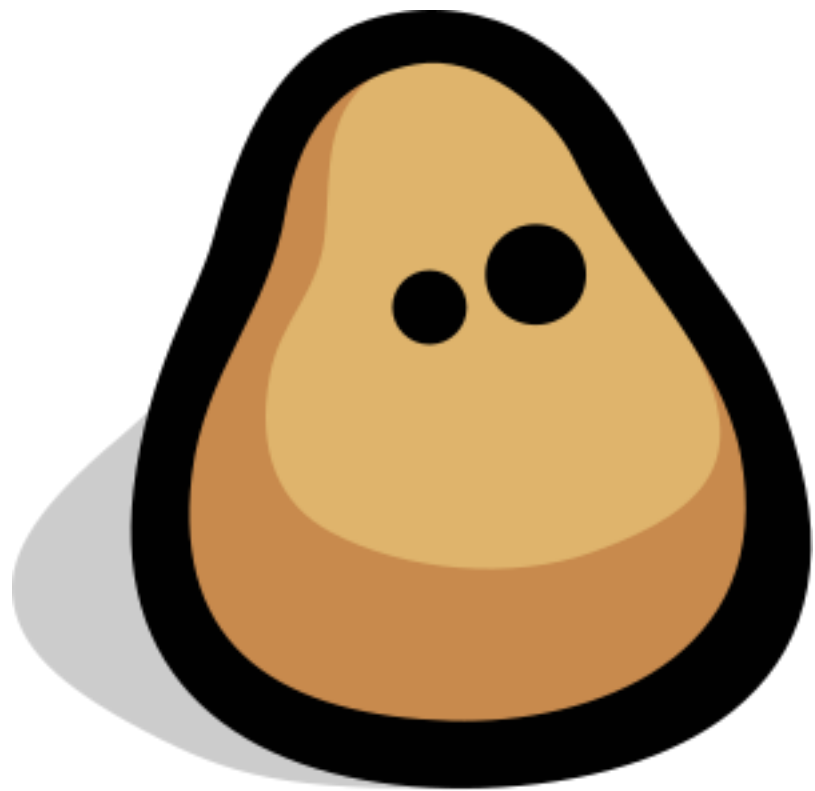


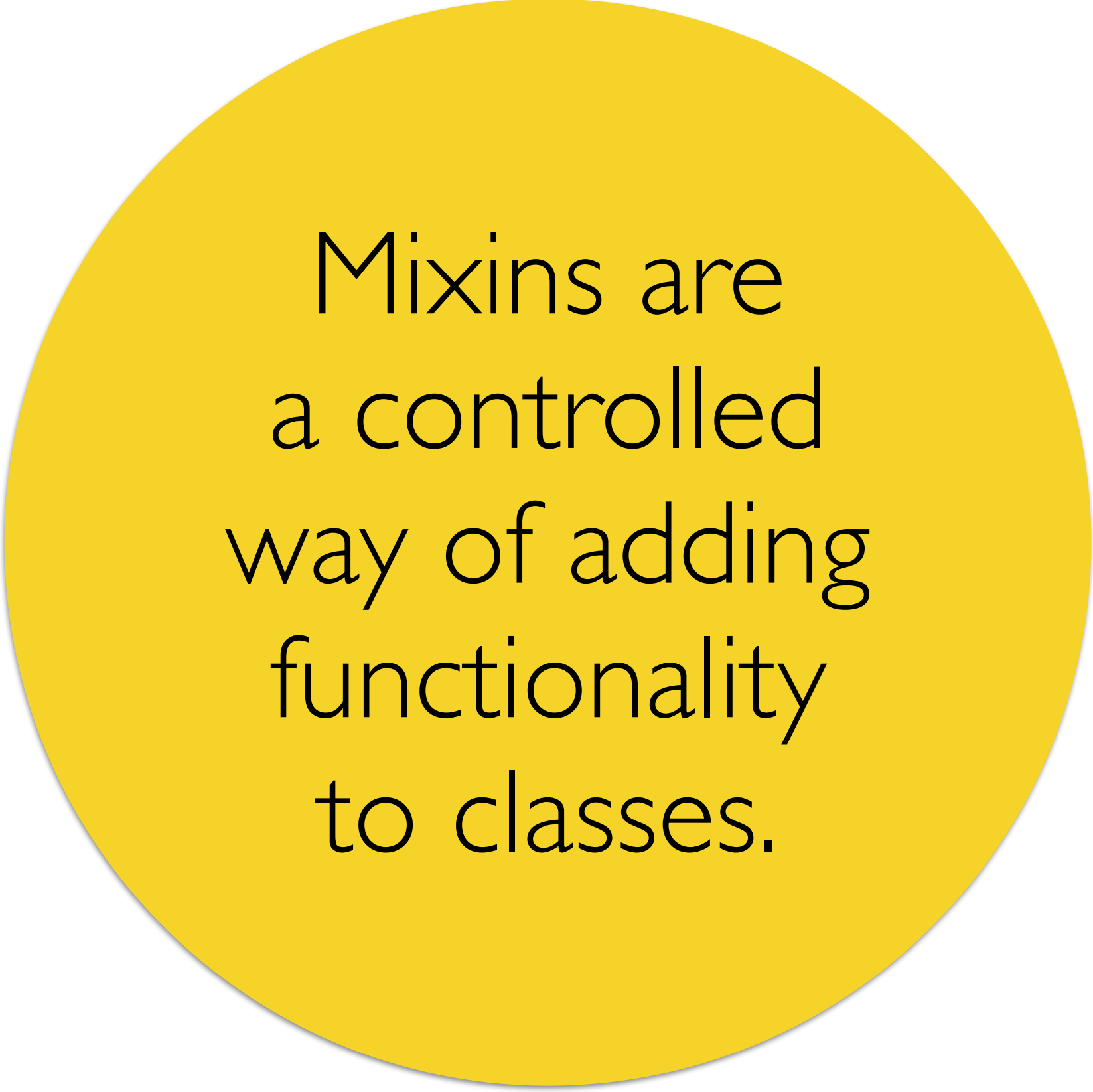
Demystifying Mixins

with 

 @anabalica



I work at Potato



Mixins are
a controlled
way of adding
functionality
to classes.

Mixins are **not**
special language
constructs.

