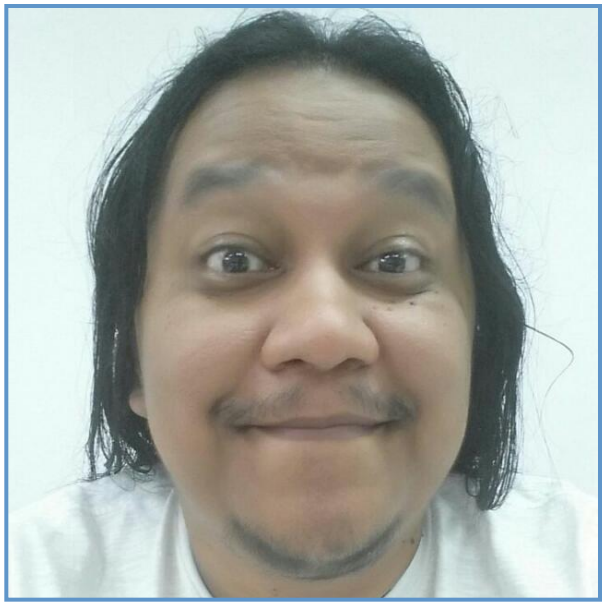
Capítulo 06

Classes e métodos abstratos e Herança múltipla

Márcio Palheta, M.Sc.
marcio.palheta@gmail.com

Apresentação

- Programador desde 2000
- Professor de programação desde 2009
- Mestre pelo ICOMP/UFAM – 2013
- Fundador da Buritech – 2014
- Doutorando pelo ICOMP/UFAM
- Pesquisador das áreas: Banco de Dados, Recuperação da Informação, Big Data, Mineração de dados e Aprendizado de Máquina



Agenda

- Revisão da aula anterior
- Herança entre classes
- Sobrescrita de métodos
- Polimorfismo pythônico
- Decorator @classmethod
- Herança múltipla

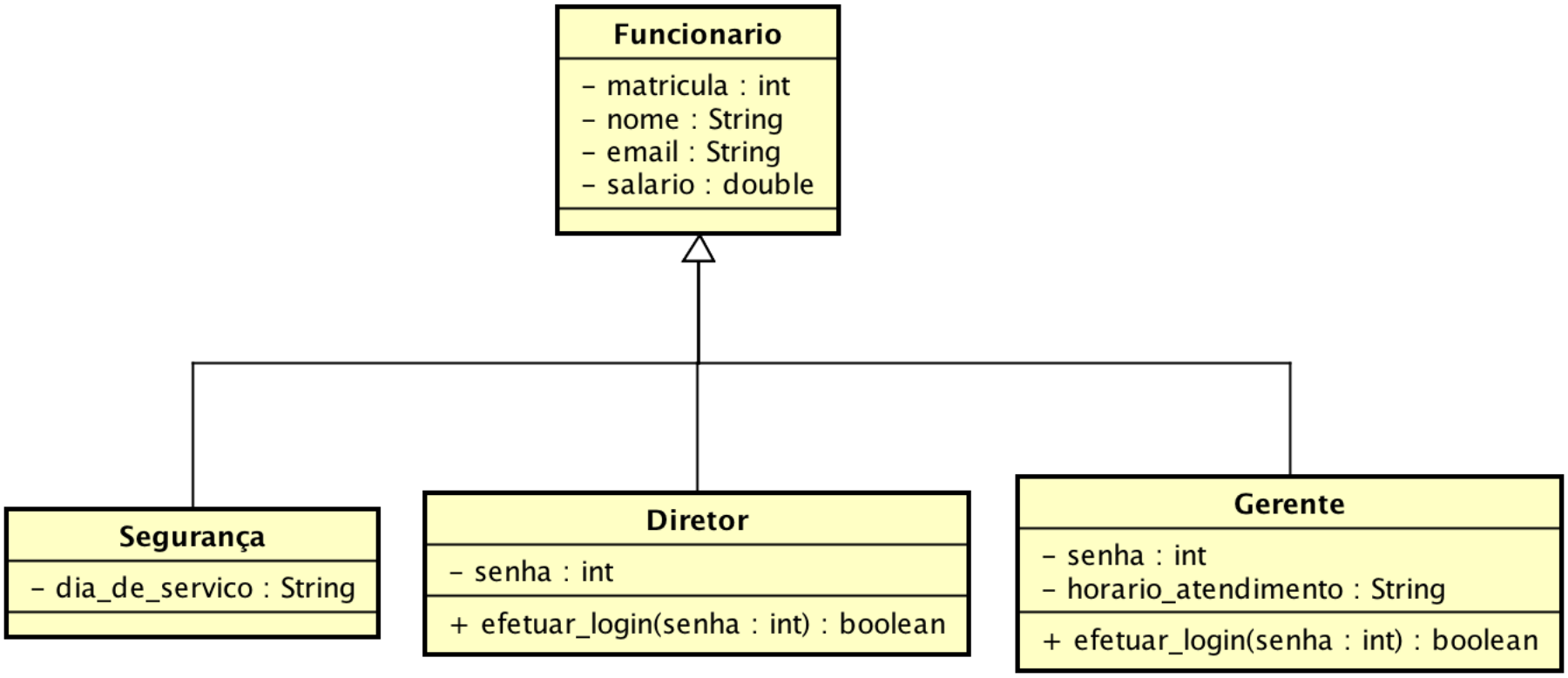
Revisão

- Encapsulamento
- Atributos privados
- Decorators @property e @atrib.setter
- Old style versus new style class
- Atributos e métodos estáticos
- Decorator @staticmethod

O que temos por aqui?

- O que é e como devemos usar a Herança?
- O papel da classe object
- Sobrescrita de métodos
- Como implementar a Herança múltipla ?
- Qual a ordem de execução de métodos na herança múltipla?

O que temos até aqui?



Pensando na modelagem

- Implementamos o diagrama de classes, onde definimos: Funcionario, Seguranca, Diretor e Gerente
- Se todo funcionário da instituição tiver papel bem definido, não faz sentido criar uma instância de funcionário:
  - funcionario = Funcionario()

Classes Abstratas

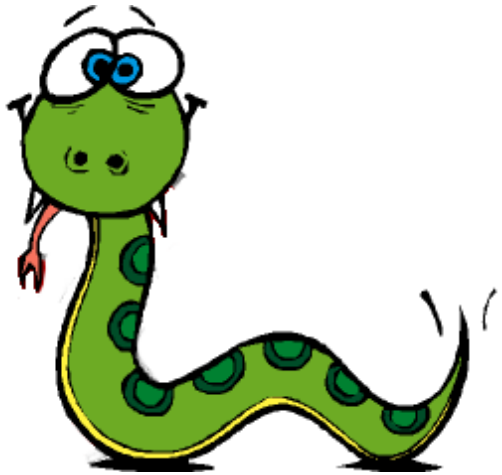
- Em Orientação a Objetos, existem as Classes Abstratas, que são classes que servem de modelo para outras classes
- Não podem ser instanciadas
- Podem conter atributos e métodos
- Seus métodos e propriedades podem ser abstratos(sem implementação)

Classes Abstratas

- Em Python (e linguagem como C++) usam classes abstratas para definição de bases comuns e formalizar interfaces
- Surgiu no Python 2.6. O Python passou 20 anos fazendo sucesso, sem elas.
- Por ser um recurso novo, poucos projetos usam formalmente

Definição de Classes Abstratas

```python
# Python 3.4+
from abc import ABC, abstractmethod
class Abstract(ABC):
    @abstractmethod
    def foo(self):
        pass

# Python 3.0+
from abc import ABCMeta, abstractmethod
class Abstract(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

# Python 2
from abc import ABCMeta, abstractmethod
class Abstract:
    __metaclass__ = ABCMeta

    @abstractmethod
    def foo(self):
        pass
```



Definição de Classes Abstratas

```python
# Python 3.4+
from abc import ABC, abstractmethod
class Abstract(ABC):
    @abstractmethod
    def foo(self):
        pass
```

```python
# Python 3.0+
from abc import ABCMeta, abstractmethod
class Abstract(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass
```

```python
# Python 2
from abc import ABCMeta, abstractmethod
class Abstract:
    __metaclass__ = ABCMeta

    @abstractmethod
    def foo(self):
        pass
```

Em Python 3.4+

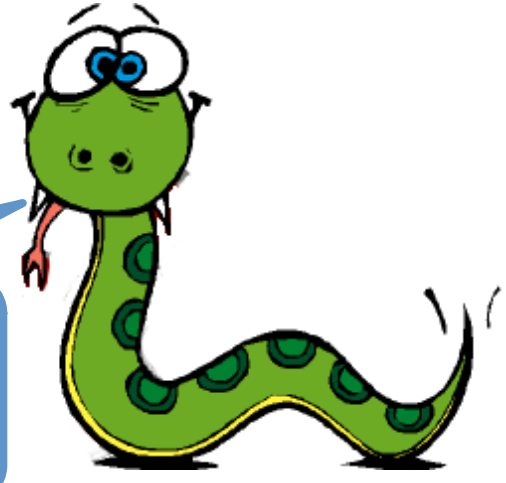
Definição de Classes Abstratas

```python
# Python 3.4+
from abc import ABC, abstractmethod
class Abstract(ABC):
    @abstractmethod
    def foo(self):
        pass
```

```python
# Python 3.0+
from abc import ABCMeta, abstractmethod
class Abstract(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass
```

```python
# Python 2
from abc import ABCMeta, abstractmethod
class Abstract:
    __metaclass__ = ABC

    @abstractmethod
    def foo(self):
        pass
```

Em Python 3.0+



Definição de Classes Abstratas

```python
# Python 3.4+
from abc import ABC, abstractmethod
class Abstract(ABC):
    @abstractmethod
    def foo(self):
        pass

# Python 3.0+
from abc import ABCMeta, ab
class Abstract(metaclass=AB
    @abstractmethod
    def foo(self):
        pass

# Python 2
from abc import ABCMeta, abstractmethod
class Abstract:
    __metaclass__ = ABCMeta

    @abstractmethod
    def foo(self):
        pass
```

