Selenium Design Patterns: Beyond the Page Object

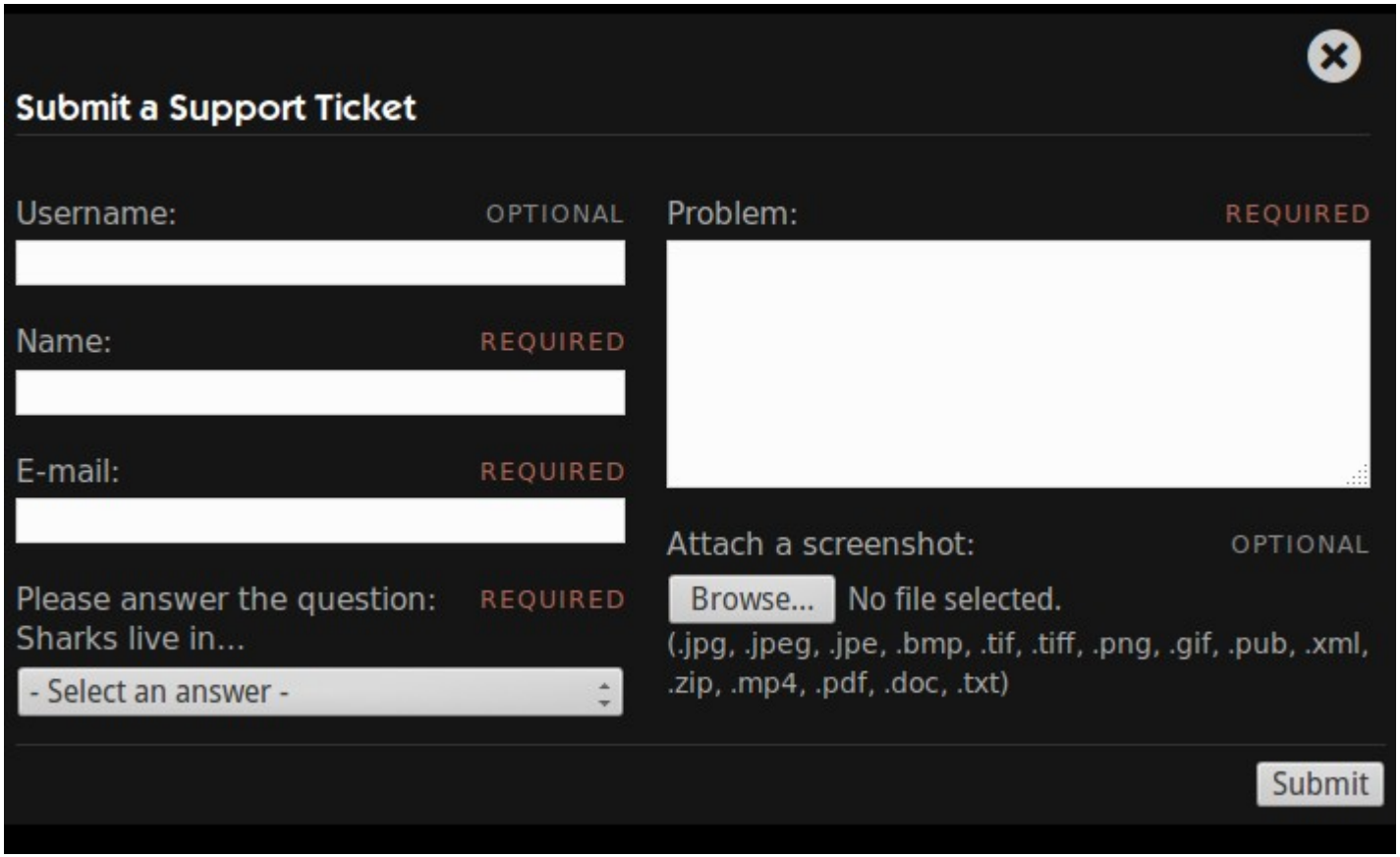
Selenium Conference 2014
Sept. 5, 2014

Derrick Kearney
Purdue University
telldsk@gmail.com

Agenda

- WebForm Pattern
- ItemList Pattern
- IframeWrap Pattern

Working with Web Forms



WebForm Pattern

Standardize the interface for filling out a web form.

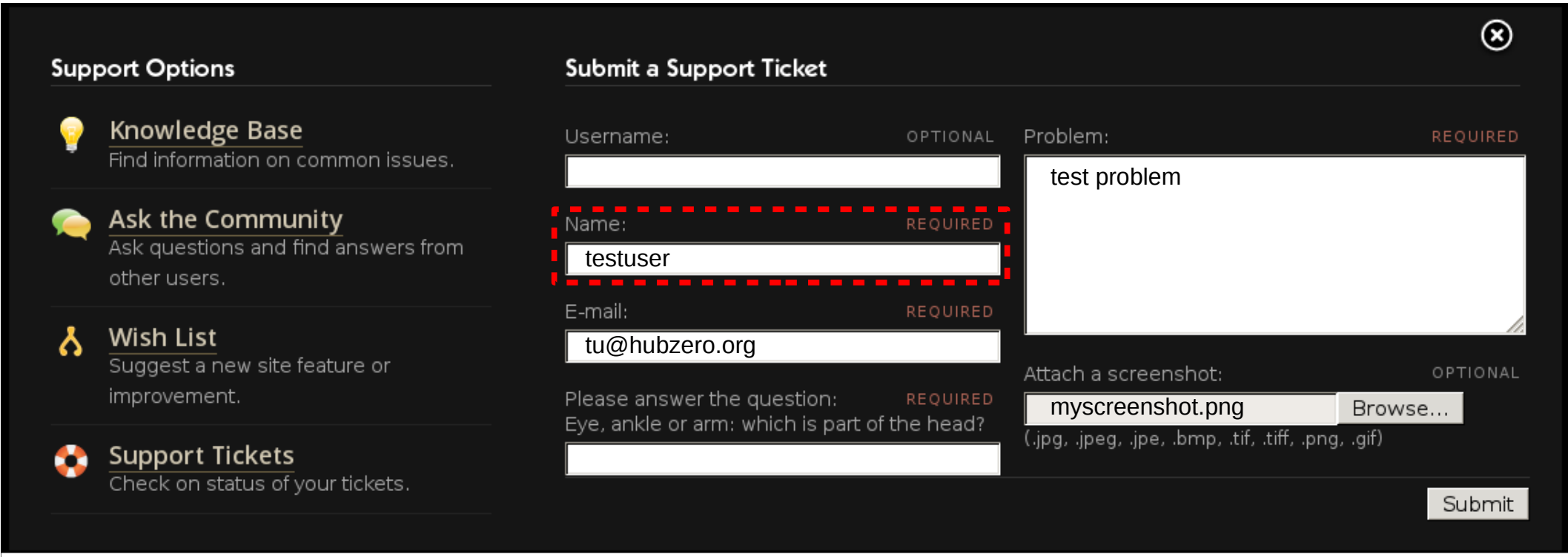
Usual Interface

```
po = TroubleReportForm()

po.set_name('testuser')
po.set_email('tu@hubzero.org')
po.set_problem('test problem')
po.set_upload('myscreen.png')

po.submit.click()
```

Proposed Interface

```
po = TroubleReportForm()

data = {
    'name'    : 'testuser',
    'email'   : 'tu@hubzero.org',
    'problem' : 'test problem',
    'upload'  : 'myscreenshot.png',
}
po.populate_form(data)

po.submit_form()
```

1. Give page objects a 'value'




```
...
e = browser.find_element_by_id('trName')
e.clear()
e.send_keys("testuser")
...
```

→

```
...
name = Text('#trName')
name.value = 'testuser'
...
```

Setting the value of a Checkbox

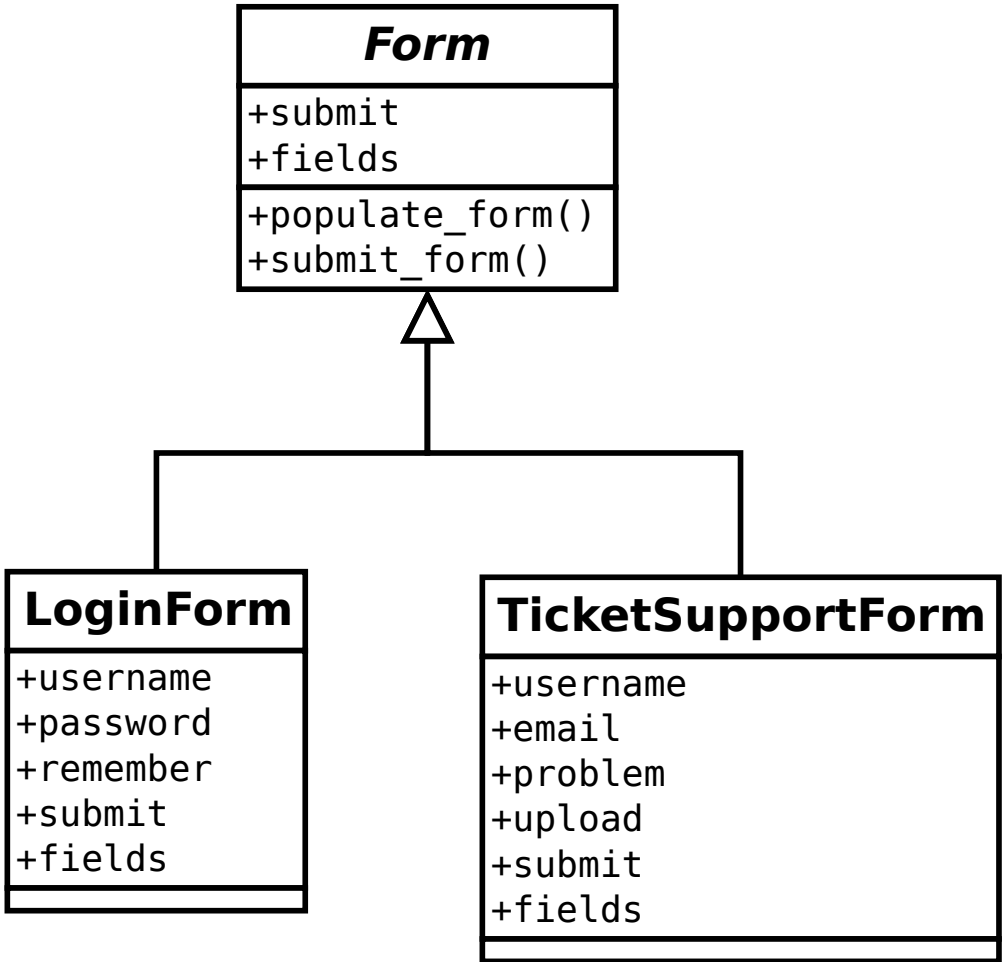
Usual Interface

```
# get the value

e = browser.find_element_by_id('ckbox')
e.is_selected()


# set the value

e = browser.find_element_by_id('ckbox')
If e.is_selected() is not val:
    e.click()
```