In fact, mixins are ordinary Python classes.

```
1  class SomeMixin(object):  
2      """My smart mixin"""  
3  
4      def test_method(self):  
5          pass
```

Why use mixins?

to improve modularity

When to use mixins?

want to reuse a
particular feature in a lot
of different classes

Properties

Properties

- single responsibility

Properties

- single responsibility
- not meant to be extended

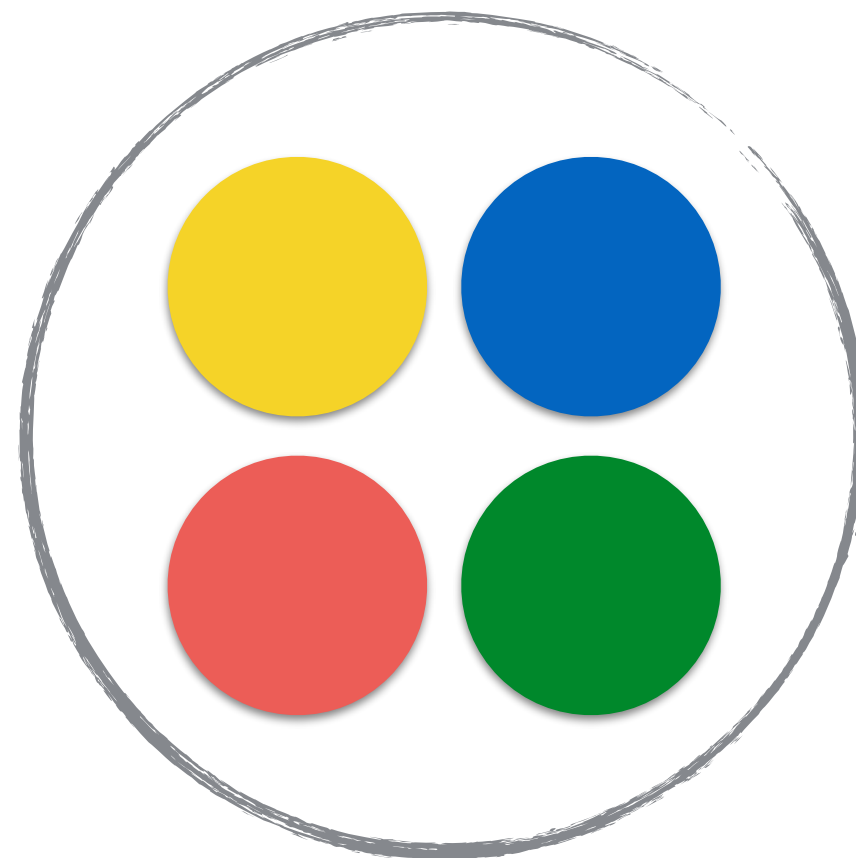
Properties

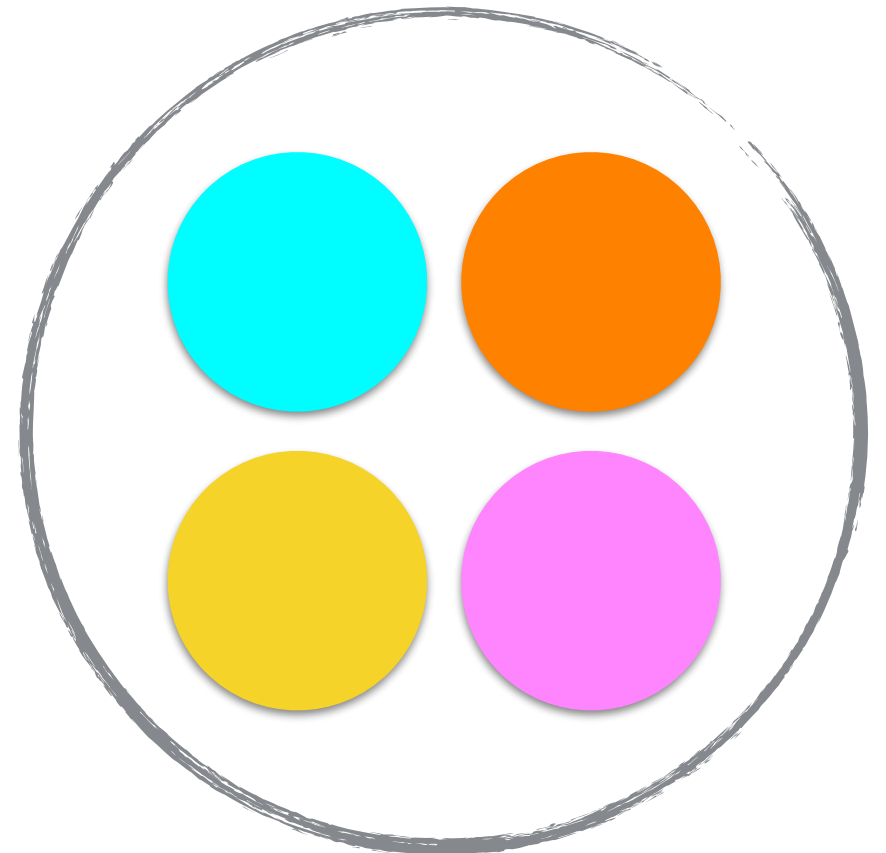
- single responsibility
- not meant to be extended
- not meant to be instantiated

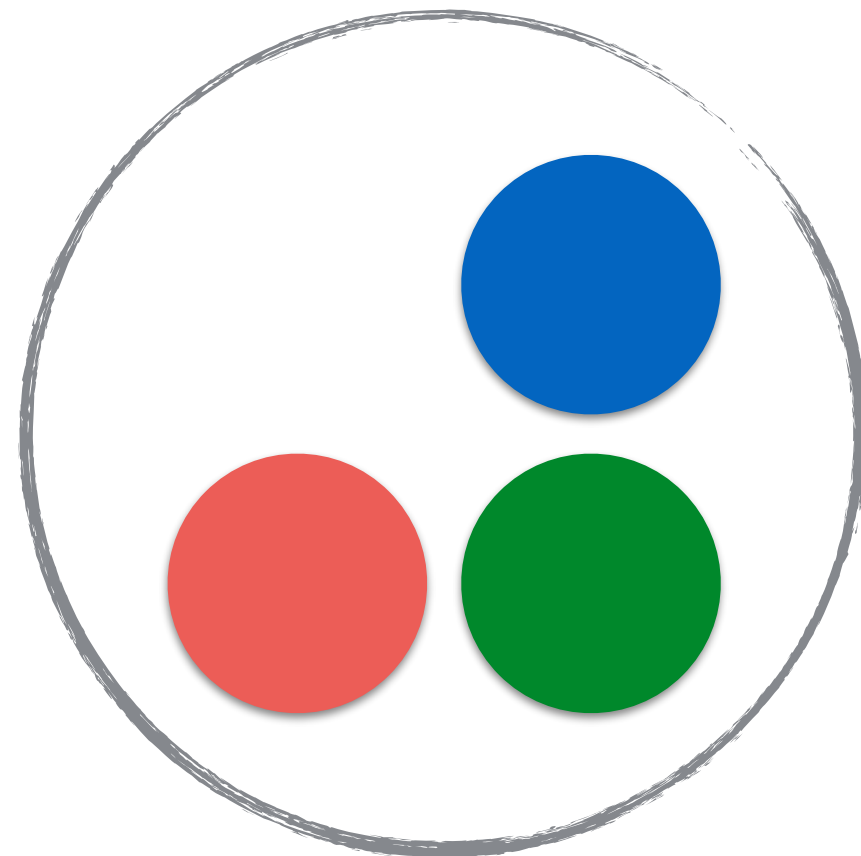
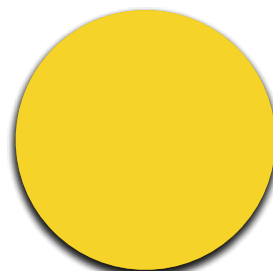
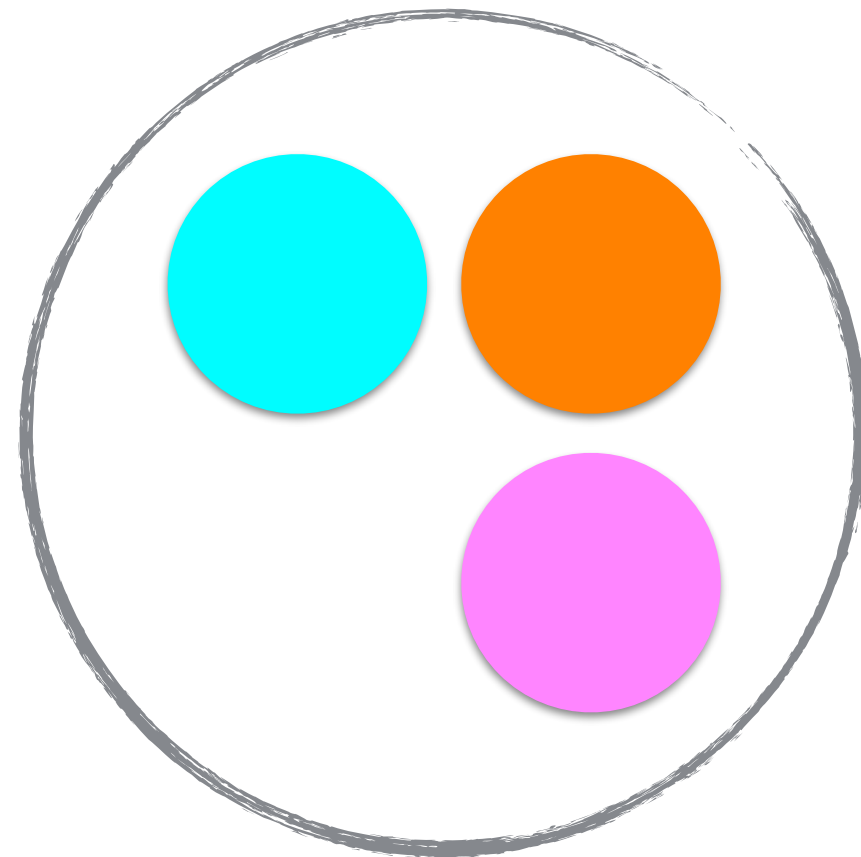