Em Python 2

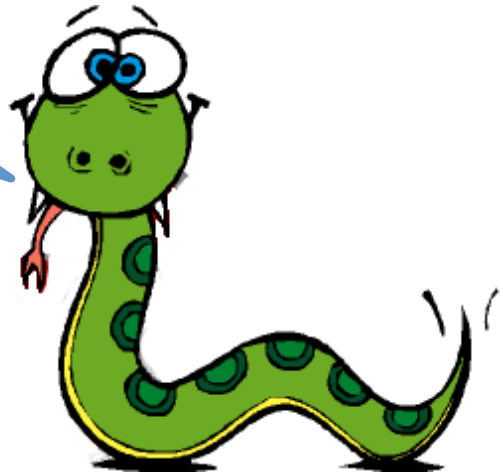
Exercício 1: Funcionário Abstrato

```
1   # -*- coding: UTF-8 -*-
2   # model.py
3
4   from abc import ABCMeta
5
6   class Funcionario(object):
7       __metaclass__ = ABCMeta
8       __contador = 0
9
10      def __init__(self, nome=None, email=N
11          # incrementa o contador
12          Funcionario.__contador += 1
13          # matricula auto-incremento
14          self.__matricula = Funcionario.__
15          self.nome = nome
16          self.email = email
17          self.salario = salario
18
```

Exercício 1: Funcionário Abstrato

```
1   # -*- coding: UTF-8 -*-
2   # model.py
3
4   from abc import ABCMeta
5
6   class Funcionari
7       __metaclass_
8       __contador =
9
10      def __init__
11          # incrementa o contador
12          Funcionario.__contador += 1
13          # matricula auto-incremento
14          self.__matricula = Funcionario.__
15          self.nome = nome
16          self.email = email
17          self.salario = salario
18
```

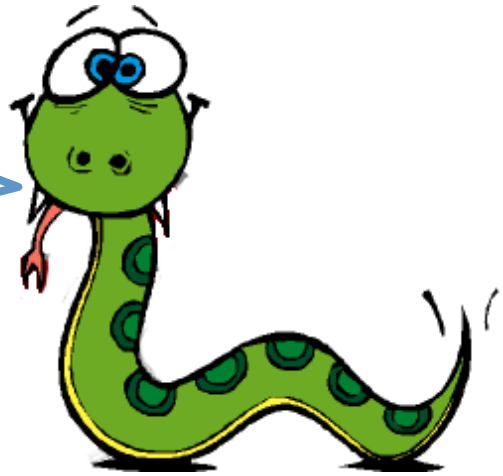
Importe a classe
ABCMeta



Exercício 1: Funcionário Abstrato

```
1   # -*- coding: UTF-8 -*-
2   # model.py
3
4   from abc import ABCMeta
5
6   class Funcionario(object):
7       __metaclass__ = ABCMeta
8       __contador = 0
9
10      def __ini                                    =N
11          # inc
12          Funci
13          # mat
14          self.__
15          self.nome = nome
16          self.email = email
17          self.salario = salario
18
```

Marque a classe como abstrata



Execute o teste do controlador

```
1   # -*- coding: UTF-8 -*-
2   # teste_controlador_bonus.py
3
4   from model import Funcionario
5   from model import Seguranca
6   from model import Diretor
7   from model import Gerente
8   from control import ControleBonus
9
10  controlador = ControleBonus()
11  controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

Execute o teste do controlador

```
1   # -*- coding: UTF-8 -*-
2
3
4
5
6   from model import Diretor
7   from model import Gerente
8   from control import ControleBo
9
10  controlador = ControleBonus()
11  controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

Tentando instanciar uma classe abstrata

Execute o teste do controlador

```
1   # -*- coding: UTF-8 -*-
2
3
4
5
6   from model import Diretor
7   from
8   from
9
10  controlador = ControleBonus()
11  controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

Tentando instanciar uma classe abstrata

Em Java, isso daria erro

Resultado no console

Run teste_controlador_bonus

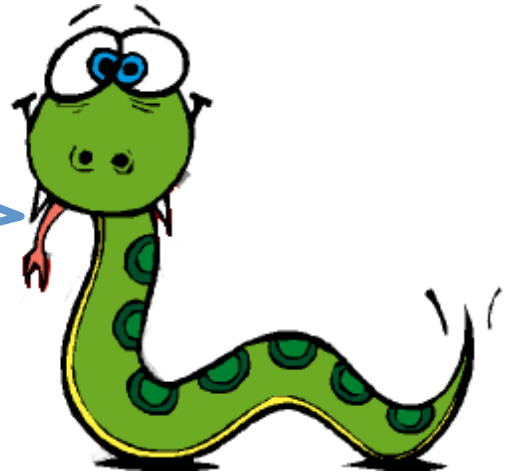
```
/Library/Frameworks/Python.framework/
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1500.0
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1200.0
Bônus total: 5100.0
Bônus soma:  5100.0
Após a exclusão do segurança
Bônus total: 3900.0
Bônus soma:  3900.0
```

Resultado no console

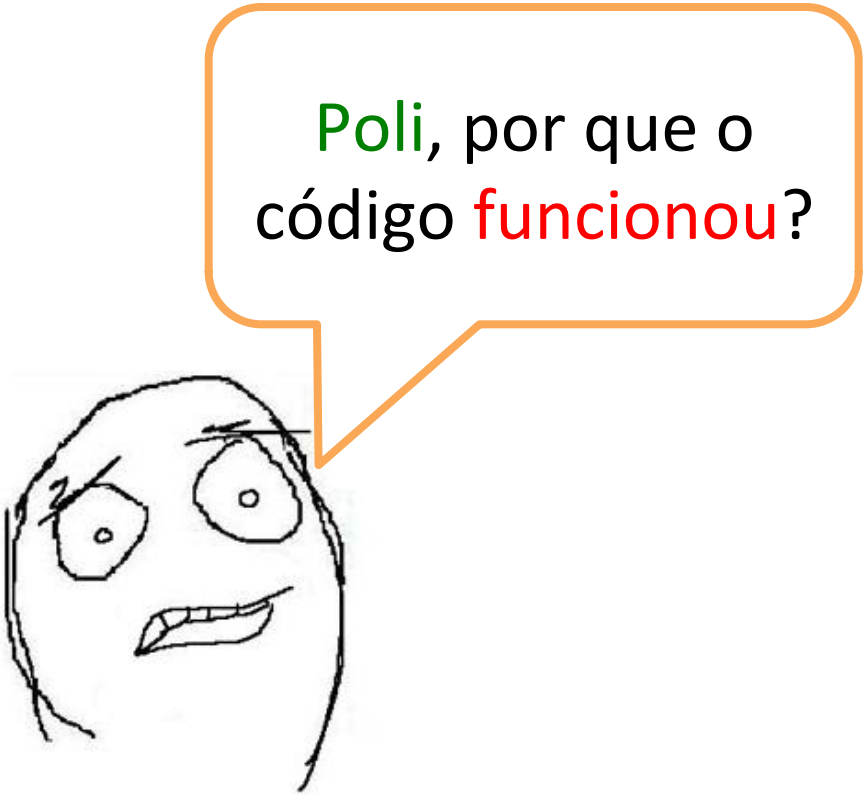
Run teste_controlador_bonus

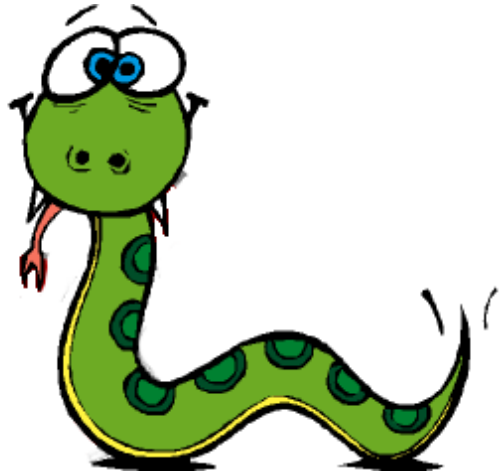
```
/Library/Frameworks/Python.framework/
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1500.0
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1200.0
Bônus
Bônus
Após a
Bônus total: 3900.0
Bônus soma:  3900.0
```

Mas em Python,
deu tudo certo



Pensando em classes abstratas

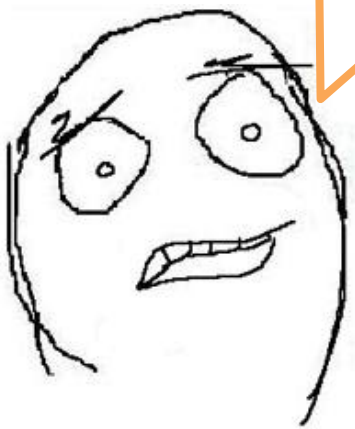




Pensando em clas

Morfismo, funcionou porque não definimos nenhum método ou propriedade abstrata

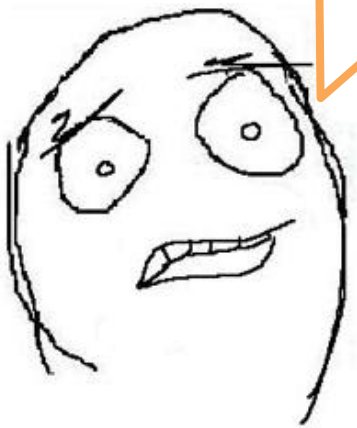
Poli, por que o código funcionou?

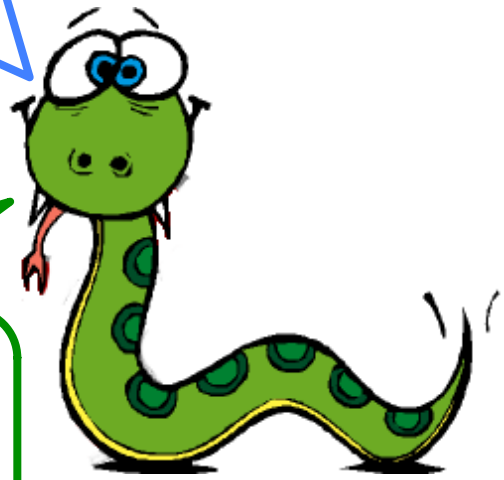


Pensando em clas

Morfismo, funcionou porque não definimos nenhum método ou propriedade abstrata

Poli, por que o código funcionou?





Classes sem componentes abstratos podem ser instanciadas normalmente

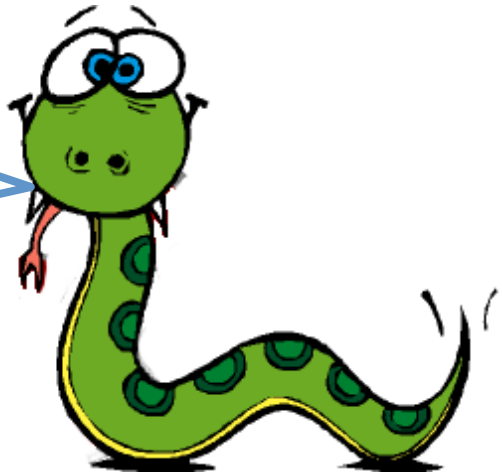
Exercício 2: bônus abstrato