Proposed Interface

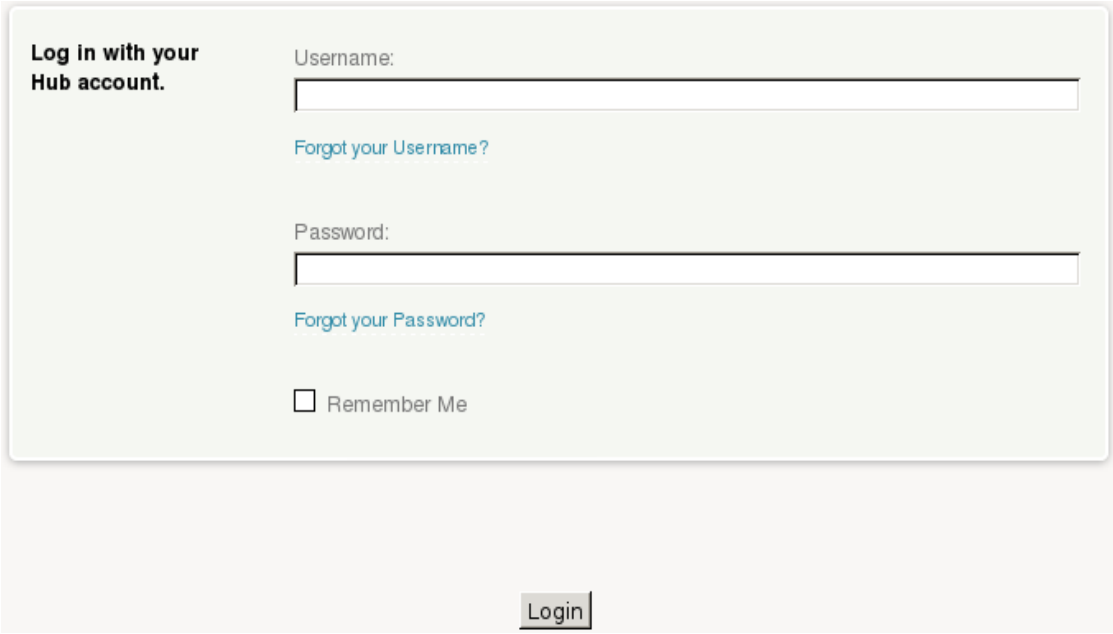
```
# get the value

e = Checkbox('#ckbox')
x = e.value


# set the value

e = Checkbox('#ckbox')
e.value = True
```

2. Use an abstract *Form* base class



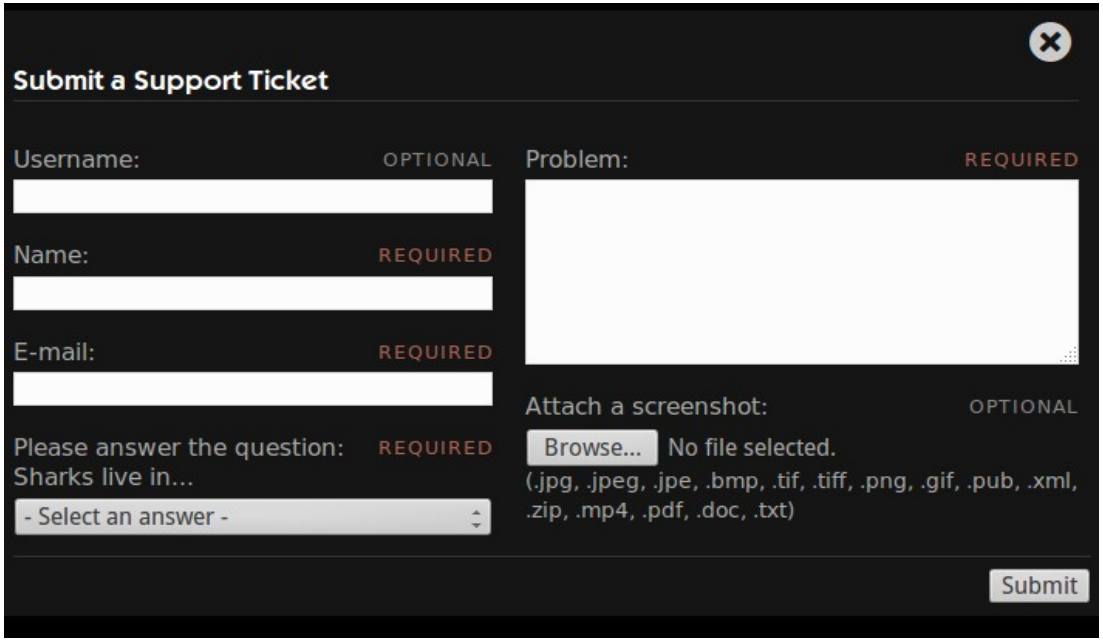
2. Use an abstract *Form* base class

```
class LoginForm(object):
    def __init__(self):
        ...

    def set_username(self,username):
        ...

    def set_password(self,username):
        ...

    def login_as(self,username,passwd):
        ...
```

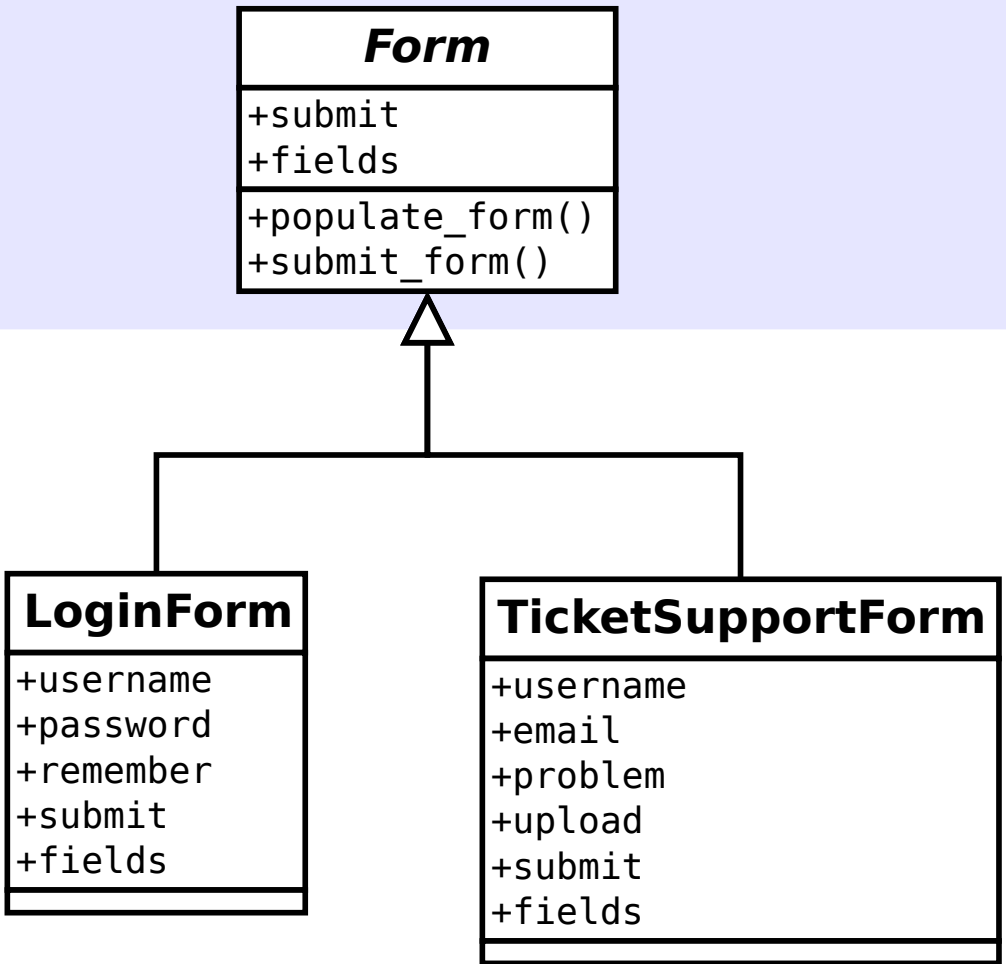


2. Use an abstract *Form* base class

```
class TroubleReportForm(object):
    def __init__(self):
        ...

    def set_username(self,username):
        ...

    def set_name(self,name):
        ...

    def set_email(self,address):
        ...

    def submit_ticket(self,username,name,...):
        ...
```