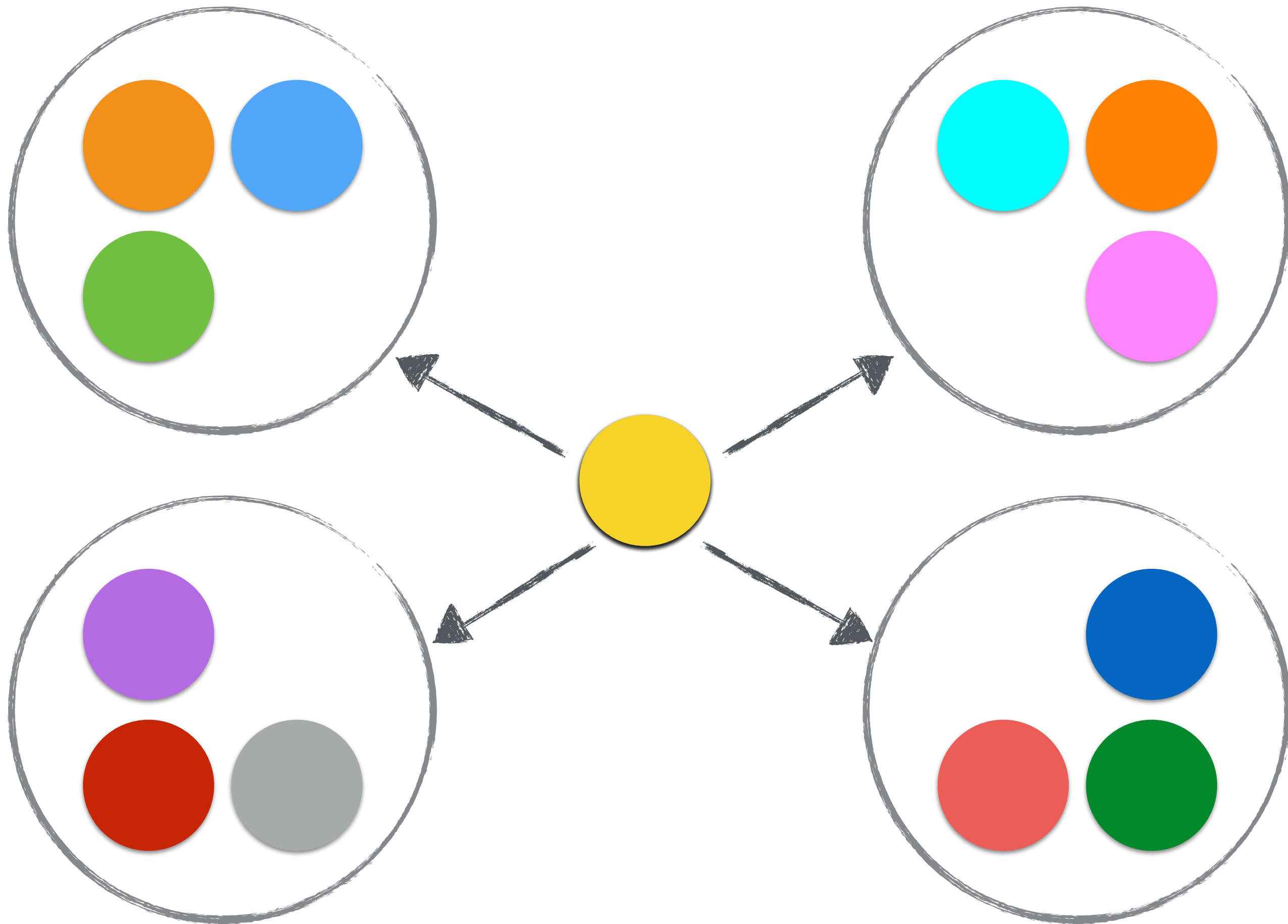












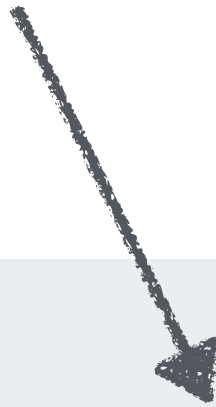
In Python the concept of mixins is implemented using multiple inheritance.