```
from abc import ABCMeta
from abc import abstractproperty

class Funcionario(object):
    __metaclass__ = ABCMeta
    __contador = 0

    def __init__(self, nome=None, email=None, salario=0.0):
        Funcionario.__contador += 1
        self.__matricula = Funcionario.__contador
        self.nome = nome
        self.email = email
        self.salario = salario

    @abstractproperty
    def bonus(self):
        pass
```

Exercício 2: bônus abstrato

```
from abc import ABCMeta
from abc import abstractproperty

class Funcio
    __metacl
    __contac

    def __in
        Funcionario.__contador += 1
        self.__matricula = Funcionario.__
        self.nome = nome
        self.email = email
        self.salario = salario

    @abstractproperty
    def bonus(self):
        pass
```

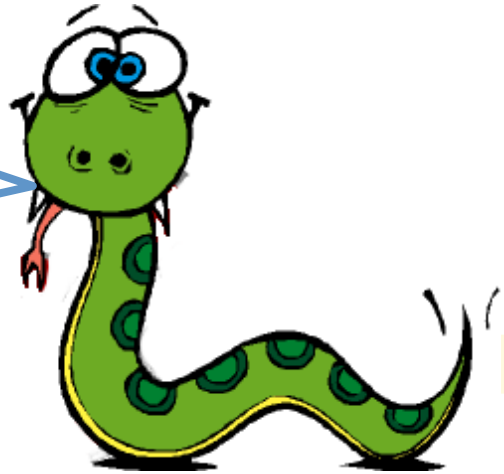
No script model.py, atualize os imports



Exercício 2: bônus abstrato

```
from abc import ABCMeta
from abc import abstractproperty

class Funcionario(object):
    __metaclass__ = ABCMeta
    __contador = 0

    def __init__(self, nome=None, email=None, salario=0.0):
        Func
        self
        self
        self
        self

    @abstractproperty
    def bonus(self):
        pass
```

Torne a propriedade bônus abstrata



Execute o teste novamente

- Agora, ocorre um TypeError, informando que não pode instanciar Funcionário, porque possui o método abstrato bonus

```
10  controlador = ControleBonus()
11  controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

```
Run teste_controlador_bonus
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/marciopalheta/p
Traceback (most recent call last):
  File "/Users/marciopalheta/python/codigofonte/financeiro/teste_controlador_bonus.py"
    controlador.lista = Funcionario(salario=1000)
TypeError: Can't instantiate abstract class Funcionario with abstract methods bonus

Process finished with exit code 1
```

Execute o teste novamente

- Agora, ocorre um TypeError, informando que não pode instanciar Funcionário, porque possui o método abstrato bonus

```
10  controlador = ControleBonus()
11  controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

```
Run teste_controlador_bonus
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/marciopalheta/p
Traceback (most recent call last):
  File "/Users/marciopalheta/python/codigofonte/financeiro/teste_controlador_bonus.py"
    controlador.lista = Funcionario(salario=1000)
TypeError: Can't instantiate abstract class Funcionario with abstract methods bonus

Process finished with exit code 1
```

Exercício 3: teste sem funcionário

- Comente a linha que instancia Funcionário, teste novamente e veja que o erro mudou

```
10  controlador = ControleBonus()
11  #controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

```
Run  teste_controlador_bonus
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/marciopalhe
Traceback (most recent call last):
salario: 1000 bonus: 1500.0
  File "/Users/marciopalheta/python/codigofonte/financeiro/teste_controlador_bonus
    controlador.lista = Gerente(salario=1000)
TypeError: Can't instantiate abstract class Gerente with abstract methods bonus

Process finished with exit code 1
```

Exercício 3: teste sem funcionário

A instância de Gerente não foi criada porque a classe NÃO implementa a propriedade bônus

```
11  #controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

```
Run  teste_controlador_bonus
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/marciopalhe
Traceback (most recent call last):
salario: 1000 bonus: 1500.0
  File "/Users/marciopalheta/python/codigofonte/financeiro/teste_controlador_bonus
    controlador.lista = Gerente(salario=1000)
TypeError: Can't instantiate abstract class Gerente with abstract methods bonus

Process finished with exit code 1
```

Exercício 3: teste sem funcionário

- A instância de Diretor foi criada porque a classe já implementa a propriedade bônus

```
10
11  #controlador.lista = Funcionario(salario=1000)
12  controlador.lista = Diretor(salario=1000)
13  controlador.lista = Gerente(salario=1000)
14  seguranca = Seguranca(salario=1000)
15  controlador.lista = seguranca
16
```

```
Run  teste_controlador_bonus
/Library/Frameworks/Python.framework/Versions/2.7/bin/python2.7 /Users/marciopalhe
Traceback (most recent call last):
salario: 1000 bonus: 1500.0
  File "/Users/marciopalheta/python/codigofonte/financeiro/teste_controlador_bonus
    controlador.lista = Gerente(salario=1000)
TypeError: Can't instantiate abstract class Gerente with abstract methods bonus

Process finished with exit code 1
```

Exercício 4: implementando bônus

```
class Gerente (Funcionario):
    def __init__(self, senha=None, horario_atendimento=None,
                 salario=None):
        super(Gerente, self).__init__(salario=salario)
        self.senha = senha
        self.horario_atendimento = horario_atendimento

    def efetuar_login(self, senha):
        return self.senha == senha

    @property
    def bonus(self):
        return self.salario * 1.2

class Seguranca(Funcionario):
    def __init__(self, dia_servico=None, salario=None):
        Funcionario.__init__(self, salario=salario)
        self.dia_servico = dia_servico

    @property
    def bonus(self):
        return self.salario * 1.2
```

Exercício 4: implementando bônus

Implemente a propriedade de bônus nas classes Gerente e Segurança, assumindo 20%

```
class Gerente (Funcionario):
    ...io_a

    @property
    def bonus(self):
        return self.salario * 1.2

class Seguranca(Funcionario):
    def __init__(self, dia_servico=None, salario=None):
        Funcionario.__init__(self, salario=salario)
        self.dia_servico = dia_servico

    @property
    def bonus(self):
        return self.salario * 1.2
```

Tudo voltou a funcionar ☺

Run teste_controlador_bonus

```
/Library/Frameworks/Python.framework/
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1500.0
salario: 1000 bonus: 1200.0
salario: 1000 bonus: 1200.0
Bônus total: 5100.0
Bônus soma:  5100.0
Após a exclusão do segurança
Bônus total: 3900.0
Bônus soma:  3900.0
```

Além da herança

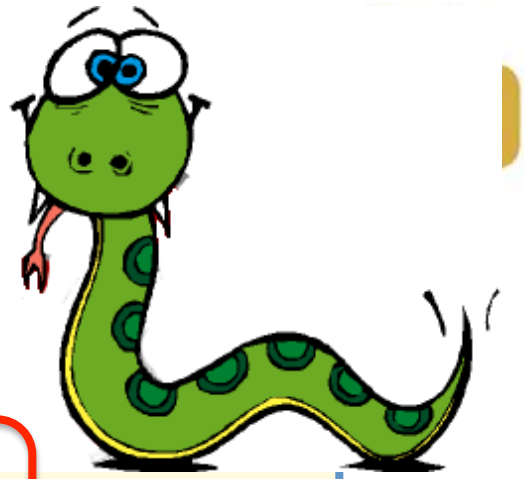
- Na maioria das linguagens OO, o uso de classes abstratas está intimamente ligado ao conceito de herança
- Uma classe pai abstrata e várias classes filhas, concretas ou abstratas.
- Em Python podemos registrar a classe concreta em tempo de execução

Uma classe abstrata

```
from abc import ABCMeta
from abc import abstractmethod

class PluginBase(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def load(self, data):
        pass

    @abstractmethod
    def save(self, output, data):
        pass
```

Ur

Classe abstrata,
filha de object



```
from abc
from abc                                    et

class PluginBase(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def load(self, data):
        pass

    @abstractmethod
    def save(self, output, data):
        pass
```

Uma classe abstrata

```
from abc i
from abc i

class Plug
    __metaclass__ = ABC
```

Métodos abstratos

```
@abstractmethod
def load(self, data):
    pass
```

```
@abstractmethod
def save(self, output, data):
    pass
```

Definição da implementação

```
class RegisteredImplementation(object):
    def __int__(self):
        pass

    def load(self, data):
        return data.read()

    def save(self, output, data):
        return output.write(data)

# Equivalente ao metodo main de Java e C
if __name__ == '__main__':

    # Definicao da implementacao de PluginBase, em tempo de execucao
    PluginBase.register(RegisteredImplementation)

    print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

Definição da implementação

```
class RegisteredImplementation(object):
    def __int__(self):
        pass

    def load(self, data):
        return data.read(

    def save(self, output
        return output.wri

# Equivalente ao metodo m
if __name__ == '__main__':

    # Definicao da implementacao de PluginBase, em tempo de execucao
    PluginBase.register(RegisteredImplementation)

    print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

Classe concreta, filha de object

Defini

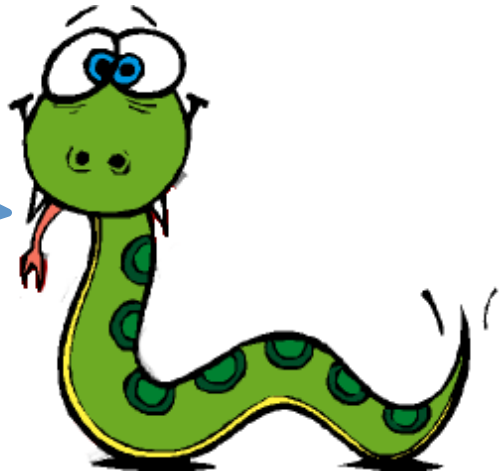
Métodos concretos