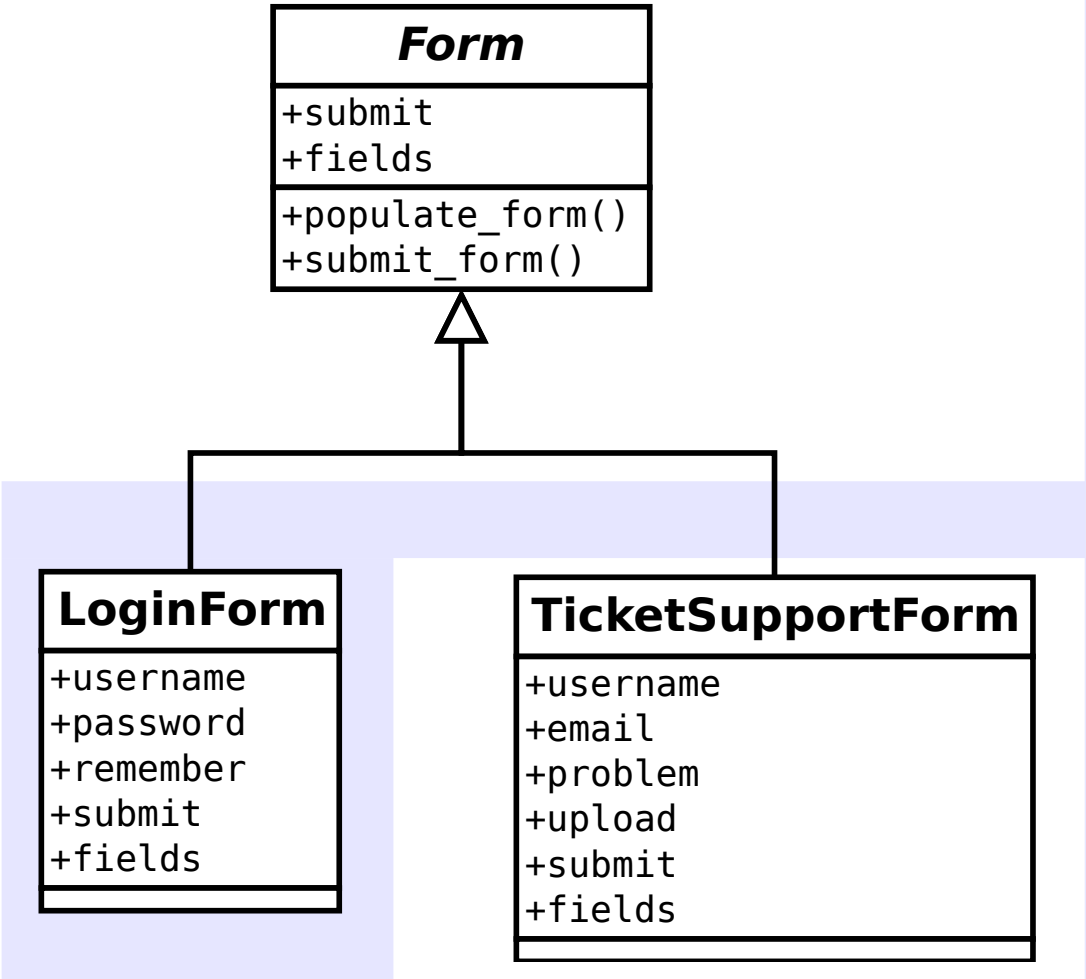


Proposed Interface: Form class



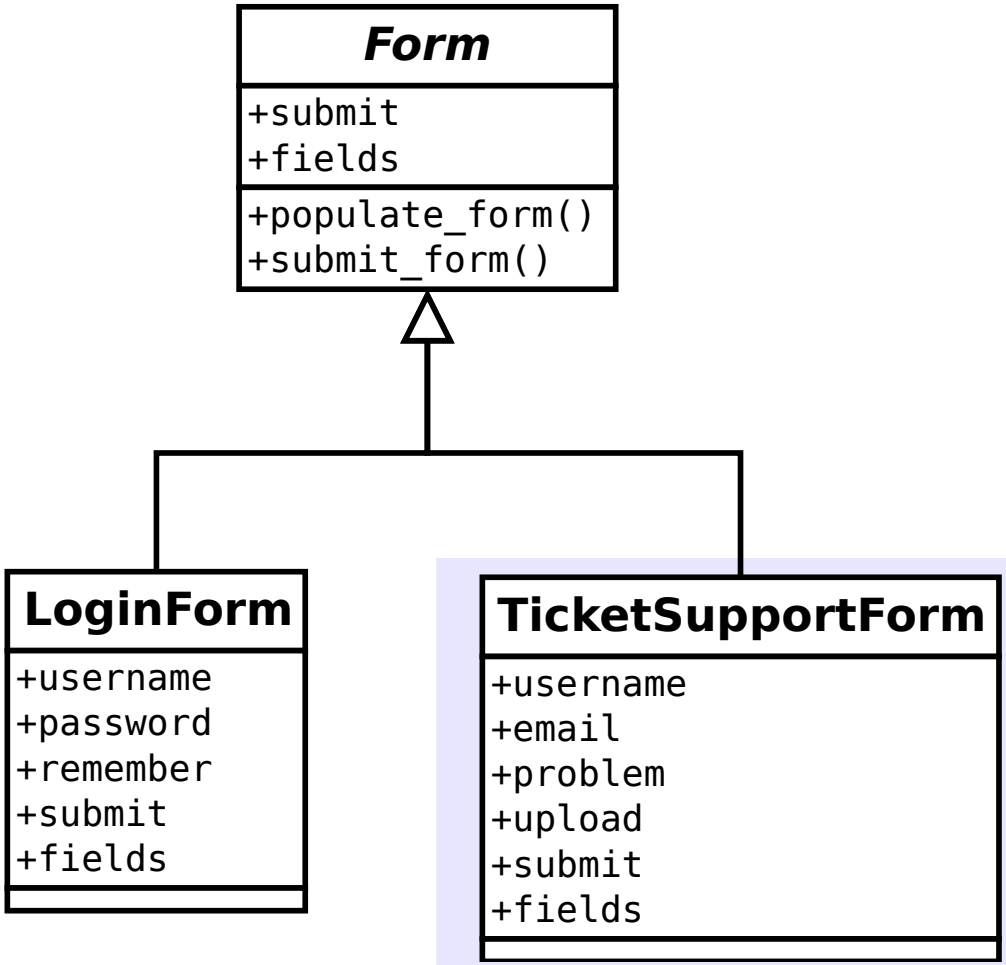

```
class Form(object):
    ...

    def populate_form(self, data):
        for (k,v) in data:
            # find the widget in the object's
            # dictionary and set its value
            widget = getattr(self,k)
            widget.value = v

    def submit_form(self,data={}):
        self.populate_form(data)
        result = self.submit.click()
        return result
```

LoginForm subclasses Form



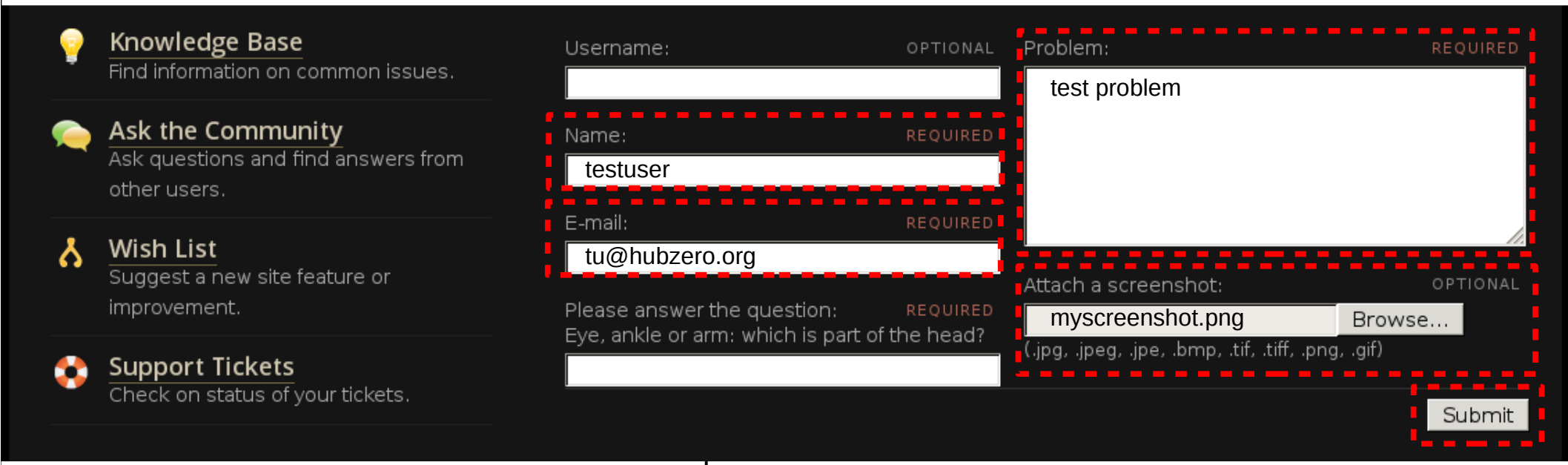

```
class LoginForm(Form):

    def __init__ (self):

        self.username   = Text('#userame')
        self.password   = Text('#password')
        self.remember   = Checkbox('#rmbr')
        self.submit     = Button('#submit')
```

TicketSupportForm subclasses Form




```
class TicketSupportForm(Form):

    def __init__ (self):

        self.name        = Text('#trName')
        self.email       = Text('#trEmail')
        self.problem     = Text('#trProblem')
        self.upload      = Text('#trUpload')
        self.submit      = Button('#submit')
```