Order matters

```
1  class Foo(BaseFoo, SomeMixin):  
2      pass
```

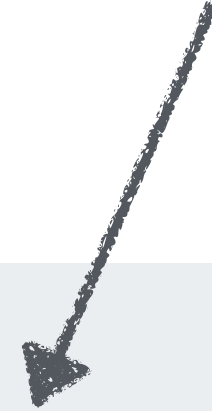
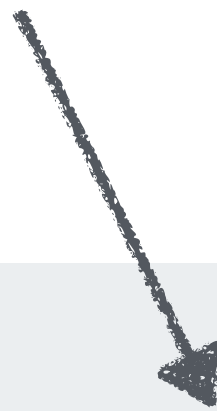
base class



```
1 class Foo(BaseFoo, SomeMixin):  
2     pass
```

base class

mixin



```
1 class Foo(BaseFoo, SomeMixin):  
2     pass
```




```
1 class Foo(BaseFoo, SomeMixin):  
2     pass
```

```
1  class Foo(SomeMixin, BaseFoo):  
2      pass
```



```
1  class Foo(SomeMixin, BaseFoo):  
2      pass
```

```
1  # some_app/views.py
2  from django.views.generic import TemplateView
3
4
5  class AboutView(TemplateView):
6      template_name = "about.html"
```

```
1 # some_app/views.py
2 from django.views.generic import TemplateView
3
4
5 class AboutView(SomeMixin, TemplateView):
6     template_name = "about.html"
```

```
1 # some_app/views.py
2 from django.views.generic import TemplateView
3
4
5 class AboutView(SomeMixin, TemplateView):
6     template_name = "about.html"
```



My first mixin

```
1  # some_app/views.py
2
3
4  class LoginRequiredMixin(object):
5
```



```
1  # some_app/views.py
2
3
4  class LoginRequiredMixin(object):
5
6      def dispatch(self, request, *args, **kwargs):
7
```

```
1  # some_app/views.py
2  from django.core.exceptions import PermissionDenied
3
4
5  class LoginRequiredMixin(object):
6
7      def dispatch(self, request, *args, **kwargs):
8          if not request.user.is_authenticated():
9              raise PermissionDenied
10
```

```
1  # some_app/views.py
2  from django.core.exceptions import PermissionDenied
3
4
5  class LoginRequiredMixin(object):
6
7      def dispatch(self, request, *args, **kwargs):
8          if not request.user.is_authenticated():
9              raise PermissionDenied
10
11         return super(LoginRequiredMixin, self).\
12             dispatch(request, *args, **kwargs)
13
```

```
1 # some_app/views.py
2 from django.views.generic import TemplateView
3
4
5 class AboutView(LoginRequiredMixin, TemplateView):
6     template_name = "about.html"
```

