Writing testable code (not only) in Python

Martin Sivak

Red Hat

June 13, 2012

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

Table of contents

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

Why this topic?

- Software is getting bigger and more complex
- We have API for external users, but keep forgetting about the internal one
- Our packages have to be maintained for up to **ten years**
- and we hesitate to touch them for fear of breaking something unexpected

The patterns I am about to show are really easy. We just tend to forget to use them.

A bit of motivation

If software had **99.9%** reliability:

- 9 703 cheques would be payed from bad accounts every hour

Those numbers were taken from sofware engineering class lectures vpodzime attended this semester.

Number of Anaconda bugs is derived from about 2 mil. downloads of F16.

A bit of motivation

If software had **99.9%** reliability:

- 9 703 cheques would be payed from bad accounts every hour
- 27 800 letters would be lost every hour

Those numbers were taken from sofware engineering class lectures vpodzime attended this semester.

Number of Anaconda bugs is derived from about 2 mil. downloads of F16.

A bit of motivation

If software had **99.9%** reliability:

- 9 703 cheques would be payed from bad accounts every hour
- 27 800 letters would be lost every hour
- 3 000 000 bad drug prescriptions would get issued every year

Those numbers were taken from sofware engineering class lectures vpodzime attended this semester.

Number of Anaconda bugs is derived from about 2 mil. downloads of F16.

A bit of motivation

If software had **99.9%** reliability:

- 9 703 cheques would be payed from bad accounts every hour
- 27 800 letters would be lost every hour
- 3 000 000 bad drug prescriptions would get issued every year
- 8 605 flights would crash during takeoff every year

Those numbers were taken from sofware engineering class lectures vpodzime attended this semester.

Number of Anaconda bugs is derived from about 2 mil. downloads of F16.

A bit of motivation

If software had **99.9%** reliability:

- 9 703 cheques would be payed from bad accounts every hour
- 27 800 letters would be lost every hour
- 3 000 000 bad drug prescriptions would get issued every year
- 8 605 flights would crash during takeoff every year
- about 2000 bugs would be filled against Anaconda during F16 lifecycle

Those numbers were taken from sofware engineering class lectures vpodzime attended this semester.

Number of Anaconda bugs is derived from about 2 mil. downloads of F16.

Cyclomatic complexity

As little sidenote: Cyclomatic complexity by Thomas J. McCabe, Sr. (1976).

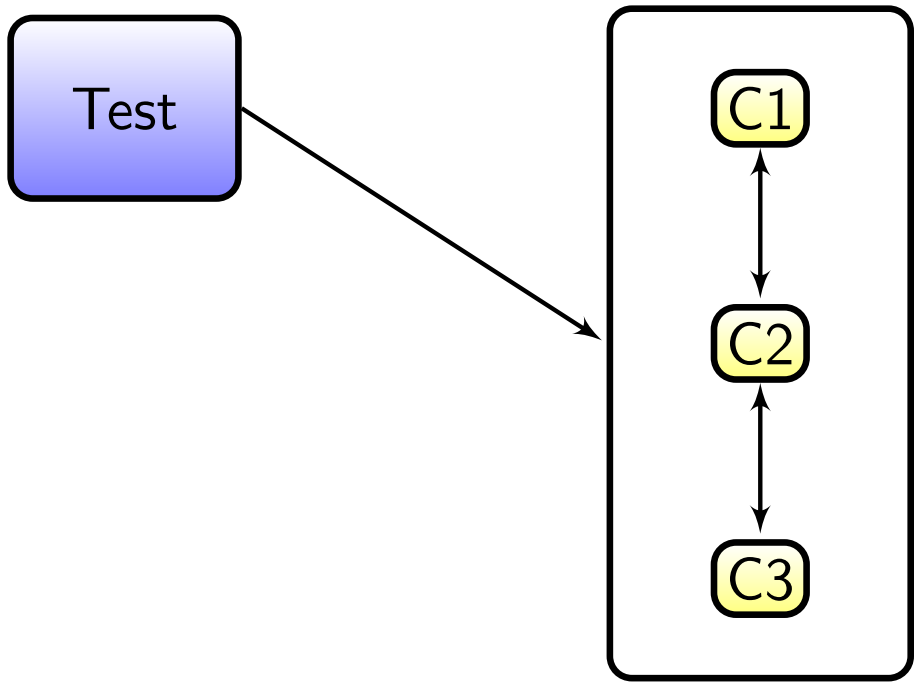
- Human brain has limits
- the maximum it can track is about 7 independent code paths
- Cyclomatic complexity describes the number of lineary independent code paths in method

Use Pygenie to analyze python code.