```
class RegisteredIm
    def __int__(se
        pass

    def load(self, data):
        return data.read()

    def save(self, output, data):
        return output.write(data)

# Equivalente ao metodo main de Java e C
if __name__ == '__main__':

    # Definicao da implementacao de PluginBase, em tempo de execucao
    PluginBase.register(RegisteredImplementation)

    print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

Definição da implementação

```
class RegisteredImplementation(object):
    def __int__(self):
        pass

    def load(
        retur

    def save(
        retur

# Equivalente ao metodo main de Java e C
if __name__ == '__main__':

    # Definicao da implementacao de PluginBase, em tempo de execucao
    PluginBase.register(RegisteredImplementation)

    print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
    print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

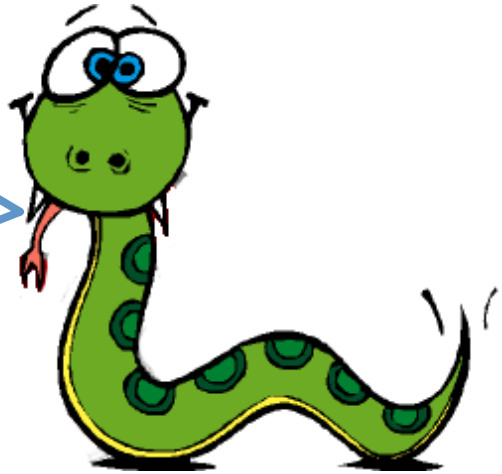
Definição de script de execução



Definição da implementação

```
class RegisteredImplementation(object):
    def __int__(self):
        pass

    def load(self, data):
        pass

    def s
        p

# Equiva
if __name_
```

Definição da classe que implementa a abstrata

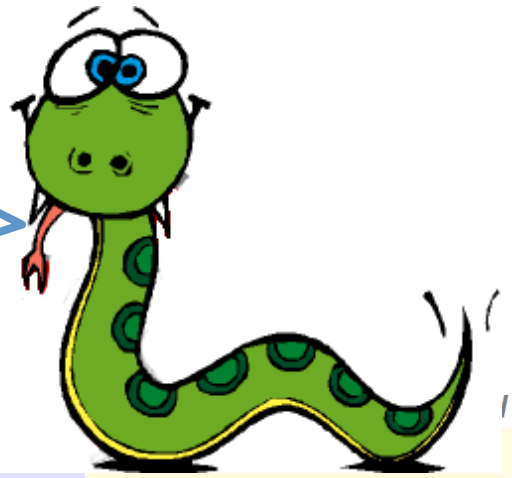


```
# Definicao da implementacao de PluginBase, em tempo de execucao
PluginBase.register(RegisteredImplementation)

print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

Definição da implementação

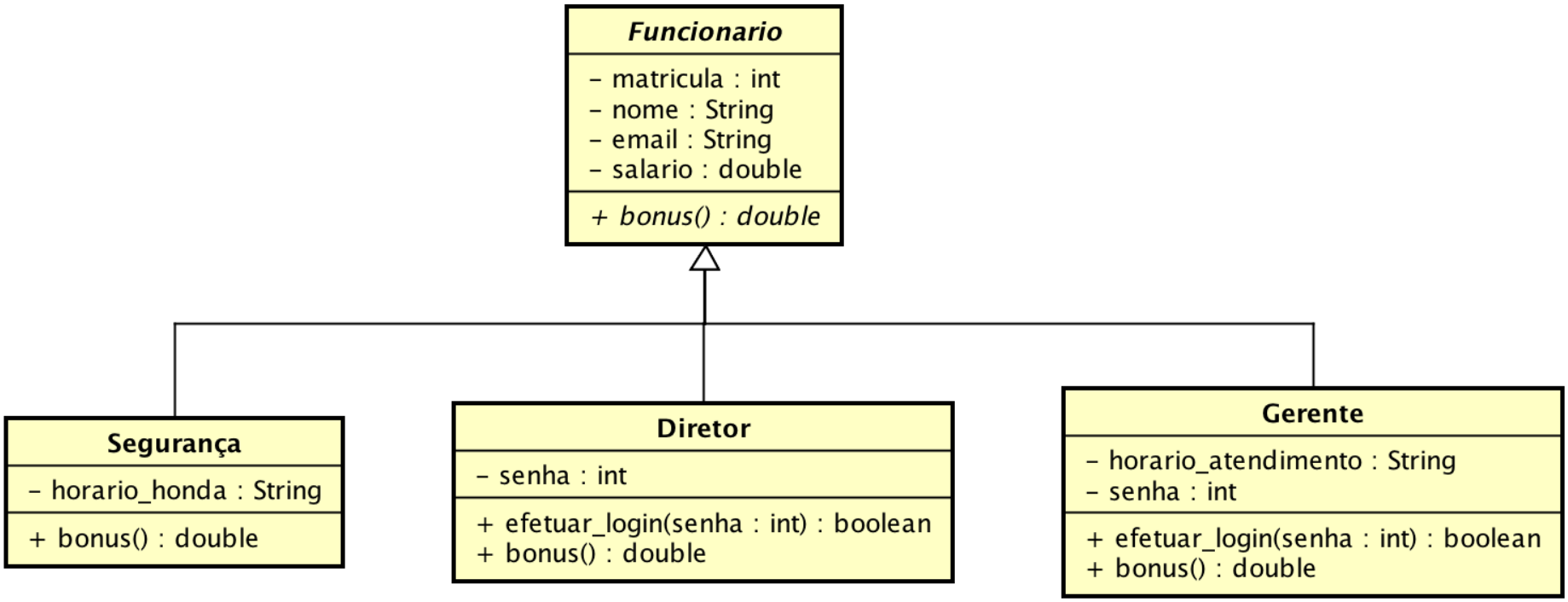
```
class RegisteredImplementation(object):
    def __int__(self):
        pass

    def load(self, data):
        return data.read()

    def save(self, output, data):
        r

# Equiva
if __name

    # De                                    Ba                    icao
    PluginBase.register(RegisteredImplementa

print 'Subclass:', issubclass(RegisteredImplementation, PluginBase)
print 'Instance:', isinstance(RegisteredImplementation(), PluginBase)
```

Verificação de subclasse e instância



Mas a vida...

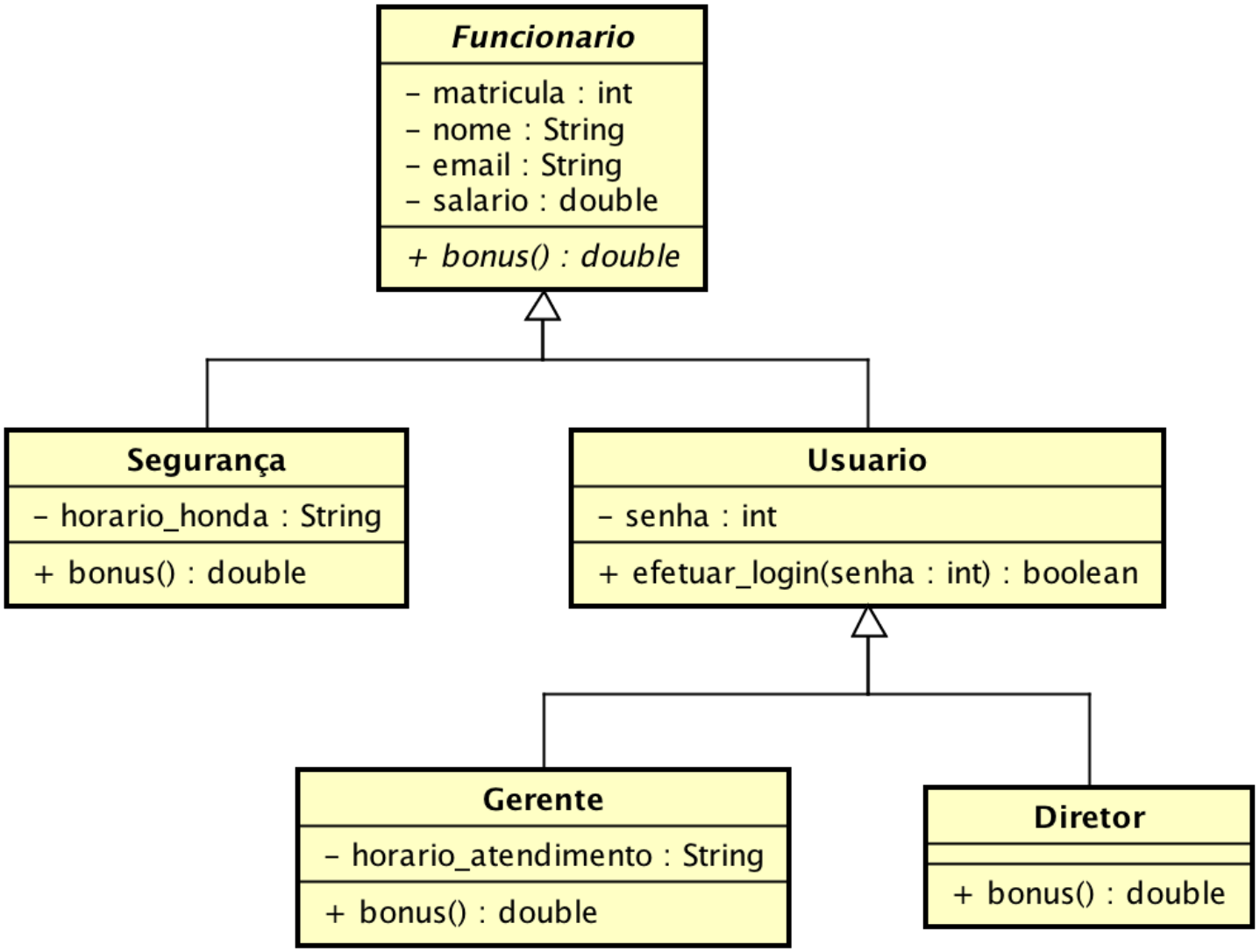
- Agora que estamos avançando no domínio das classes, bora rever isso?