LoginRequiredMixin

TemplateView

AboutView

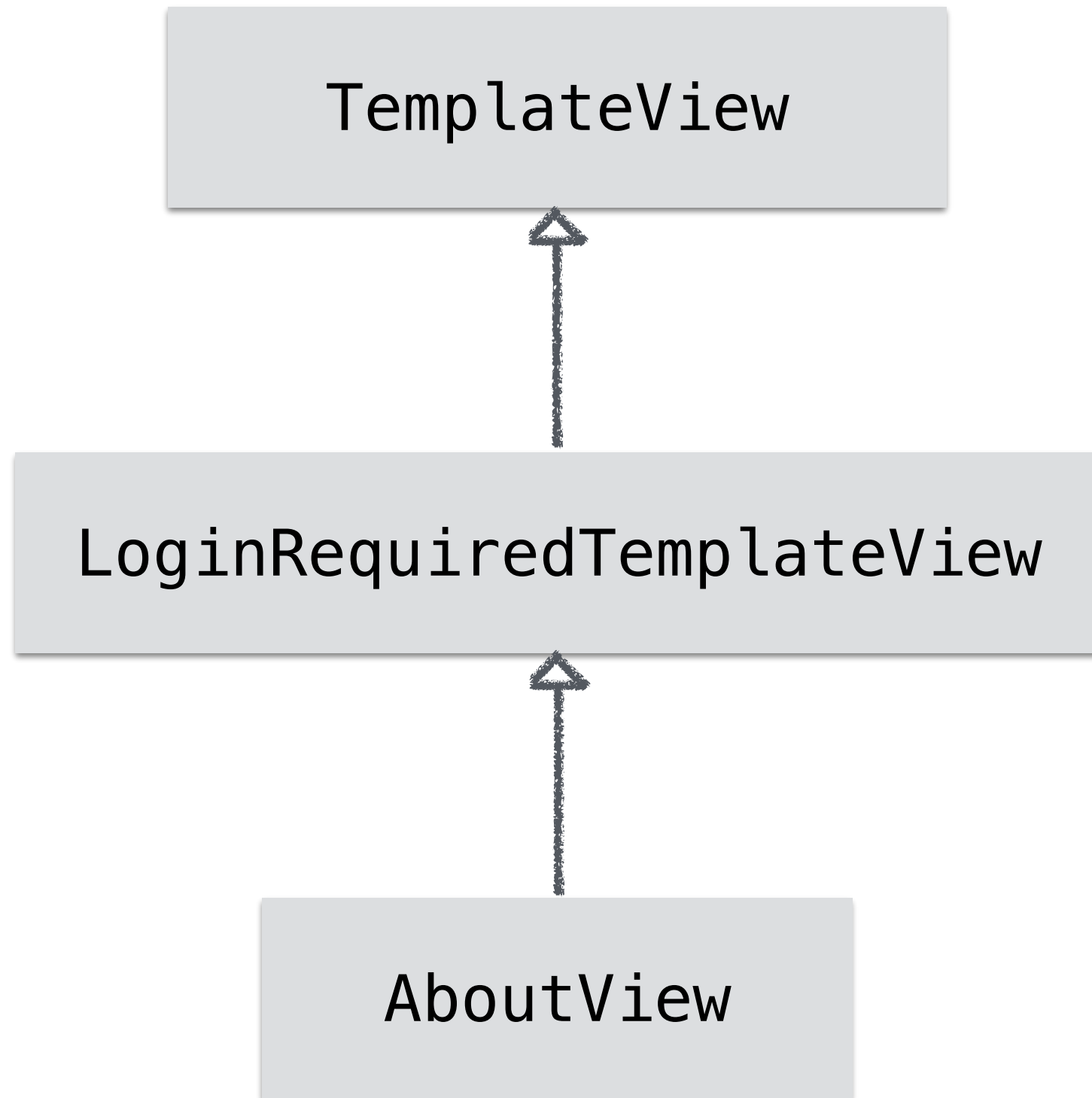


LoginRequiredMixin

DetailView

AboutView





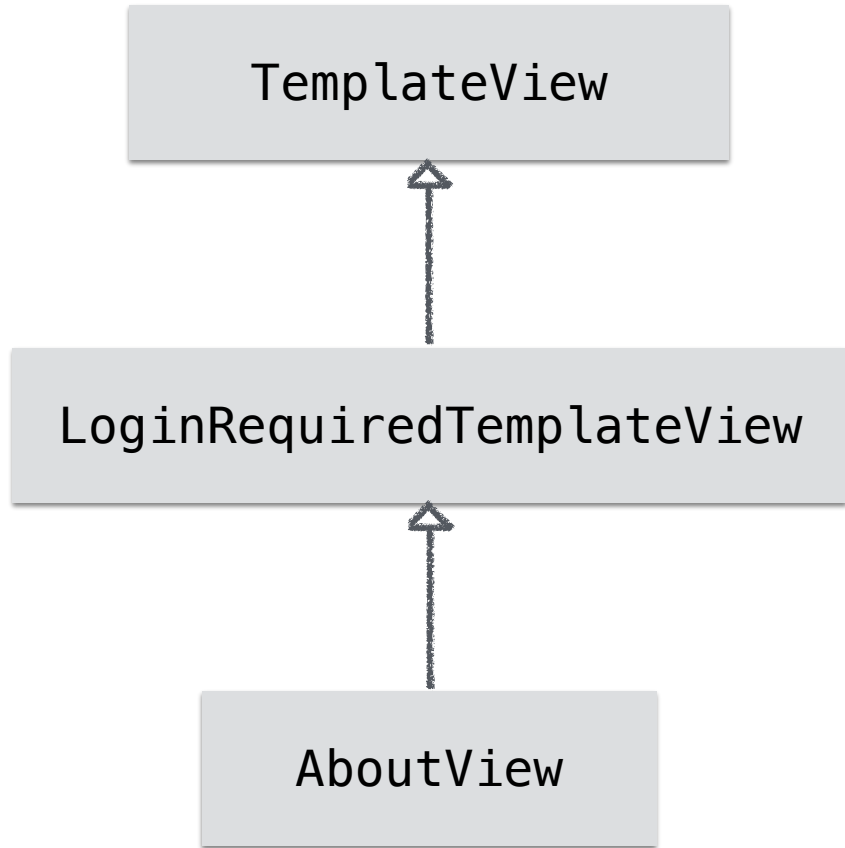
TemplateView

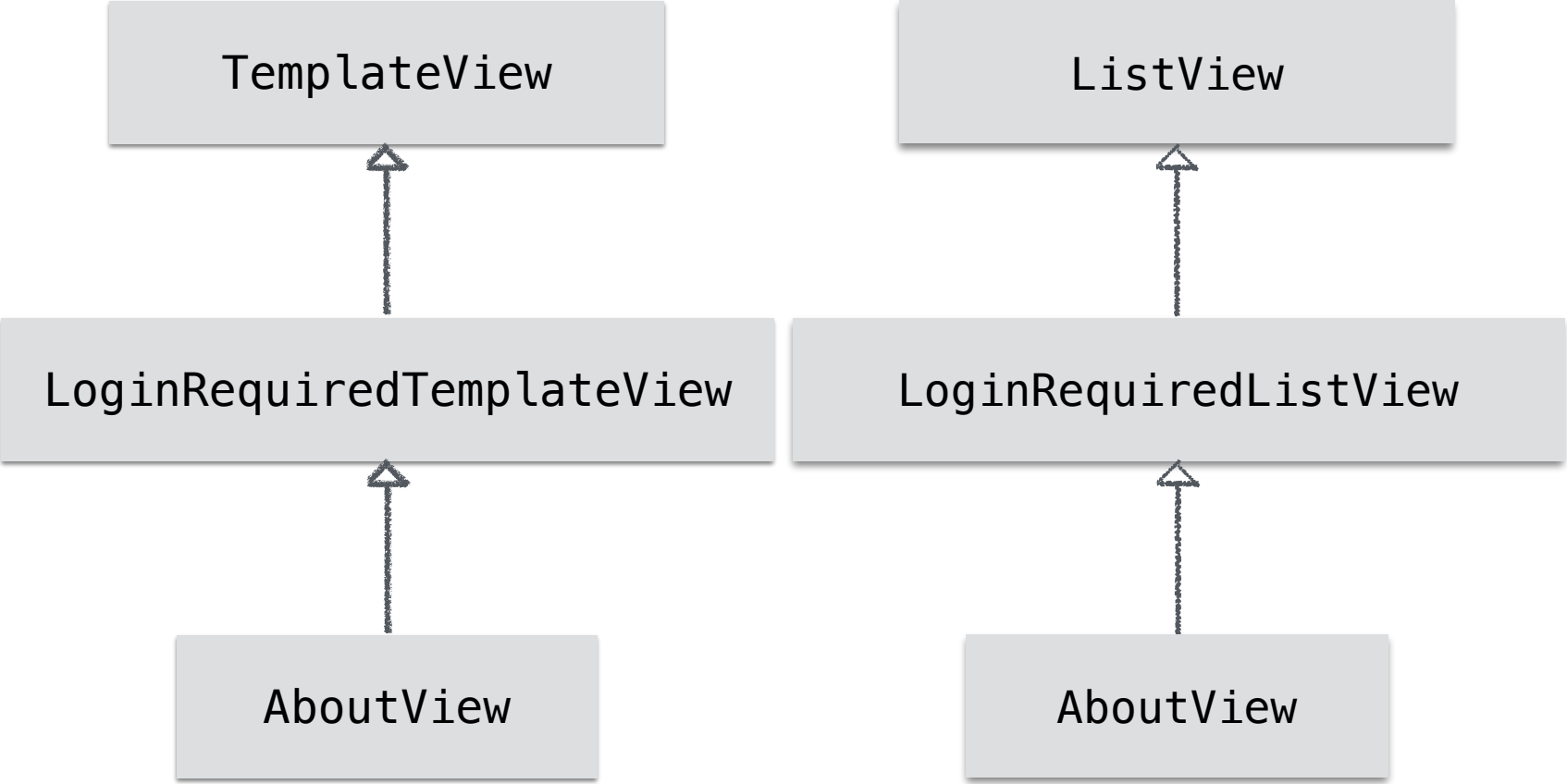


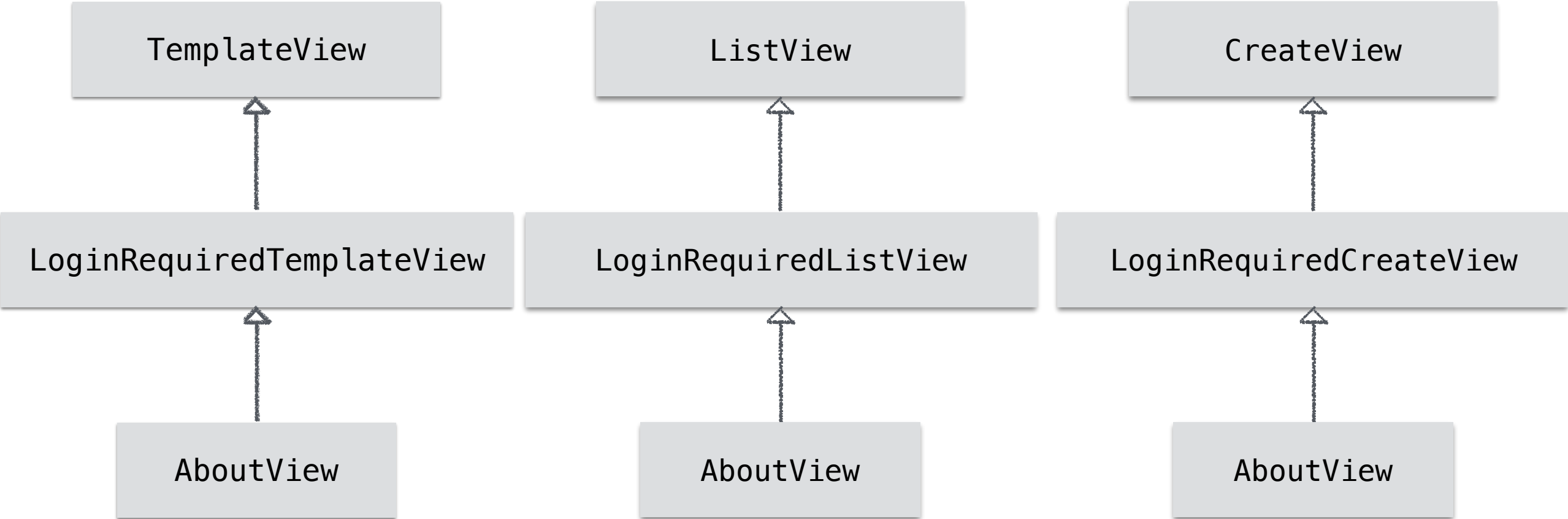
LoginRequiredTemplateView

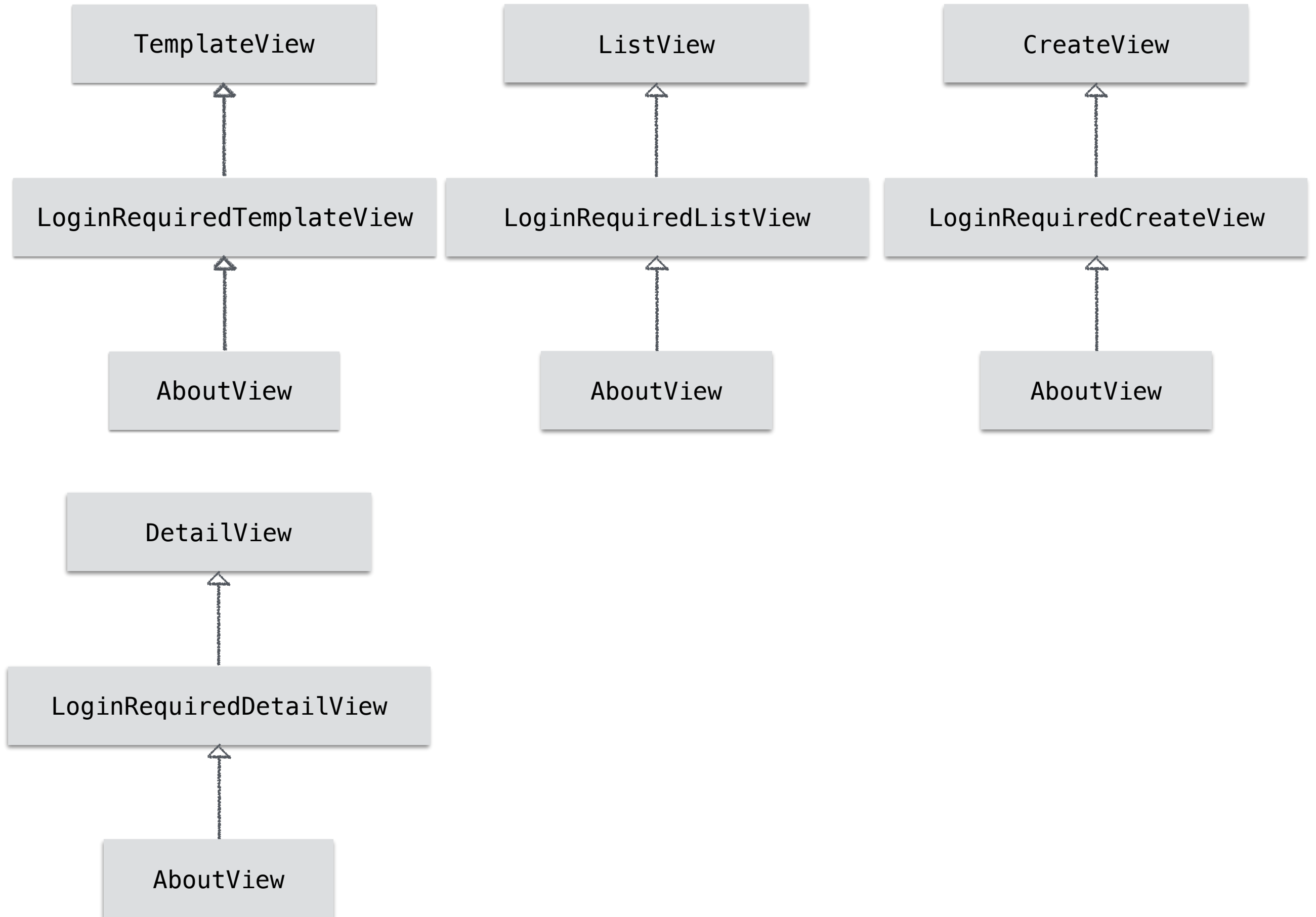


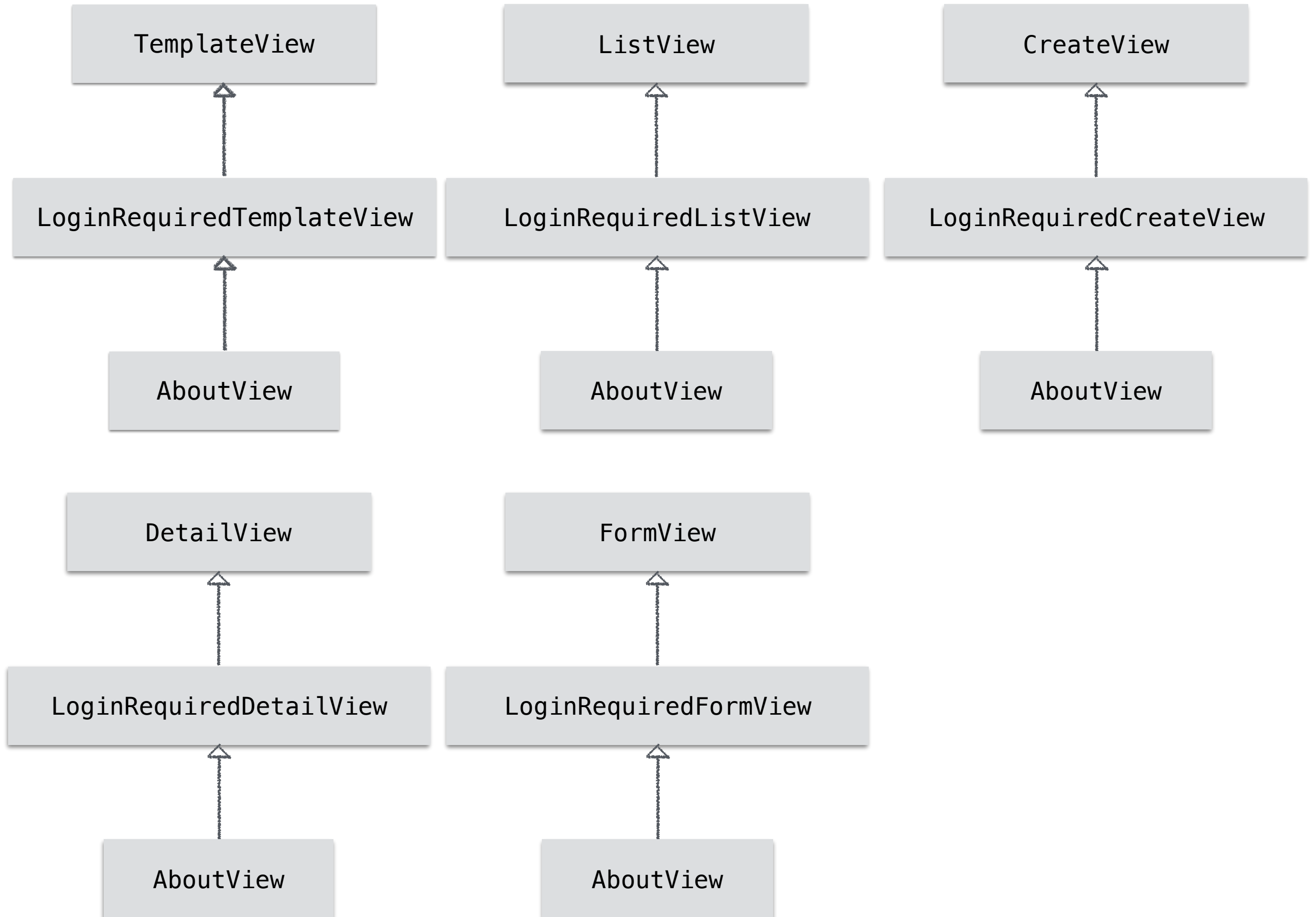
AboutView

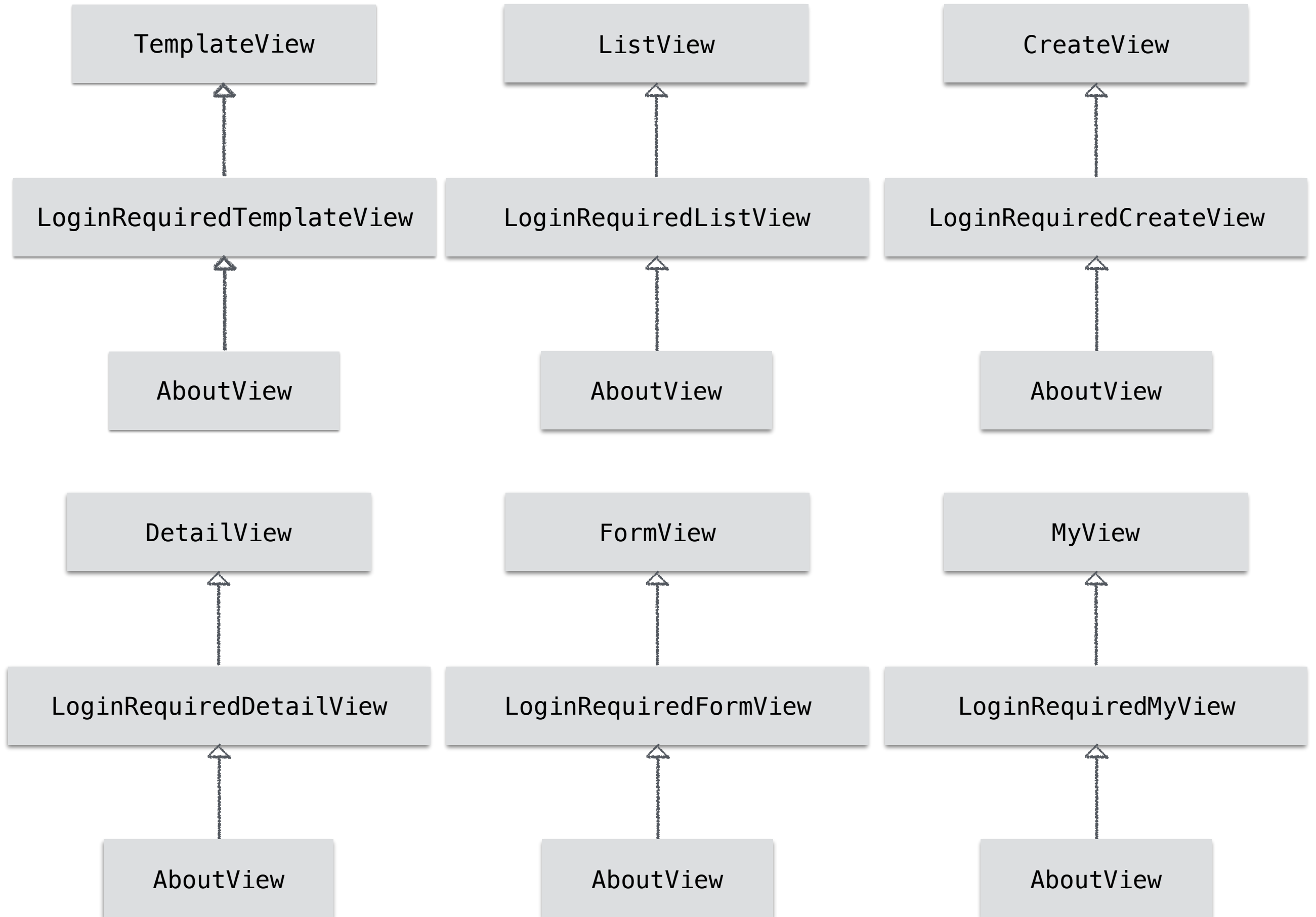












LoginRequiredMixin

TemplateView

AboutView



```
dispatch()
```

```
dispatch()
```

check if user is logged in,
has permission


```
dispatch()
```



```
get_context_data()
```

check if user is logged in,
has permission

```
dispatch()
```



```
get_context_data()
```

check if user is logged in,
has permission

add new data to