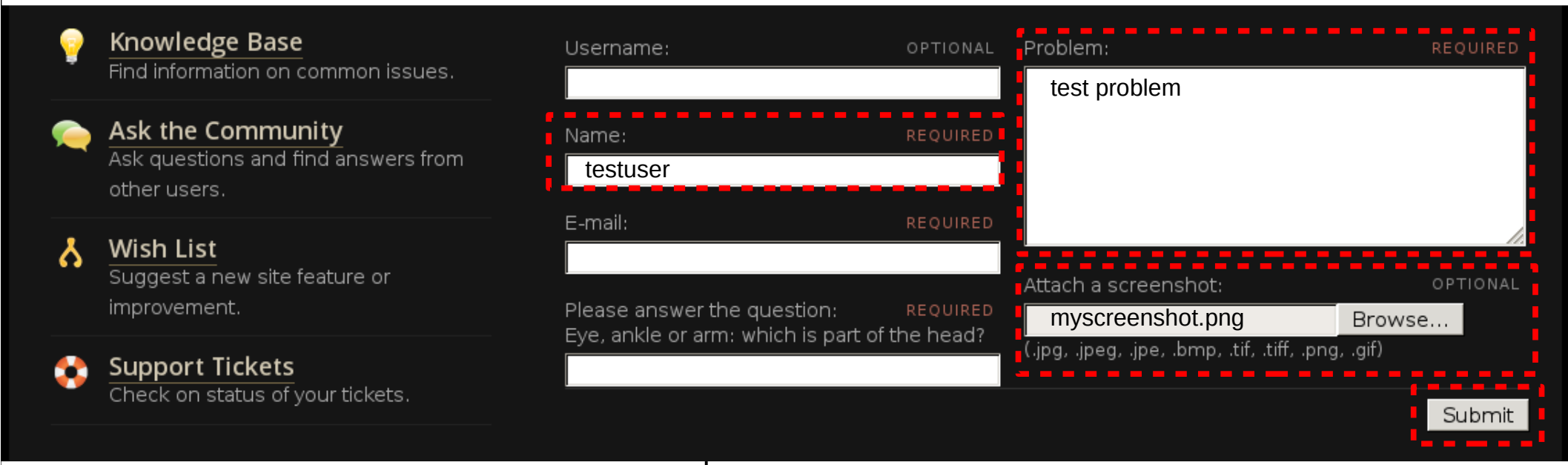
3. Organize form data into dictionaries




```
po = TroubleReportForm()

po.submit_ticket(
        'testuser',
        'tu@hubzero.org',
        'test problem',
        'myscreenshot.png'
)
```

```
po = TroubleReportForm()

data = {
    'name'    : 'testuser',
    'email'   : 'tu@hubzero.org',
    'problem' : 'test problem',
    'upload'  : 'myscreenshot.png',
}
po.populate_form(data)

po.submit_form()
```

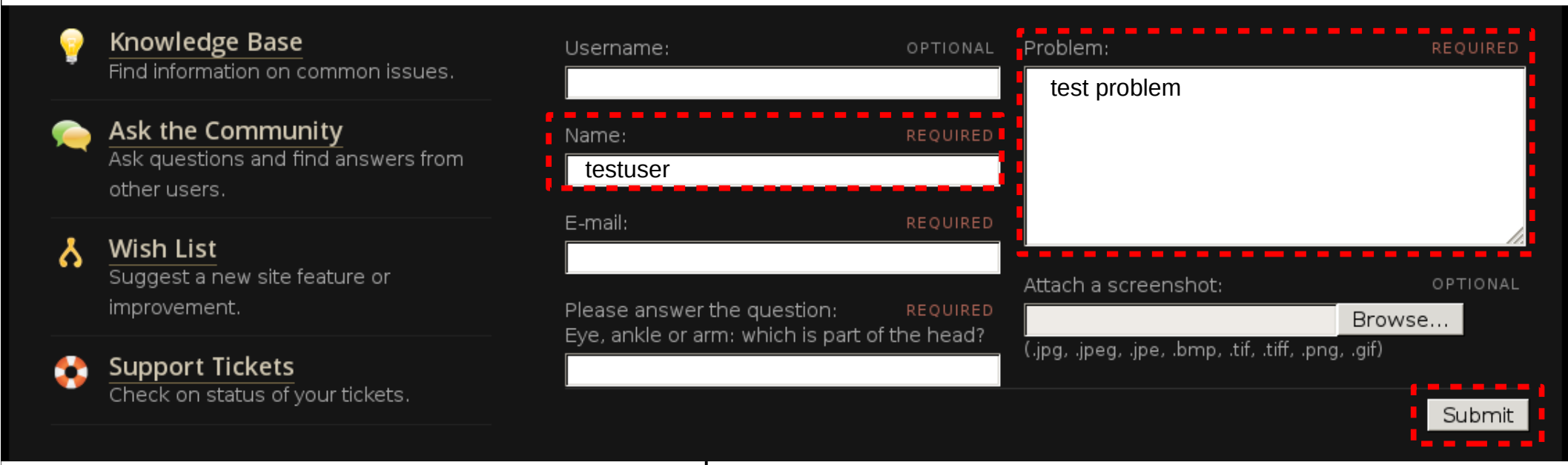
3. Organize form data into dictionaries




```
po = TroubleReportForm()

po.submit_ticket(
        'testuser',
        'tu@hubzero.org',
        'test problem',
        'myscreenshot.png'
)
```

```
po = TroubleReportForm()

data = {
    'name'    : 'testuser',

    'problem' : 'test problem',
    'upload'  : 'myscreenshot.png',
}
po.populate_form(data)

po.submit_form()
```

3. Organize form data into dictionaries




```
po = TroubleReportForm()

po.submit_ticket(
        'testuser',
        'tu@hubzero.org',
        'test problem',
        'myscreenshot.png'
)
```

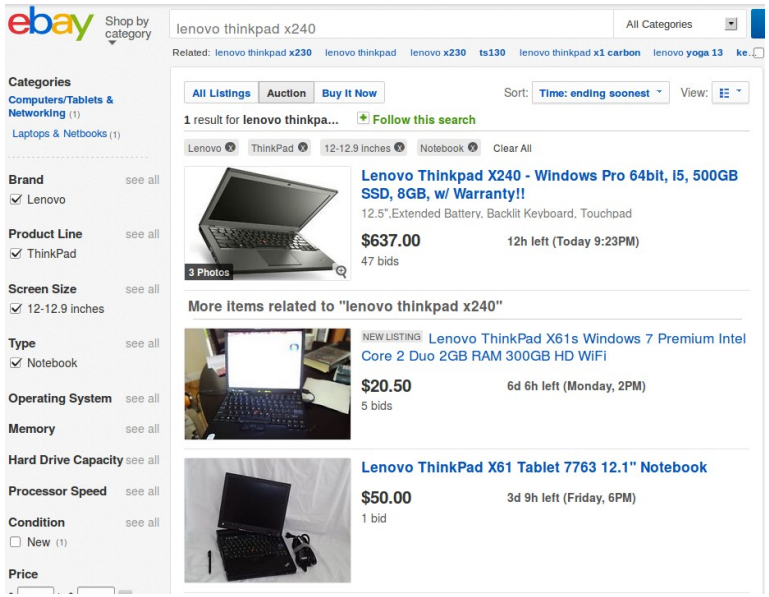
```
po = TroubleReportForm()

data = {
    'name'     : 'testuser',

    'problem' : 'test problem',

}
po.populate_form(data)

po.submit_form()
```

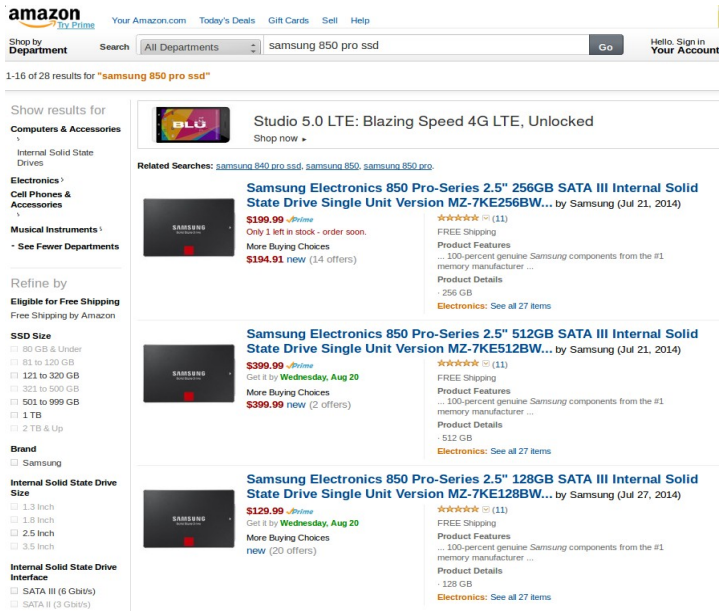
Extending the Web Form Pattern

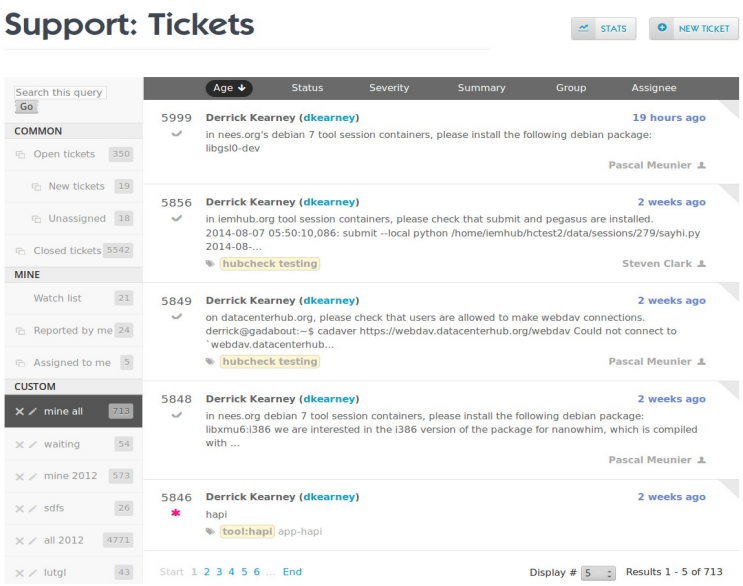
Many types of Forms can extend the base class

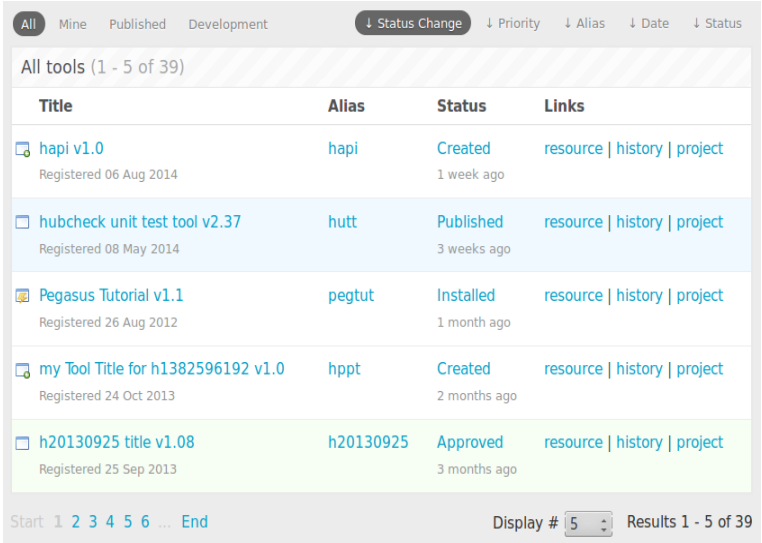
- Simple Forms : fill-in order is ambiguous
- Ordered Forms : fill-in order matters
- Multi-Button Forms: forms with cancel, back, previous button
- Preview Forms : forms with a preview button