Refatorando o modelo

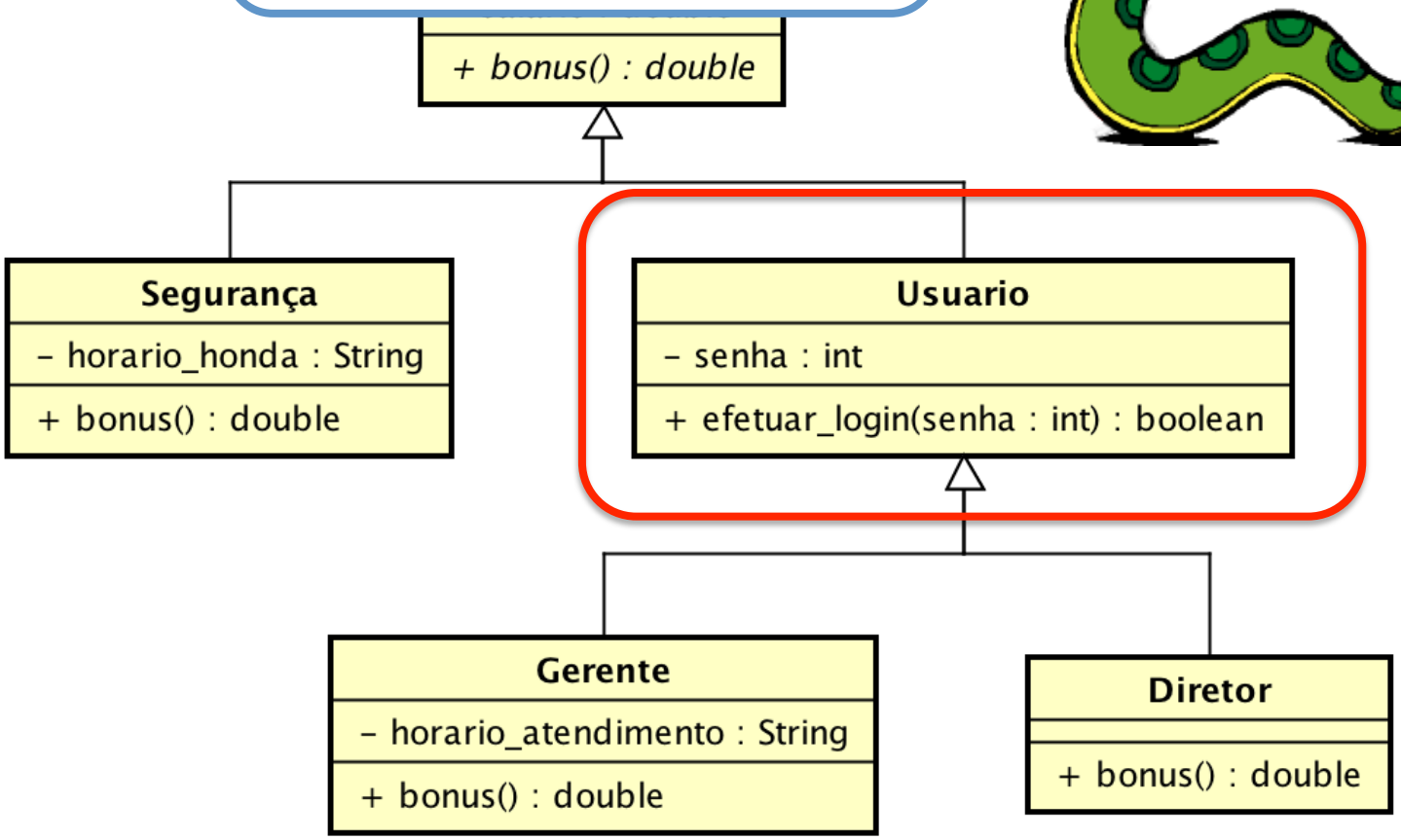
- Gerente e Diretor são classes filhas de Funcionário, assim como Segurança
- Contudo, gerente e Diretor possuem senha e o método autenticar
- E se outras classes fossem criadas e precisassem de autenticação?
- Vamos criar a classe abstrata Usuário