If you want 100% code path coverage, you have to test all paths in every unit and that takes time.

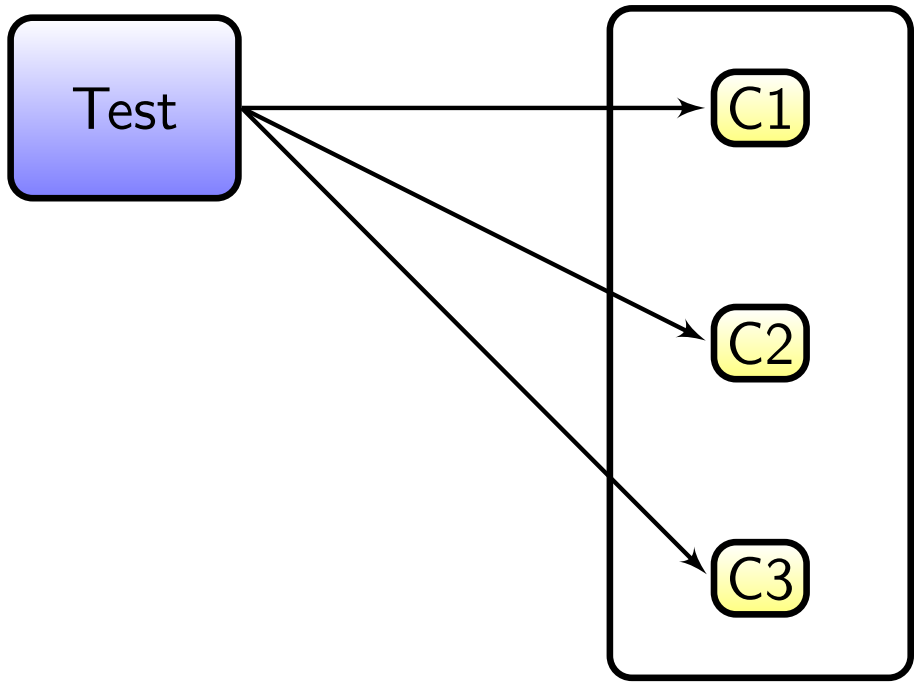
Testing levels

- Integration testing – test everything together as a black box to see if it works together and with other projects.



Testing levels

- **Unit testing** – **test** each component **isolated** from the rest to identify the exact place (and situation) where it breaks. This **requires** that some **seams** are prepared to allow passing testing data and interfaces into the unit.



Testing levels

Integration testing vs. **Unit testing**

Those two cannot be intermixed and serve different purpose! Always think about how to write isolated unit test while writing new code.

Table of contents

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

Initialization side effects

This problem can take many forms in classes and modules:

```
state = None

class Singleton(object):
  def __init__(self):
    global state
    if not state:
      state = self # taints the environment
```

```
import mymodule
logfile = open("project.log", "w")
```

When unavoidable (or unmodifiable) side-effects like this are present, the test won't have clean environment or there won't be enough seams that will allow true unit isolation.

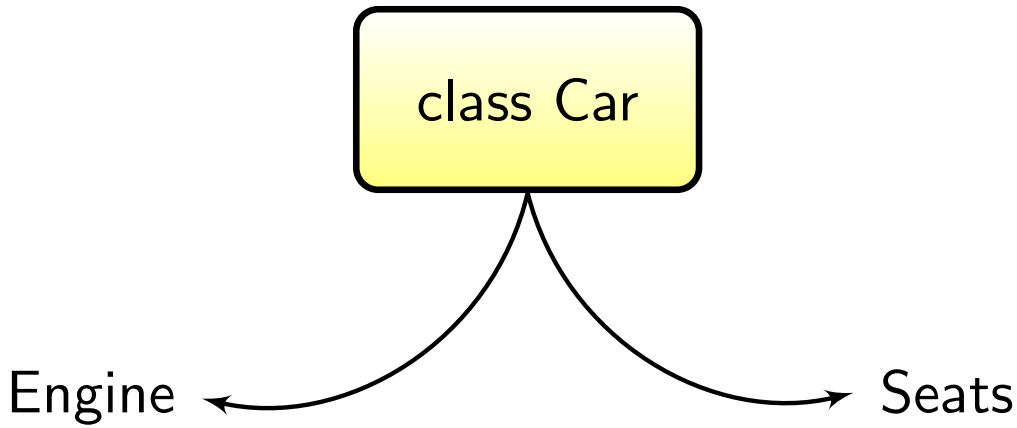
Looking for things

```
class Engine(object):
  def rpm(self, value = 1000):
    pass

class Car(object):
  def __init__(self):
    self.engine = Engine()

  def start(self):
    self.engine.rpm(1500)
```

