exclude=None*)

Convenience method used to create new items from database tables.

classmethod `new` (*name*, *attributes*)

Convenience method used to help simplify the creation of new items.

INDEX

A

Action (class in vice.plugins.actions), 39

C

connect() (vice.database.Database method), 37
create_record() (vice.database.Database method), 37
create_table() (vice.database.Database method), 37

D

Database (class in vice.database), 37

F

fromTable() (vice.plugins.items.Item class method), 40

I

integer() (in module vice.database), 38
Item (class in vice.plugins.items), 40

N

new() (vice.plugins.actions.Action class method), 39
new() (vice.plugins.items.Item class method), 40

P

Plugin (class in vice.plugins), 39
plugins() (vice.plugins.actions.Action class method), 40
plugins() (vice.plugins.Plugin class method), 39
PropertyDict (class in vice), 35

S

select() (vice.database.Database method), 38
string() (in module vice.database), 38

T

tables (vice.database.Database attribute), 38

V

vice.database (module), 37
vice.plugins (module), 39
vice.plugins.actions (module), 39
vice.plugins.items (module), 40