Refatorando a modelagem

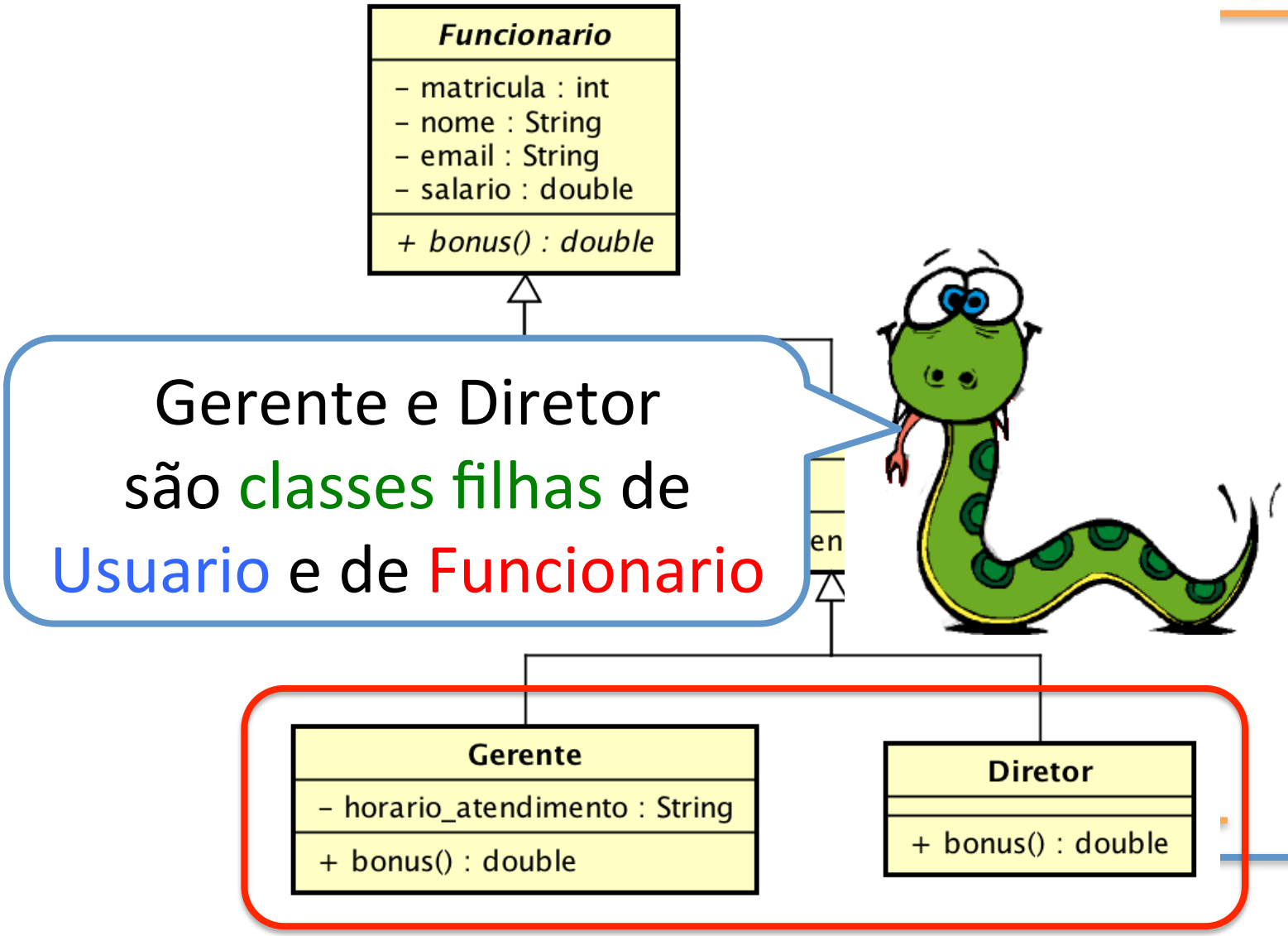


Refatorando a modelagem

Nova classe filha de Funcionario



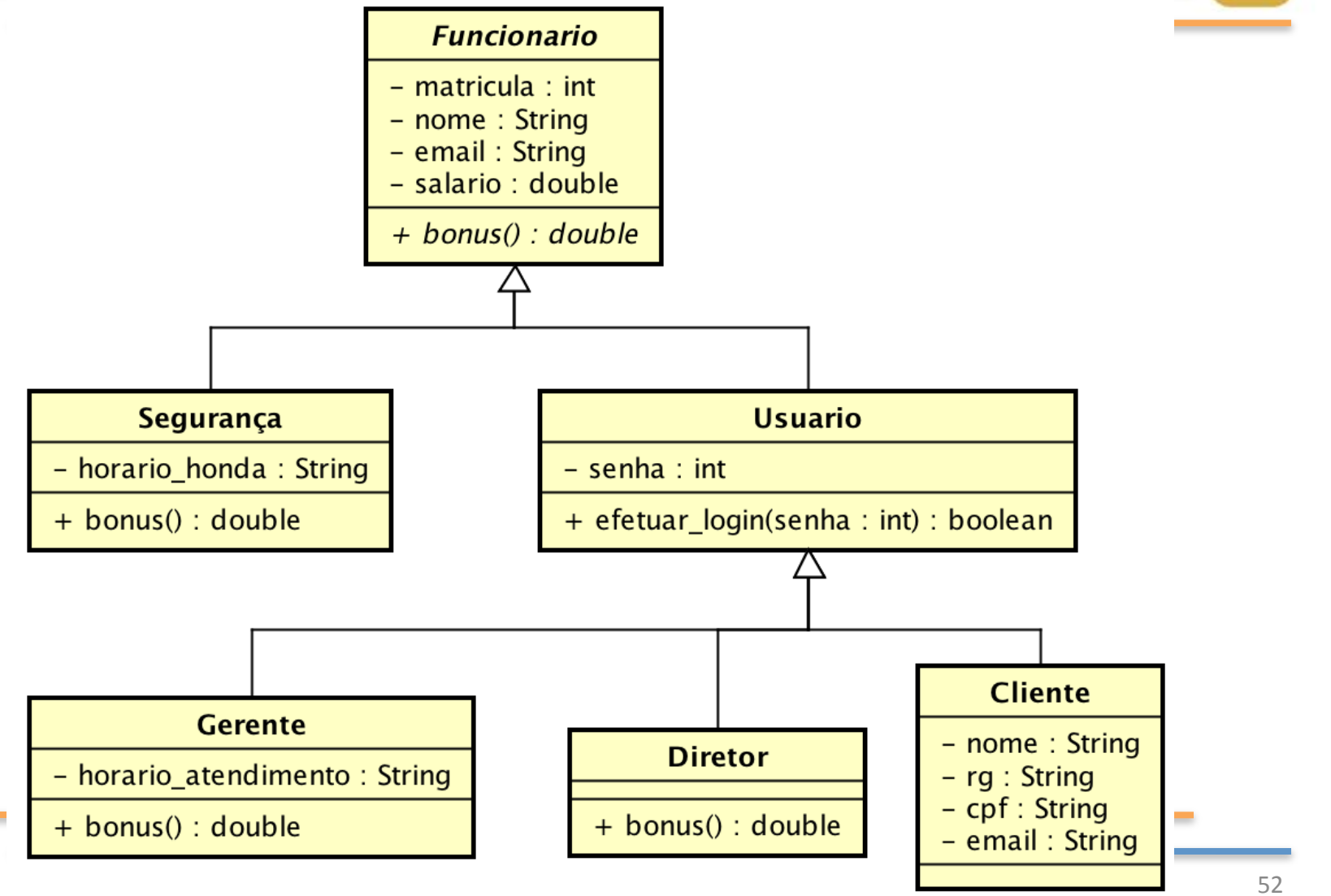
Refatorando a modelagem



Mas a vida...

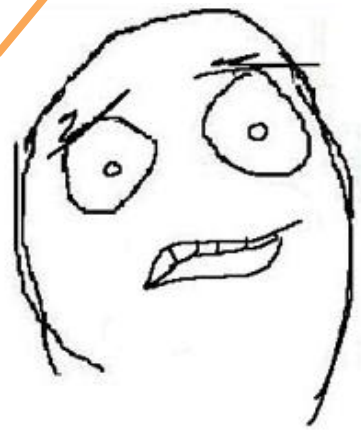
- Com o passar do tempo, o setor de controle de usuários percebeu a necessidade de controle da autenticação de clientes
- Neste novo cenário, cliente passa a representar um usuário do nosso sistema
- Como poderíamos modelar a solução?

E se o cliente fosse usuário?

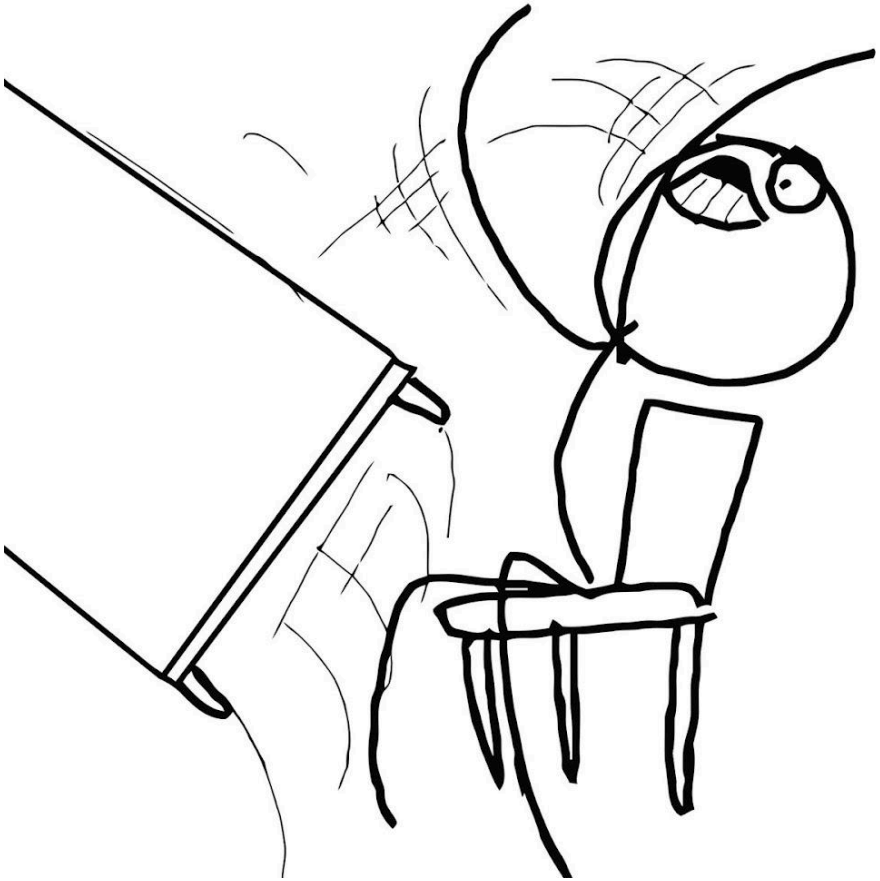


E se o cliente fosse usuário?

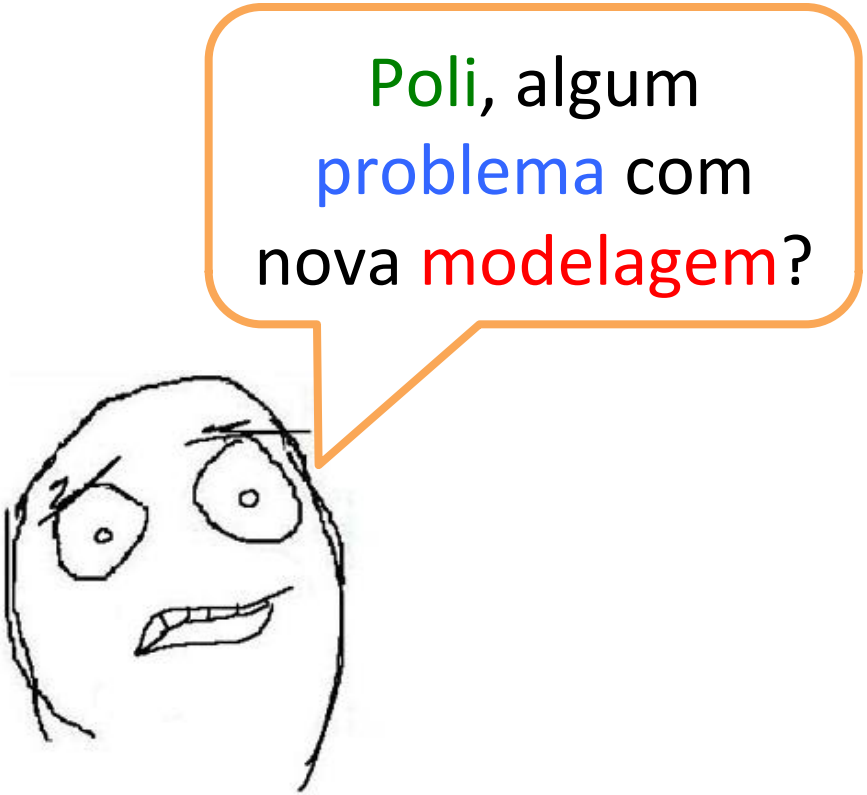
Poli, podemos criar a classe Cliente filha de Usuario

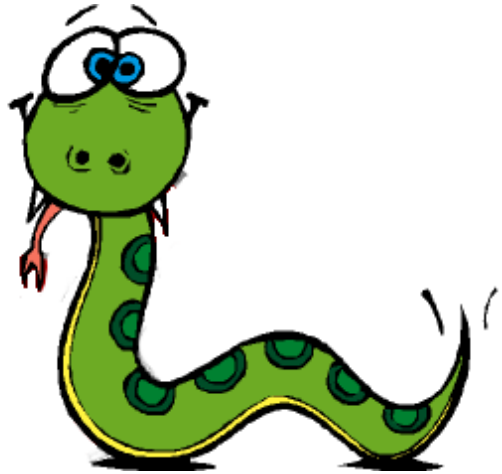


E se o cliente fosse usuário?



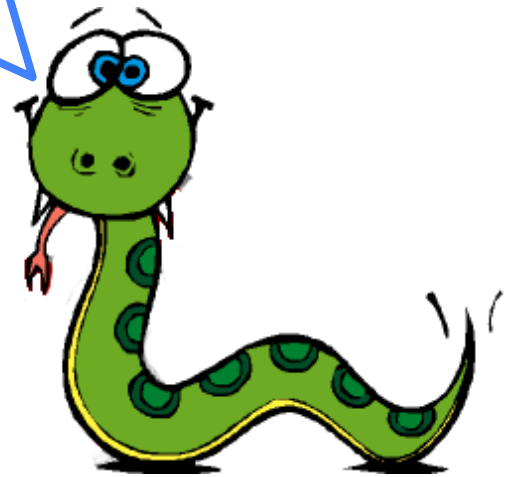
Todo cliente é funcionário?



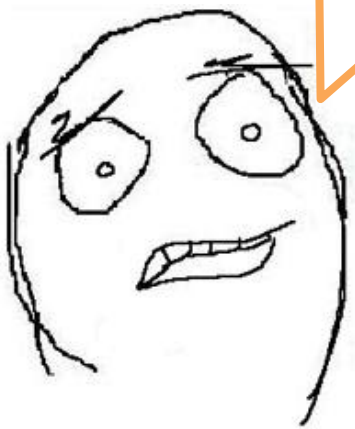


Todo cliente é

Morfismo, pelo diagrama, todo cliente é um funcionário



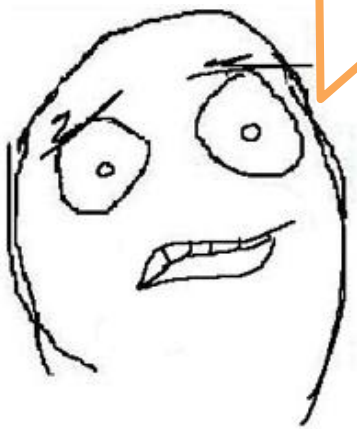
Poli, algum problema com nova modelagem?