- ✗ Car is creating it's own engine, that is a bit strange..
- ✗ How do you replace the engine? API?
- ✗ If it cannot be replaced, how do you test the car frame?
- ✗ User of this class has no way of discovering the dependency on Engine

Looking for things - visually

For now, just remember this image, we will see how it can be transformed to much better hierarchy a bit later.



Make intermixed with Use

What this causes is not too different from the previous case, but it is nearly impossible to unit test properly:

```
class Car(object):
  def __init__(self, market):
    if market == "green":
      self.engine = BioEngine()
    else:
      self.engine = DieselEngine()
    self.maximum_speed = self.engine.max_rpm
                         * TRANSMISSION
```

- Isolation means that we need to replace self.engine with our mock instance..
- ✗ Much harder to monkey patch, because you suddenly need to replicate logic in you test.
- Still doable in Python, but getting nasty fast

Table of contents

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

Monkey patching

Test setUp method replaces all references to dependencies to achieve isolation. Simplest method at first, but gets ugly fast.

```
car = Car()

# replace existing engine with dummy
dummy = MockEngine()
car.engine = dummy

car.start()
assert dummy.rpm == 1500
```

- ✓ easy for simple cases
- ✗ mocking modules – sys.modules magic

Method overriding

- ✓ Separate creation and logic
- ✗ Using inheritance instead of better composition
- Not perfect, but has it's uses (SocketServer, ThreadingMixIn)

```
class Car(object):
  def getEngine(self):
    """get engine""" + 1 # notice the syntax error
    return Engine()
```

Test can then replace engine by overriding the method:

```
class TestableCar(Car):
  def getEngine(self):
    return Dummy()
```

Test passes OK, but the actual code fails..

Service locator

```
component_store = {}
component_store["engine"] = BioEngine()
component_store["wings"] = Boeing747Wings()

class Car(object):
  def __init__(self, store):
    self.engine = store.get("engine")

car = Car(component_store)
```

- ✓ One argument passes everything needed
- ✗ Unclear API hides dependencies
- ✗ Mocks get enhanced until the test passes, but by that time the setup routine is more complicated than the code under test
- Considered as an Antipattern in modern testing lore