Interacting with lists of items



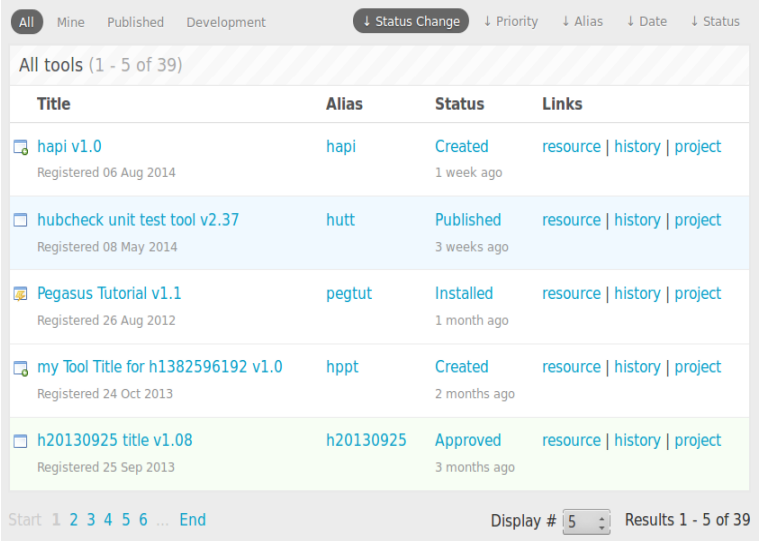






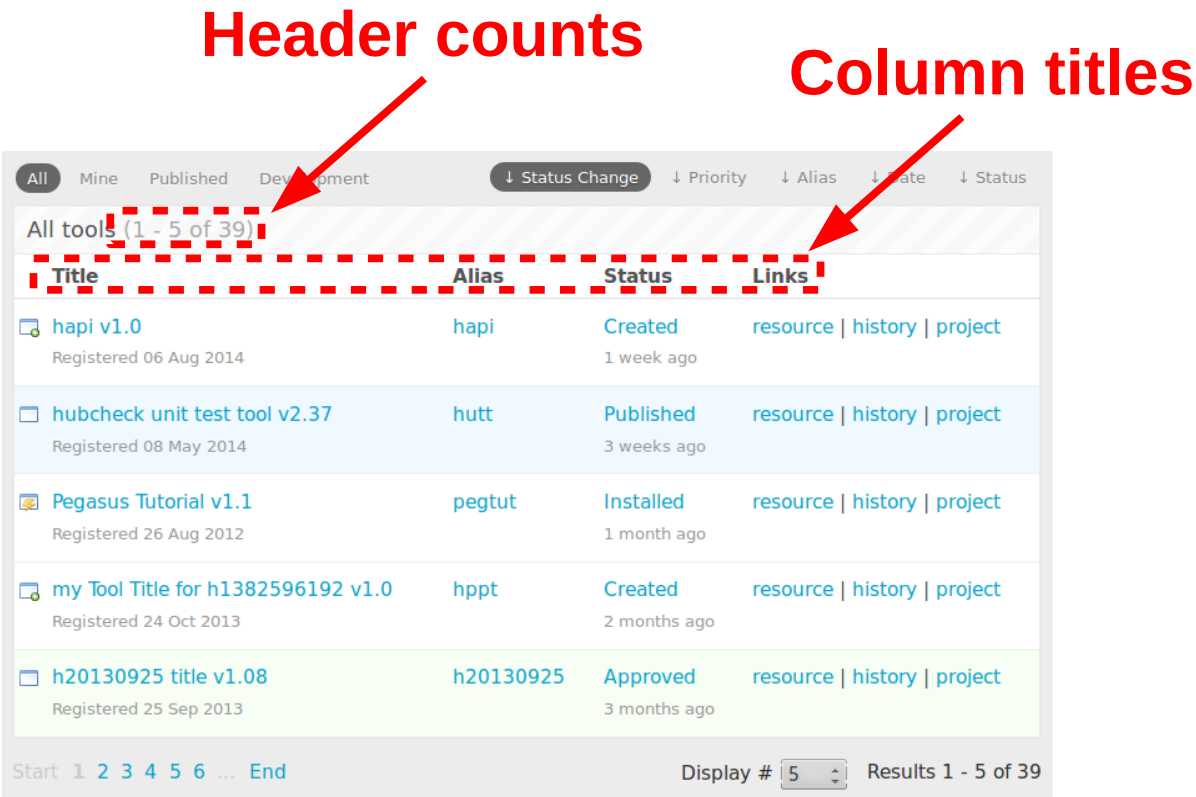
Problems representing lists

1. Can't hard code locators
2. Don't want to pre-allocate items
3. Easy access to items
4. Sequential access to items
5. Searching for items



How are lists used?

1. Interact with list meta-data
2. Present enumerable data
3. Query list item value
4. Interact with list item



How are lists used?

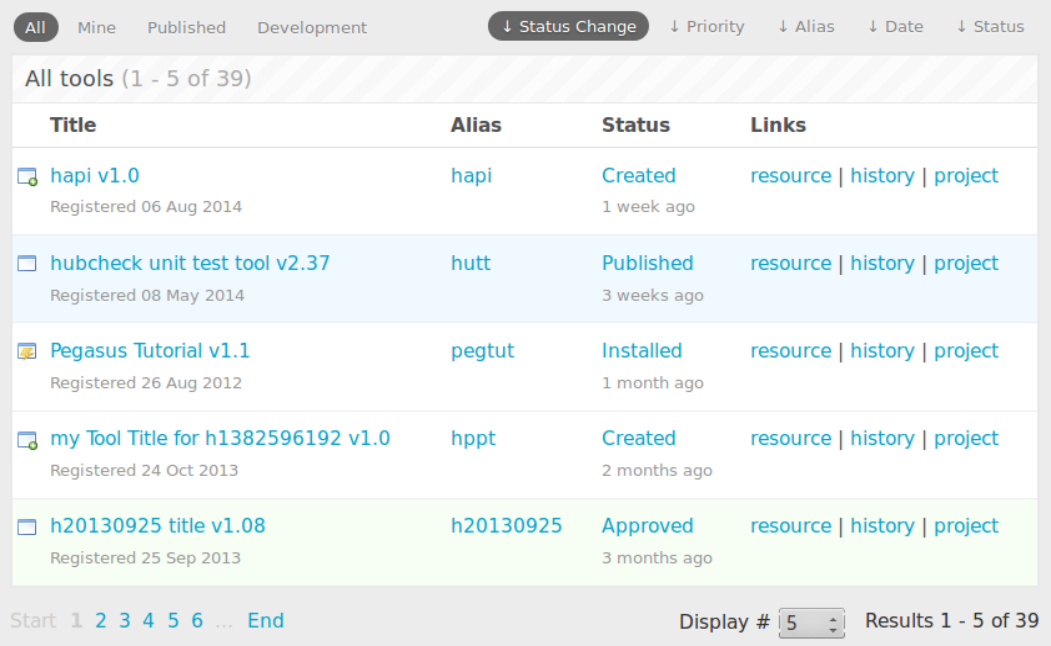
1. Interact with list meta-data
2. Present enumerable data
3. Query list item value
4. Interact with list item

Sequential

1 →

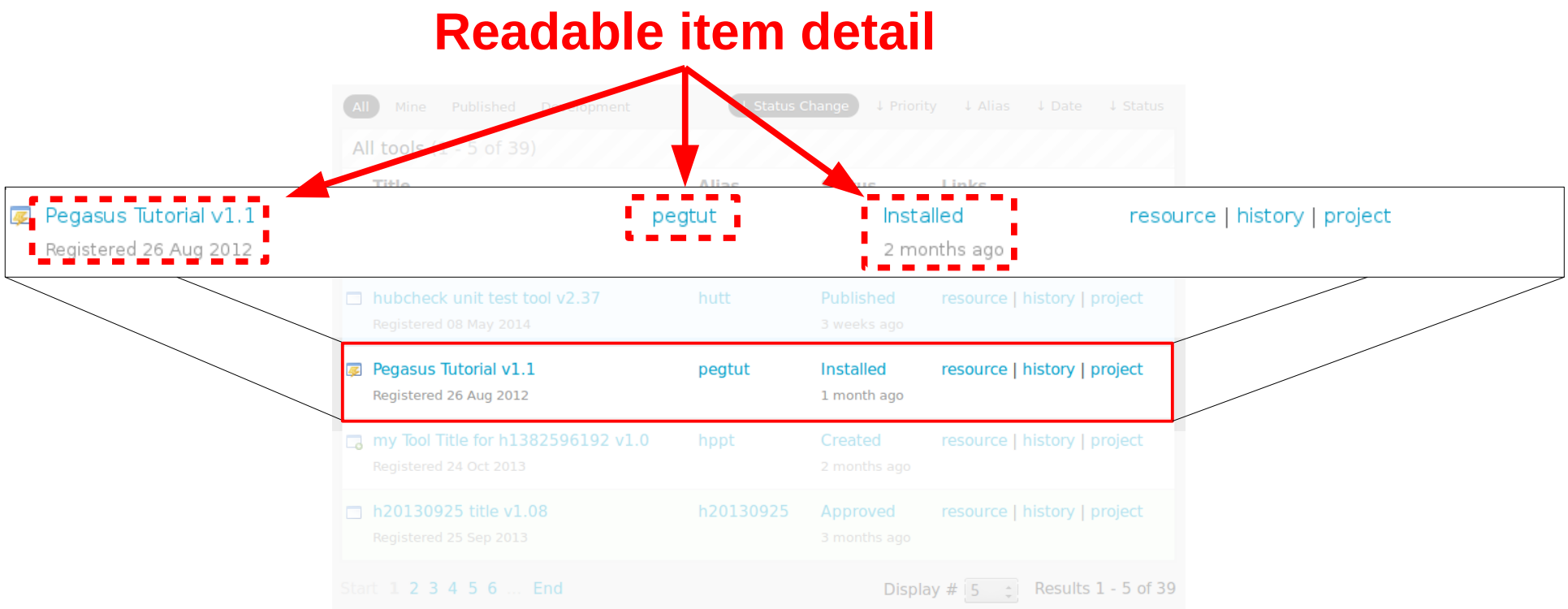
2 →

3 →



How are lists used?