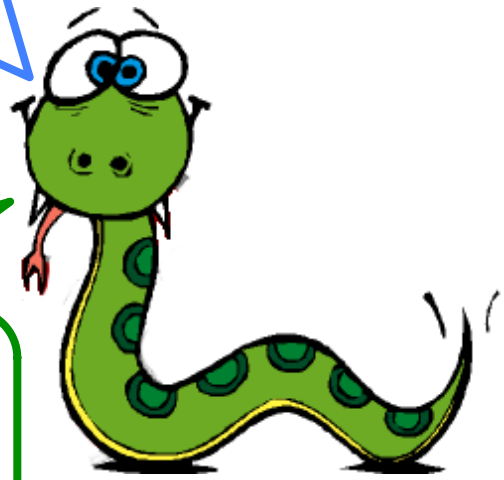
Todo cliente é

Morfismo, pelo diagrama, todo cliente é um funcionário

Poli, algum problema com nova modelagem?

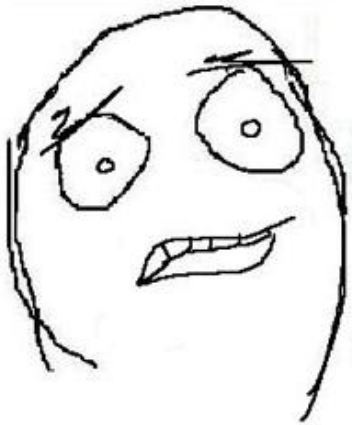


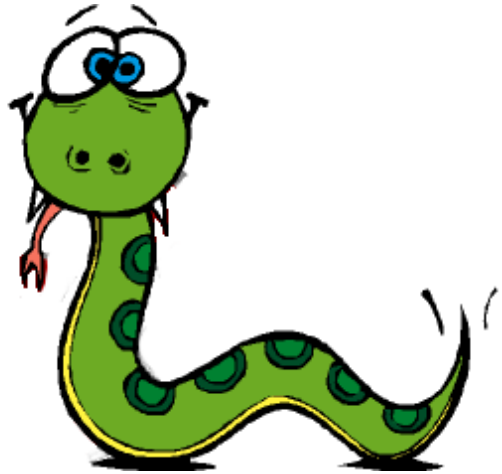
Na prática, acho que isso não é verdade



Todo cliente é funcionário?

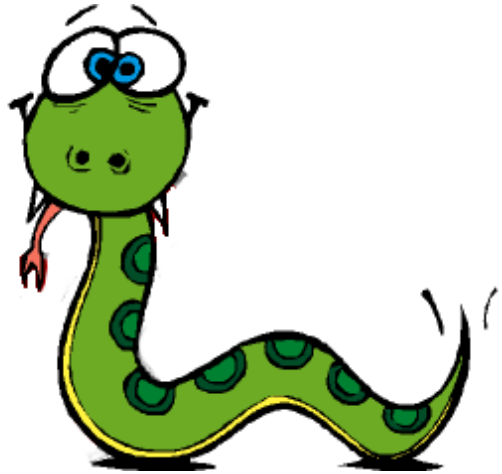
Vou escrever umas classes, pra te ajudar a pensar em uma solução



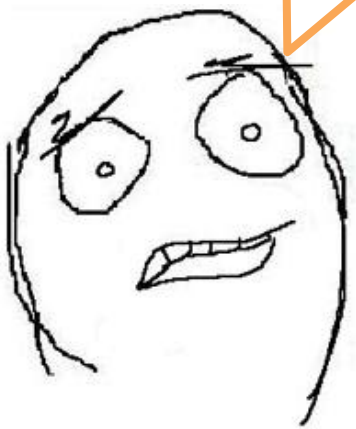


Todo cliente é funcionário?

Vou escrever umas classes, pra te ajudar a pensar em uma solução

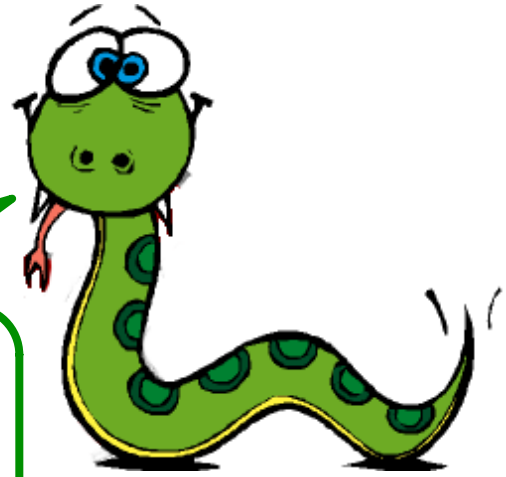


Opa, gostei

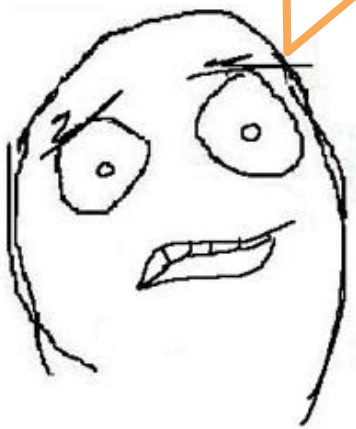


Todo cliente é funcionário?

Vou escrever umas classes, pra te ajudar a pensar em uma solução



Opa, gostei



Imagina a criação das classe SerVivo, Aranha e Homem

Pensando além

```
3  class SerVivo(object):
4      def __init__(self, nome):
5          self.nome = nome
6
7  class Homem(SerVivo):
8      def falar_nome(self):
9          print "Meu nome eh:", self.nome
10
11 class Aranha(SerVivo):
12     def jogar_teia(self):
13         print "Jogando a teia"
14
15 class HomemAranha(Homem, Aranha):
16     pass
17
18 cara = HomemAranha("Peter Parker")
19 cara.falar_nome()
20 cara.jogar_teia()
```

Pensando além

```
3  class SerVivo(object):
4      def __init__(self, nome):
5          self.nome = nome
6
7  class Homem(SerVivo):
8      def falar_nome(self):
9          print "Meu nome eh:", self.nome
10
11 class Aranha(SerVivo):
12     def jogar_teia(self):
13         print "Jogando a teia"
14
15                                  m,
16
17
18                                  te
19
20 cara.jogar_teia()
```

Herança simples

Pensando além

```
3  class SerVivo(object):
4      def __init__(self, nome):
5          self.nome = nome
6
7
8
9                                  me
10
11 class Aranha(SerVivo):
12     def jogar_teia(self
13         print "Jogando
14
15 class HomemAranha(Homem, Aranha):
16     pass
17
18 cara = HomemAranha("Peter Parker")
19 cara.falar_nome()
20 cara.jogar_teia()
```

Herança múltipla

Resultado no console

```
15  class HomemAranha(Homem, Aranha):
16      pass
17
18  cara = HomemAranha("Peter Parker")
19  cara.falar_nome()
20  cara.jogar_teia()
```

Run teste_heranca_multipla

```
/Library/Frameworks/Python.framework/
Meu nome eh: Peter Parker
Jogando a teia

Process finished with exit code 0
```

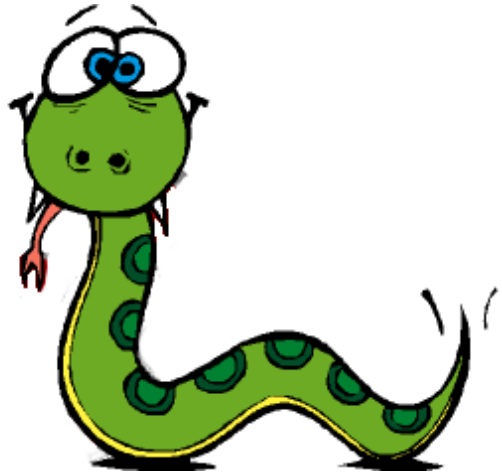
Resultado no console

```
15  class HomemAranha(Homem, Aranha):
16      pass
17
18  cara = HomemAranha("Peter Parker")
19  cara.falar_nome()
20  cara.jogar_teia()
```

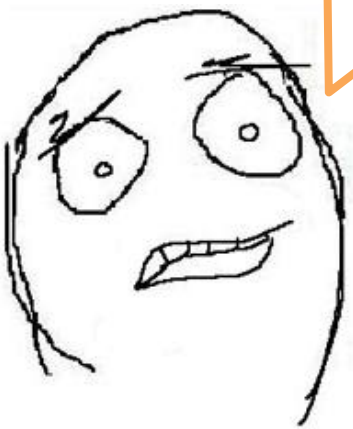
Run teste_heranca_multipla

```
/Library/Frameworks/Python.framework/
Meu nome eh: Peter Parker
Jogando a teia

Process finished with exit code 0
```