the context

`dispatch()`

check if user is logged in,
has permission



`get_context_data()`

add new data to
the context



`get_template_names()`

`dispatch()`

check if user is logged in,
has permission



`get_context_data()`

add new data to
the context



`get_template_names()`

add more flexibility to
the template names

```
1 # some_app/views.py
2 from django.views.generic import TemplateView
3
4
5 class AboutView(TemplateView):
6     template_name = "about.html"
```

`dispatch()`

check if user is logged in,
has permission



`get_context_data()`

add new data to
the context



`get_template_names()`

add more flexibility to
the template names

[docs.djangoproject.com/en/1.8/ref/
class-based-views/base/](https://docs.djangoproject.com/en/1.8/ref/class-based-views/base/)

[docs.djangoproject.com/en/1.8/ref/
class-based-views/base/](https://docs.djangoproject.com/en/1.8/ref/class-based-views/base/)

[docs.djangoproject.com/en/1.8/topics/
class-based-views/mixins/](https://docs.djangoproject.com/en/1.8/topics/class-based-views/mixins/)

[docs.djangoproject.com/en/1.8/topics/
class-based-views/mixins/](https://docs.djangoproject.com/en/1.8/topics/class-based-views/mixins/)

ccbv.co.uk/

django-braces

django-braces

Access
Mixins

django-braces

Access
Mixins

Form
Mixins

django-braces

Access
Mixins

Form
Mixins

Other
Mixins

Decorators

Decorators

`login_required()`

Decorators

`login_required()`

`user_passes_test()`

Decorators

`login_required()`

`user_passes_test()`

`permission_required()`

Good news everyone!