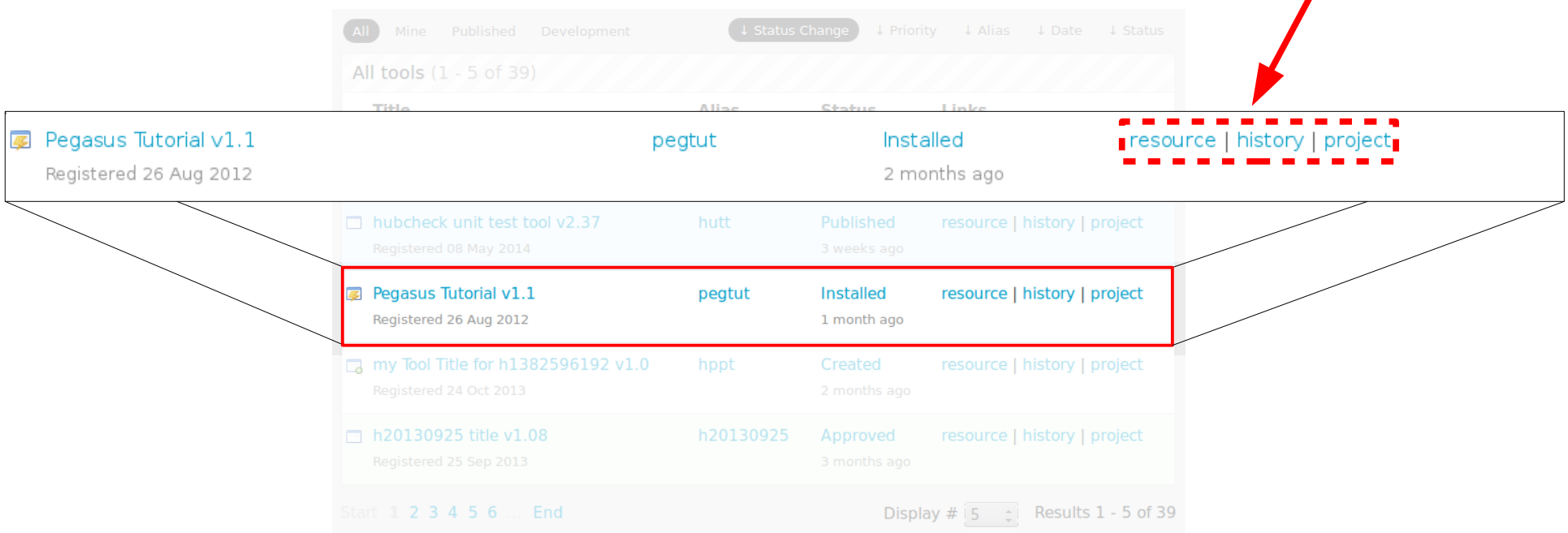
1. Interact with list meta-data
2. Present enumerable data
3. Query list item values
4. Interact with list item



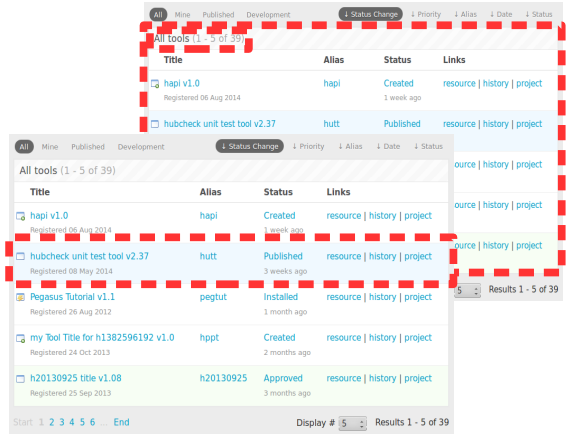
How are lists used?

1. Interact with list meta-data
2. Present enumerable data
3. Query list item value
4. Interact with list item

Links to more detail



Proposed Interface



ItemList Pattern

- Participants:
  - Container class
  - Item class
- Iterator Pattern to enumerate items
- Template Locators solve item count problem.
- Factory Method Pattern to create item objects on demand

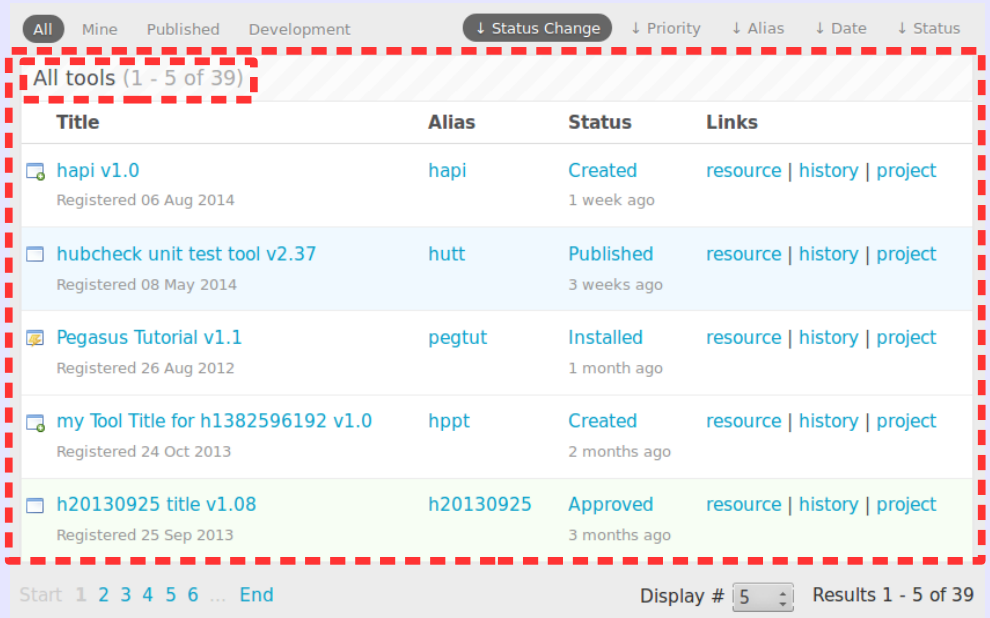
ItemList Pattern Participants

Container Class:

- Provides access to list meta-data
  - num_items()
- Can iterate through items
  - next()
- Search for list items
  - get_item_by_position()
  - get_item_by_property()

Item Class:

- Represents single item in list
- Query value of item
  - value()
- Can reference another item
  - update_item_number()



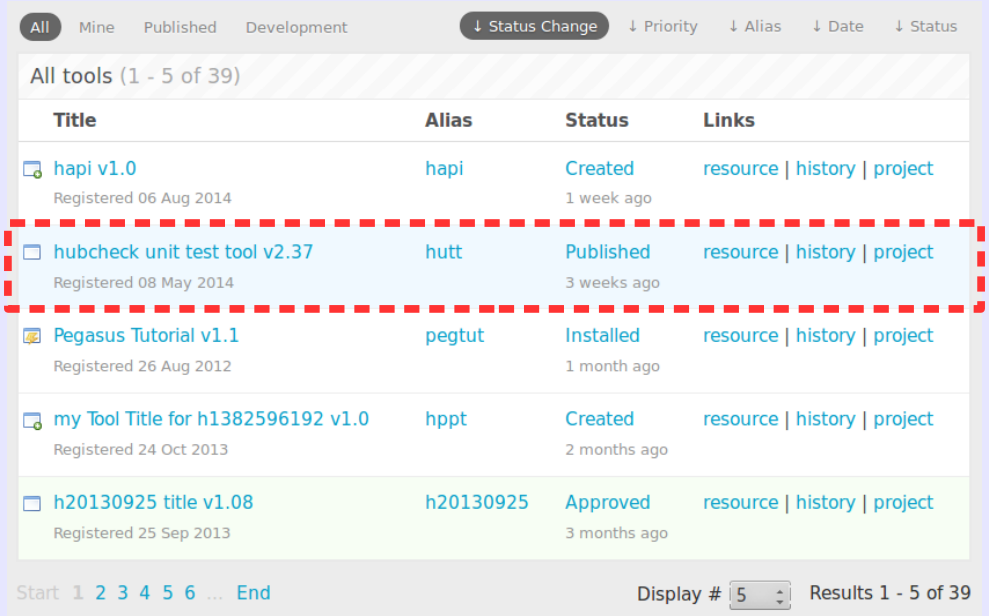
ItemList Pattern Participants

Container Class:

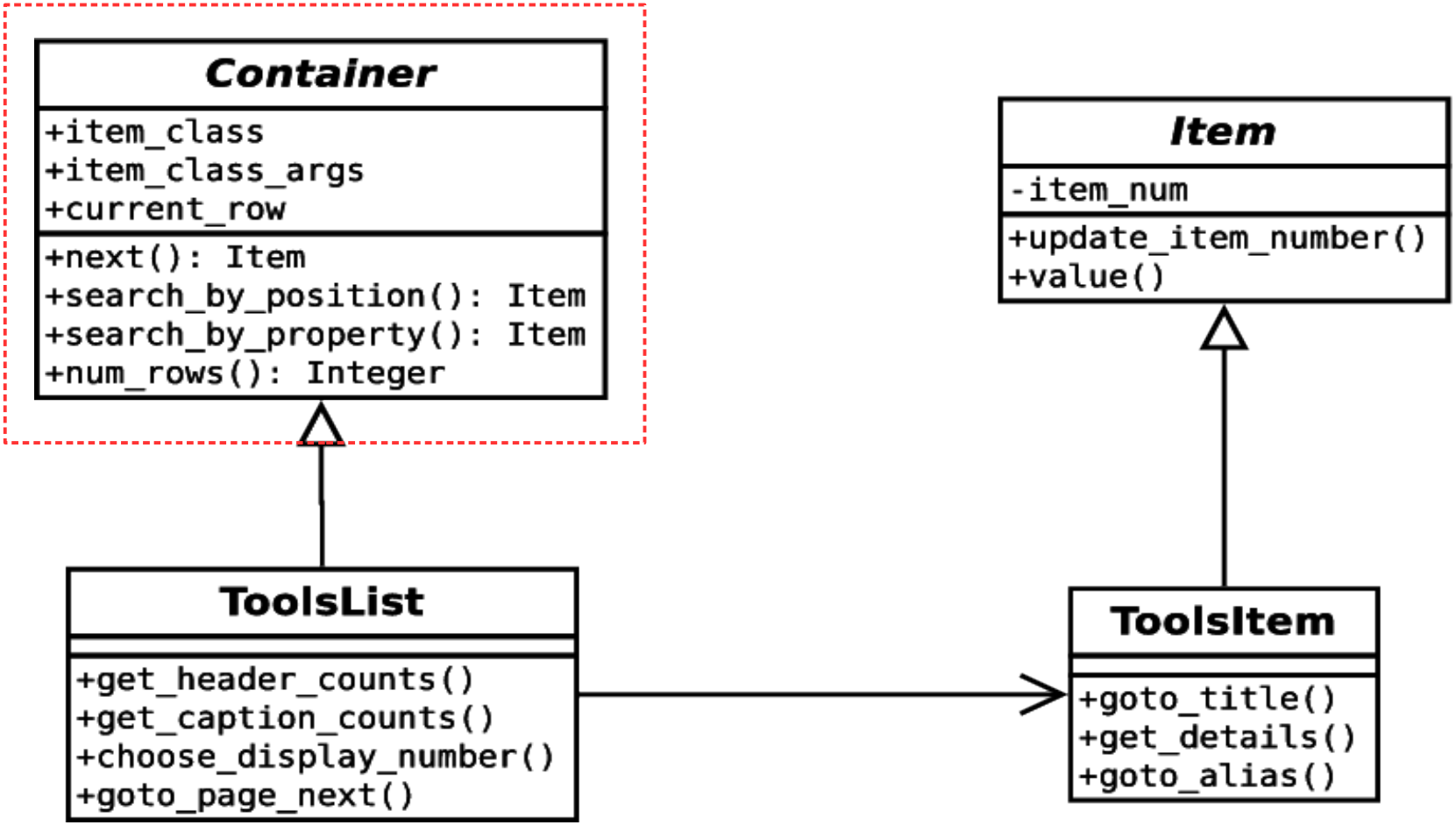
- Provides access to list meta-data
  - num_items()
- Can iterate through items
  - next()
- Search for list items
  - get_item_by_position()
  - get_item_by_property()

Item Class:

- Represents single item in list
- Query value of item
  - value()
- Can reference another item
  - update_item_number()



ItemList Pattern Participants



Container Class Overview

Access list meta-data

```
class Container(object):
    def __init__(self, locatordict, item_class, *args):
        ...
    def num_items(self):
        ...
    def __iter__(self):
        ...
    def next(self):
        ...
    def get_item_by_position(self,item_number):
        ...
    def get_item_by_property(self,prop,val):
        ...
```

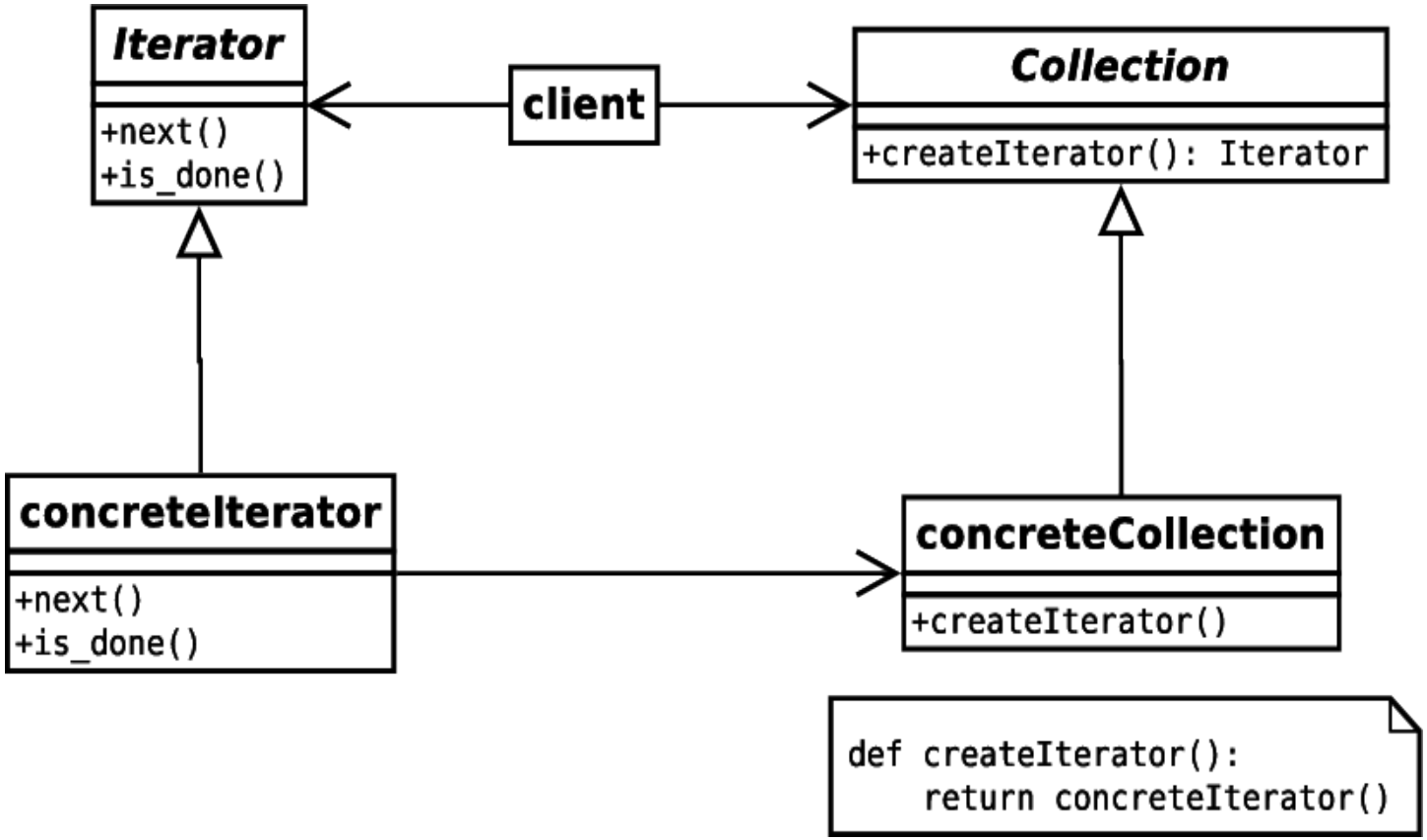
Container Class Overview

```
class Container(object):
    def __init__(self, locatordict, item_class, *args):
        ...
    def num_items(self):
        ...
    def __iter__(self):
        ...
    def next(self):
        ...
    def get_item_by_position(self,item_number):
        ...
    def get_item_by_property(self,prop,val):
        ...
```

Iterate over items

Iterator Pattern

Sequentially access each element in a collection.



Container Class Iterator

```
class Container(object):

    def __iter__(self):
        self.__current_item = 0
        return self

    def next(self):
        ...
        self.__current_item += 1

        if self.__current_item >= self__num_items:
            # reset our counter, stop iterating
            self.__current_item = 0
            raise StopIteration

        return self.get_item_by_position(self.__current_item)
```

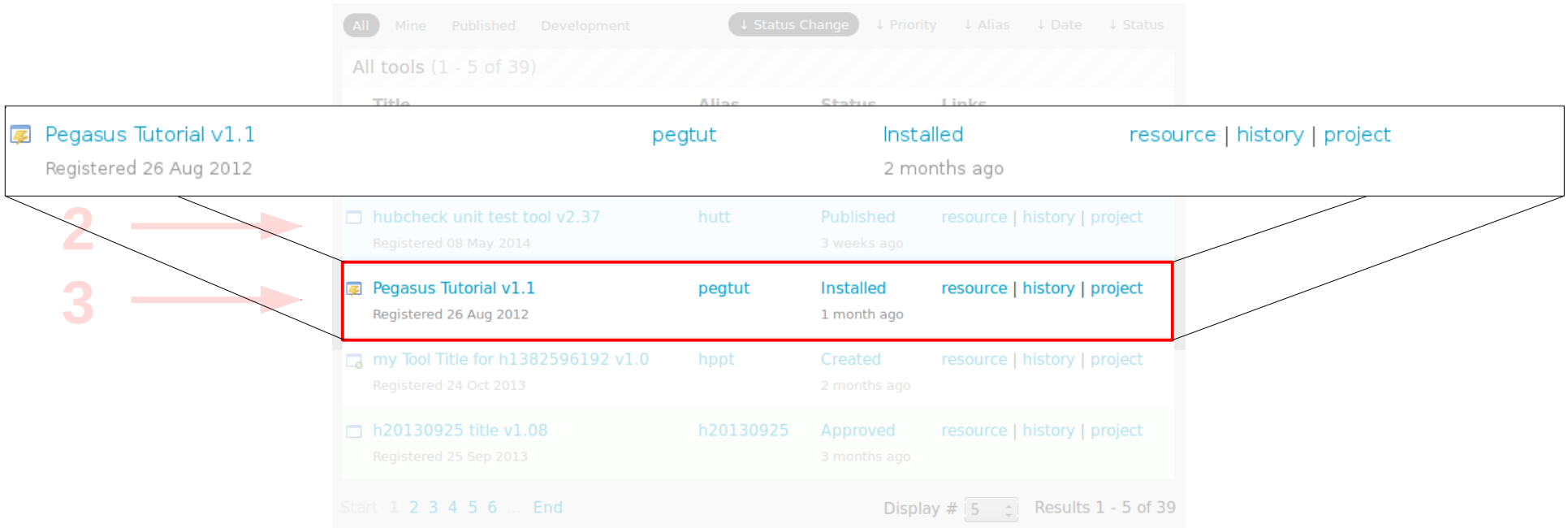
Container Class Overview

```
class Container(object):
    def __init__(self, locatordict, item_class, *args):
        ...
    def num_items(self):
        ...
    def __iter__(self):
        ...
    def next(self):
        ...
    def get_item_by_position(self,item_number):
        ...
    def get_item_by_property(self,prop,val):
        ...
```