Herança múltipla

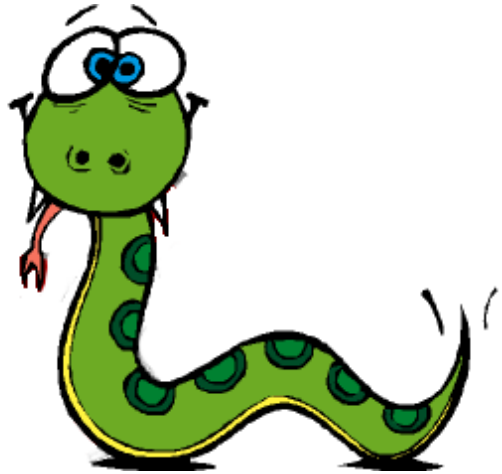


Herança múltipla

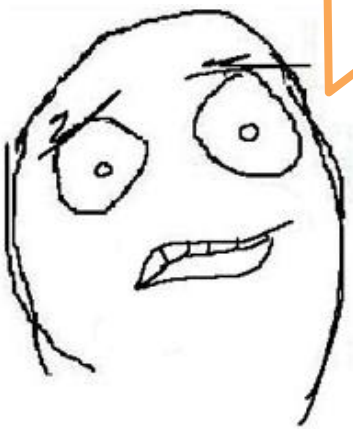


Huuuum.
Poderíamos usar a
herança múltipla

Exatamente

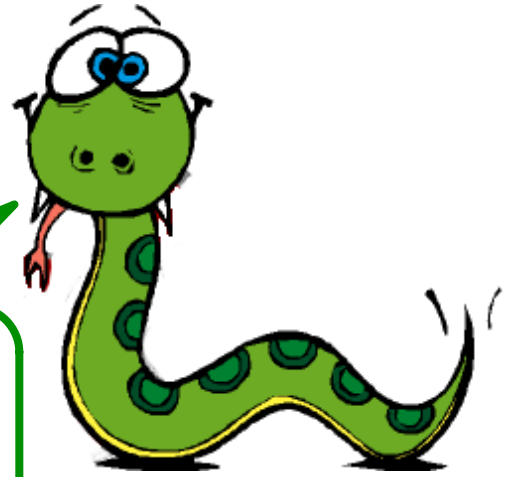


Herança múltipla



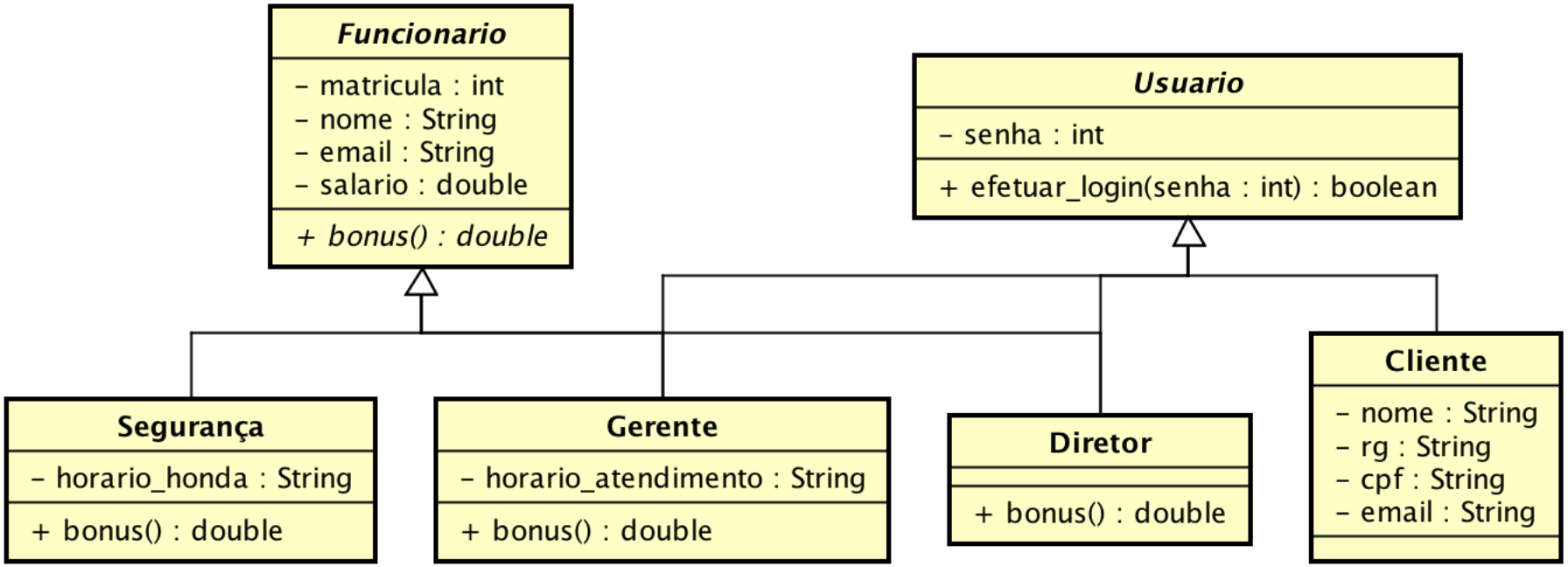
Huuuum.
Poderíamos usar a
herança múltipla

Exatamente



Vamos rever a
modelagem

Novo modelo


Funcionario
- matricula : int
- nome : String
- email : String
- salario : double
+ bonus() : double
Usuario
- senha : int
+ efetuar_login(senha : int) : boolean
Segurança
- horario_honda : String
+ bonus() : double
Gerente
- horario_atendimento : String
+ bonus() : double
Diretor
+ bonus() : double
Cliente
- nome : String
- rg : String
- cpf : String
- email : String

Exercícios

- Vamos atualizar nosso código fonte, para refletir a modelagem atual
- Crie a classe abstrata Usuario, filha de Funcionario
- Torne as classes Gerente e Diretor filhas de Usuario
- Crie a classe Cliente, filha de Usuario

Exercício 5

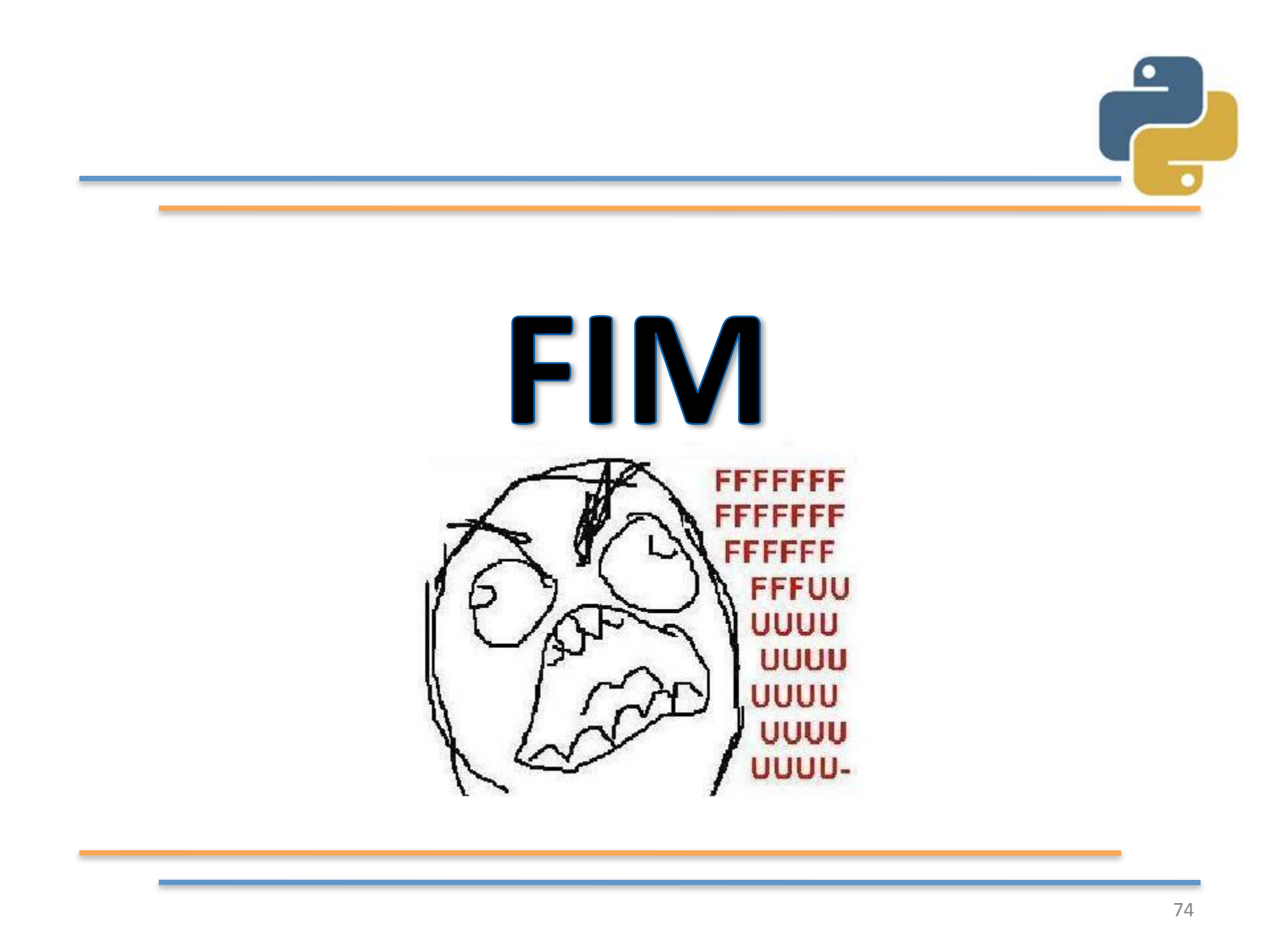
```
class Usuario(Funcionario):
    __metaclass__ = ABCMeta

    def __init__(self, nome=None, email=None,
                 salario=0.0, senha=None):
        super(Usuario, self).__init__(nome, email, salario)
        self.__senha = senha

    @abstractmethod
    def autenticar(self):
        pass

    @property
    def senha(self):
        return self.__senha

    @senha.setter
    def senha(self, valor):
        self.__senha = valor
```

Exercício 6

```
class Diretor(Funcionario, Usuario):
    def __init__(self, senha=None, salario=None):
        super(Diretor, self).__init__(salario=salario)
        self.senha = senha

    def efetuar_login(self, senha):
        return self.senha == senha

    @property
    def bonus(self):
        return self.salario * 1.5
```

Exercício 7

```
class Gerente (Funcionario, Usuario):
    def __init__(self, senha=None, horario_atendimento=None,
                 salario=None):
        super(Gerente, self).__init__(salario=salario)
        self.senha = senha
        self.horario_atendimento = horario_atendimento

    def efetuar_login(self, senha):
        return self.senha == senha

    @property
    def bonus(self):
        return self.salario * 1.2
```



FIM

Contatos

- Márcio Palheta
  - marcio.palheta@gmail.com
  - @marciopalheta
  - https://sites.google.com/site/marciopalheta/

Bibliografia

- LIVRO: Apress - Beginning Python From Novice to Professional
- LIVRO: O'Relly - Learning Python
- http://www.python.org
- http://www.python.org.br

- Mais exercícios:
  - http://wiki.python.org.br/ListaDeExercicios

- Documentação do python:
  - https://docs.python.org/2/

Capítulo 06

Classes e métodos abstratos e Herança múltipla

Márcio Palheta, M.Sc.
marcio.palheta@gmail.com