Table of contents

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

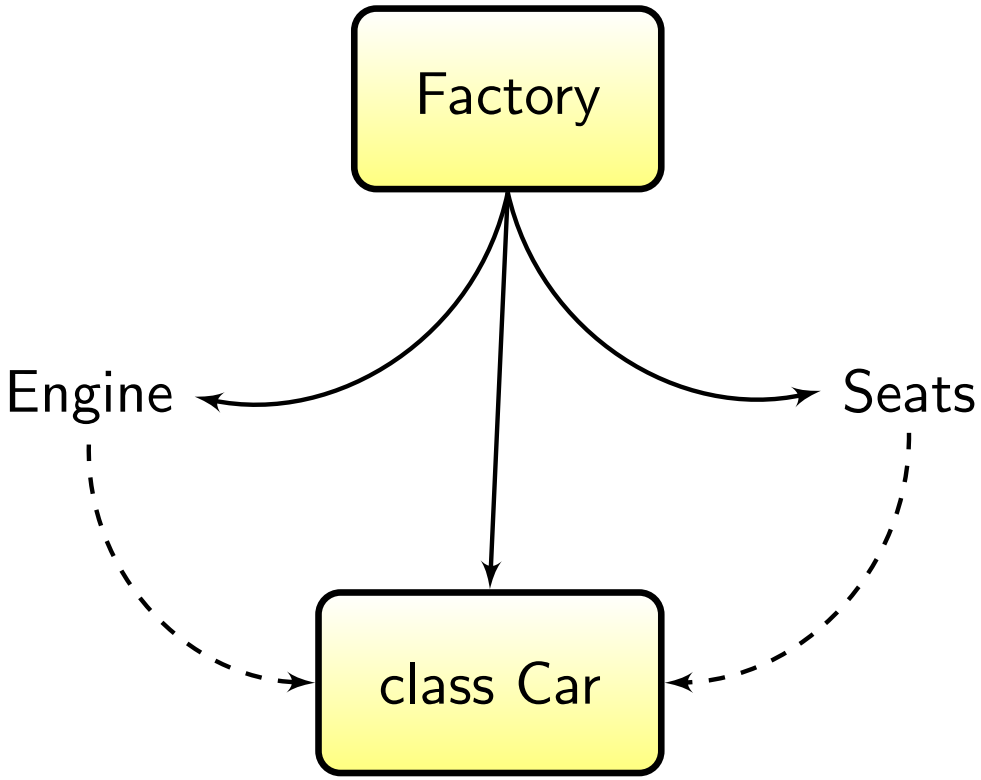
Good patterns

"Nothing that makes a test easy is wasted...We often add methods to classes simply to make the classes easier to test."
– (Robert C. Martin. Software Development, Vol. 10, No. 12, pg 50)

Dependency Injection

Dependency injection is a design style in which every class receives **all** it's **real dependencies** via API.

Factory Pattern



Inversion of Control

Inversion of Control is a concept which combines Dependency Injection and Factory pattern.

Where is the inversion?

The object is not creating it's peers, but somebody (Factory) creates (and initializes) them for him and passes them. (Dependency Injection)

Type 3 IoC – Constructor injection

Dependencies are passed using constructor argument.

```
class Car(object):
  def __init__(self, engine):
    self.engine = engine
```

- ✓ The object is always fully initialized
- ✓ Default arguments can make instantiation easy (only changes need to be specified)
- ✗ Constructor can have lots of arguments

Type 2 IoC – Setter injection

Dependencies are inserted using setters after the object was created.

```
class Car(object):
  def __init__(self):
    pass

  def setEngine(self, engine):
    self.engine = engine
```

- ✓ Less arguments in constructor
- Called only as needed (but think about where you set the defaults)
- ✗ Possibility of ending up with not fully usable object

Type 1 IoC – Interface injection

Not really applicable for Python.

It is a form of setter injection that uses interfaces to define dependencies and API.

Used in Java world, where the interfaces and connection code can be generated and properly inserted by IDE + compiler.

How to create objects inside?

What if we really need to create objects inside?

```
class Receipt(object):
  def __init__(self, shop, amount):
    pass

class Register(object):
  def payment(self, goods):
    return Receipt(self.address, goods.value())
```

How to create objects inside? – 2

Pass a **Factory**! You can even use **default arguments**.

```
class Register(object):
  def __init__(self, receiptFactory = Receipt):
    self.receiptFactory = receiptFactory

  def payment(self, goods):
    return self.receiptFactory(self.address,
                               goods.value())
```

Problematic areas

If we use something with **side effects** in our code, we have to be extra careful. Imagine having delay of 10 seconds hardcoded into code under test and 10 000 tests to run...

1. filesystem, sockets
2. threads, wait loops, delays
3. GUI

Most of the code using these can be made unit testable by using **factories** and **default arguments**:

```
def wait(until, t = time.time, s = time.sleep,
                f = open):
  while t() < until:
    s(1)
  f("data.txt", "w").write("Test!")
```

Table of contents

1. Introduction
2. Common antipatterns
3. Questionable patterns
4. Good patterns
5. Summary

Further reading

- The article that started it all by Martin Fowler
- Dependency injection @ Nette
- Principles of Test Automation
- Guide: Writing Testable Code by Misko Hevery (Google)
- Mock and Testing library by Michael Foord

Summary

- All classes and functions should publicly declare their dependencies
- Ask for a dependency only if you really need it directly
- Always think about the test while writing code (or write test first)

Questions?

Thank you for listening.

Questions?

You can contact me at msivak@redhat.com.