Django 1.9

Django 1.9

LoginRequiredMixin

Django 1.9

LoginRequiredMixin

UserPassesTestMixin

Django 1.9

LoginRequiredMixin

UserPassesTestMixin

PermissionRequiredMixin

Runtime magic



```
1 class CuteMixin:
2     def be_cute(self):
3         print "{} 🌸❤️😊❤️🌸".format(self.name)
4
```

```
1 class CuteMixin:
2     def be_cute(self):
3         print "{} 🌸❤️😊❤️🌸".format(self.name)
4
5
6 class Mascot:
7     def __init__(self, name):
8         self.name = name
9
```

```
1  class CuteMixin:
2      def be_cute(self):
3          print "{} 🌸❤️😊❤️🌸".format(self.name)
4
5
6  class Mascot:
7      def __init__(self, name):
8          self.name = name
9
10
11 if __name__ == '__main__':
12     domo = Mascot("Domo-kun")
13     Mascot.__bases__ += (CuteMixin, )
14     domo.be_cute()
```

```
1  class CuteMixin:
2      def be_cute(self):
3          print "{} 🌸❤️🌸".format(self.name)
4
5
6  class Mascot:
7      def __init__(self, name):
8          self.name = name
9
10
11 if __name__ == '__main__':
12     domo = Mascot("Domo-kun")
13     Mascot.__bases__ += (CuteMixin, )
14     domo.be_cute()
```

Domo-kun ♡♡♡♡



```
1  class CuteMixin:
2      def be_cute(self):
3          print "{} 🌸❤️🌸".format(self.name)
4
5
6  class Mascot:
7      def __init__(self, name):
8          self.name = name
9
10
11 if __name__ == '__main__':
12     domo = Mascot("Domo-kun")
13     Mascot.__bases__ += (CuteMixin, )
14     domo.be_cute()
```

```
1 class CuteMixin:
2     def be_cute(self):
3         print("{} 🌸❤️😊❤️🌸".format(self.name))
4
5
6 class Mascot:
7     def __init__(self, name):
8         self.name = name
9
```



```
1  class CuteMixin:
2      def be_cute(self):
3          print "{} 🍷❤️🍷".format(self.name)
4
5
6  class Mascot:
7      def __init__(self, name):
8          self.name = name
9
10
11 if __name__ == '__main__':
12     kumamon = Mascot("Kumamon")
13     kumamon.__class__ = type('SomeNewType',
14                             (Mascot, CuteMixin), {})
15     kumamon.be_cute()
```

```
1  class CuteMixin:
2      def be_cute(self):
3          print "{} 🍷💕💕🍷".format(self.name)
4
5
6  class Mascot:
7      def __init__(self, name):
8          self.name = name
9
10
11 if __name__ == '__main__':
12     kumamon = Mascot("Kumamon")
13     kumamon.__class__ = type('SomeNewType',
14                             (Mascot, CuteMixin), {})
15     kumamon.be_cute()
```

Kumamon ♡♡😊♡♡





With great
power comes great
responsibility

Recap

Recap

- single responsibility

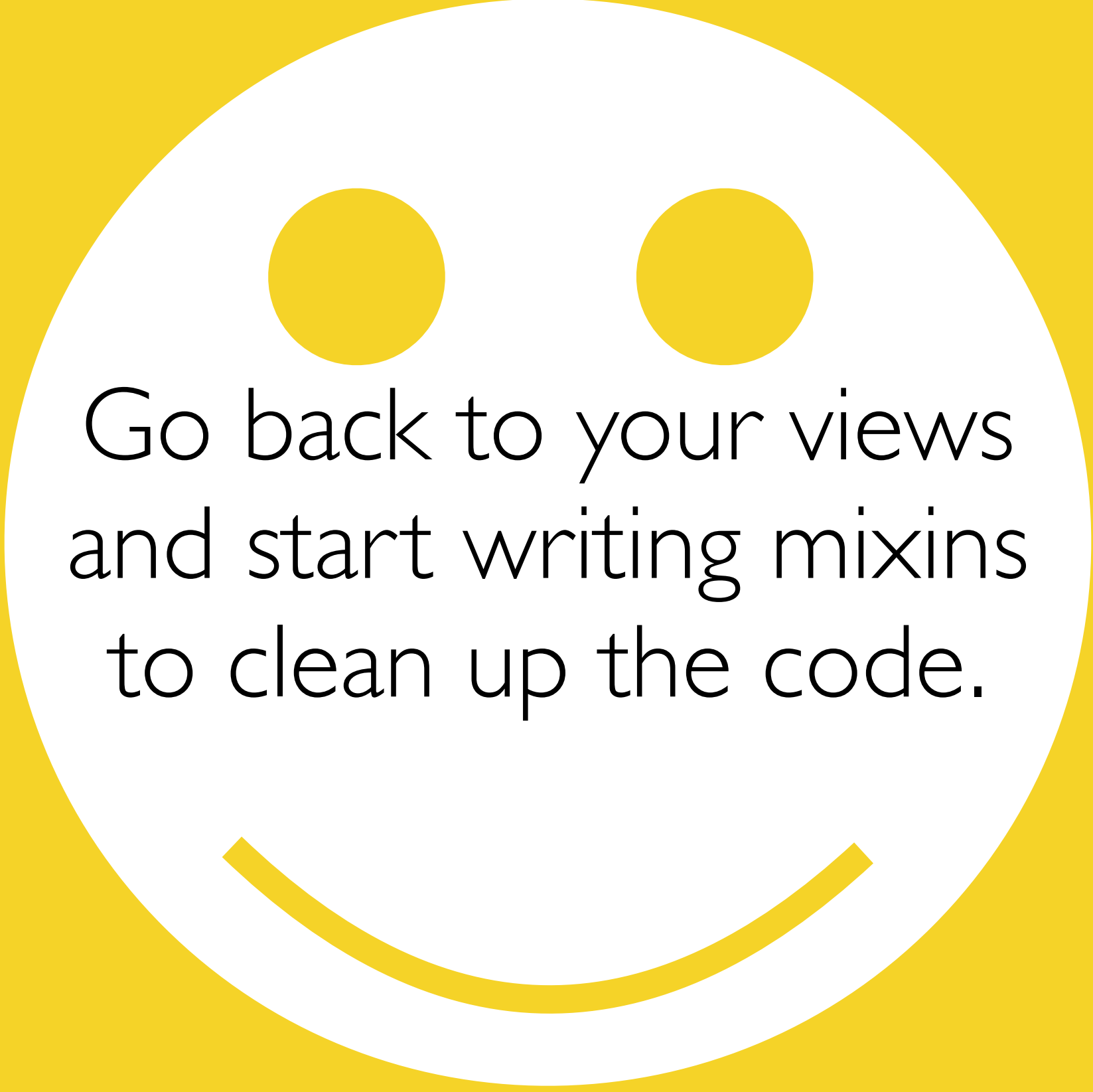
Recap

- single responsibility
- plug-in functionality

Recap

- single responsibility
- plug-in functionality
- isn't creating a subtyping relation

Go back to your views
and start writing mixins
to clean up the code.



Go back to your views
and start writing mixins
to clean up the code.