Search for list items

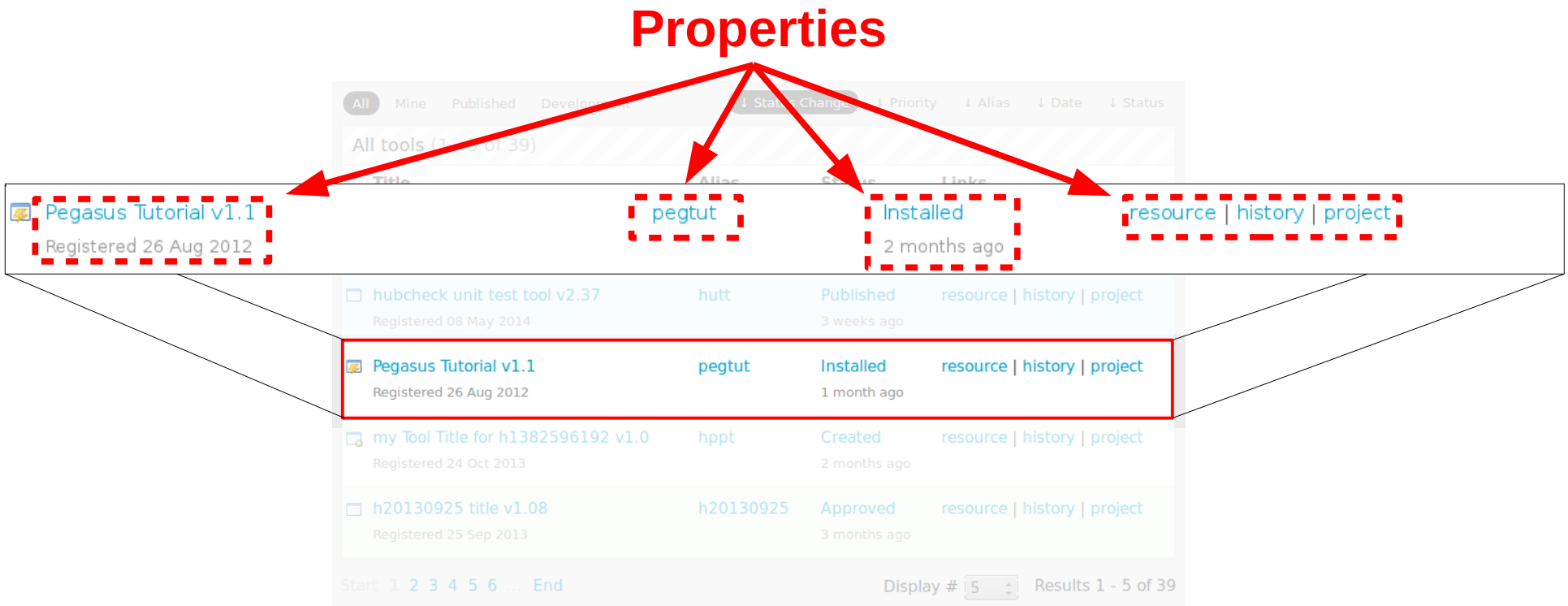
Searching for items: by position

```
def get_item_by_position(self,item_number):
    ...
    return Item(...)
```

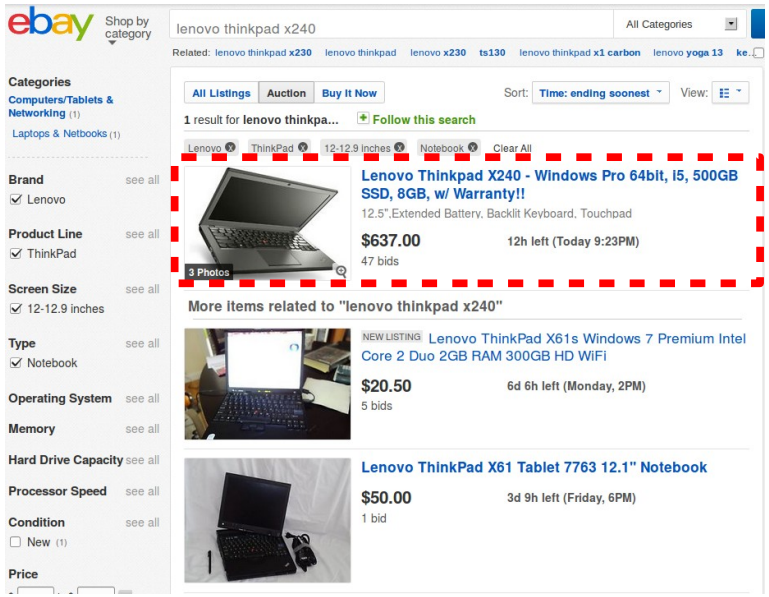


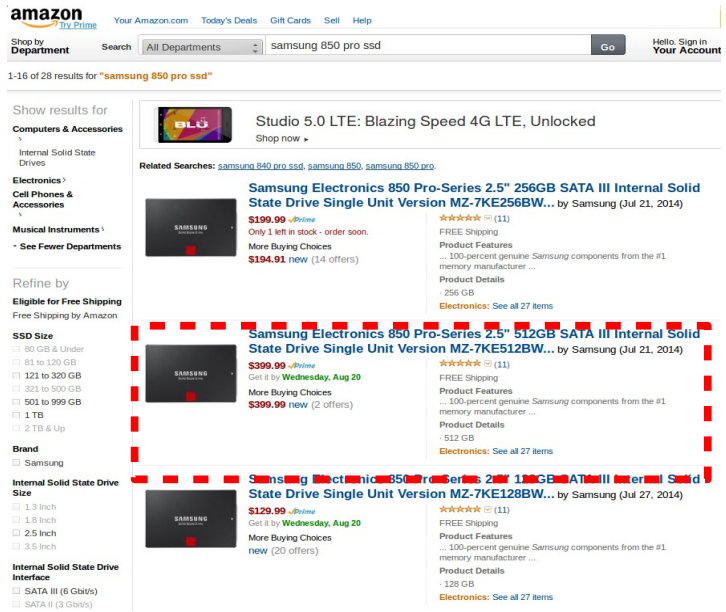
Searching for items: by property

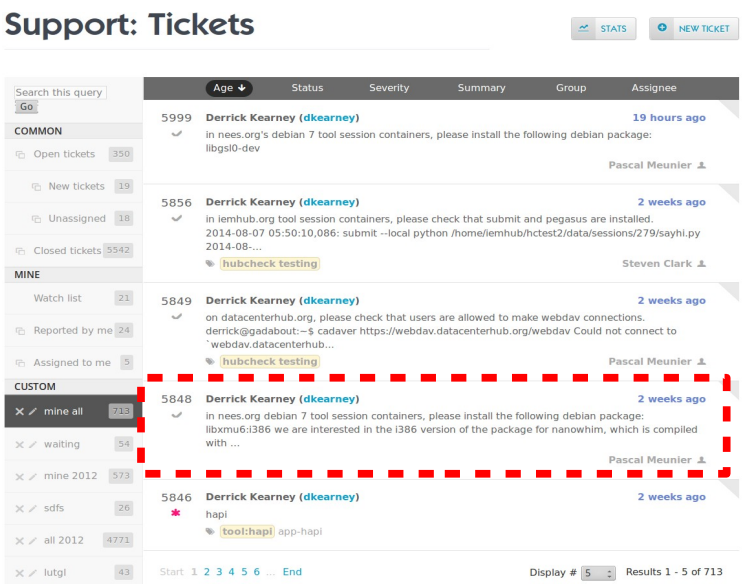
```
def get_item_by_property(self,prop,val):
    ...
    return Item(...)
```

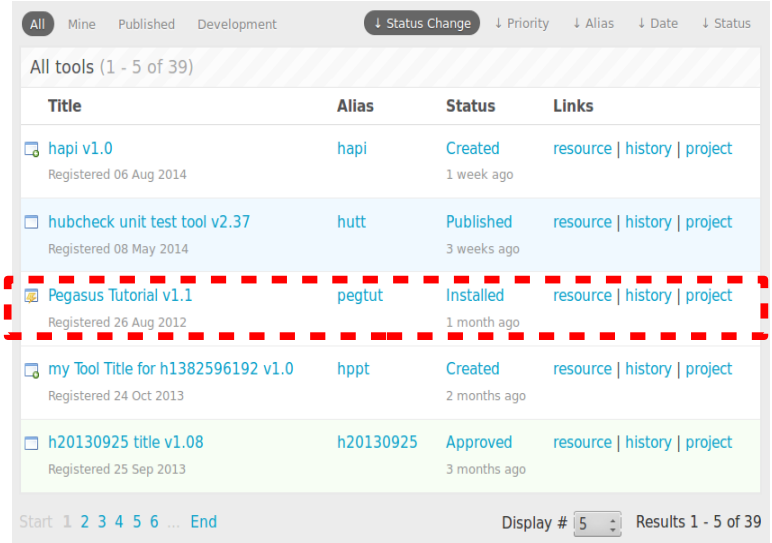


Container Defines Item Class



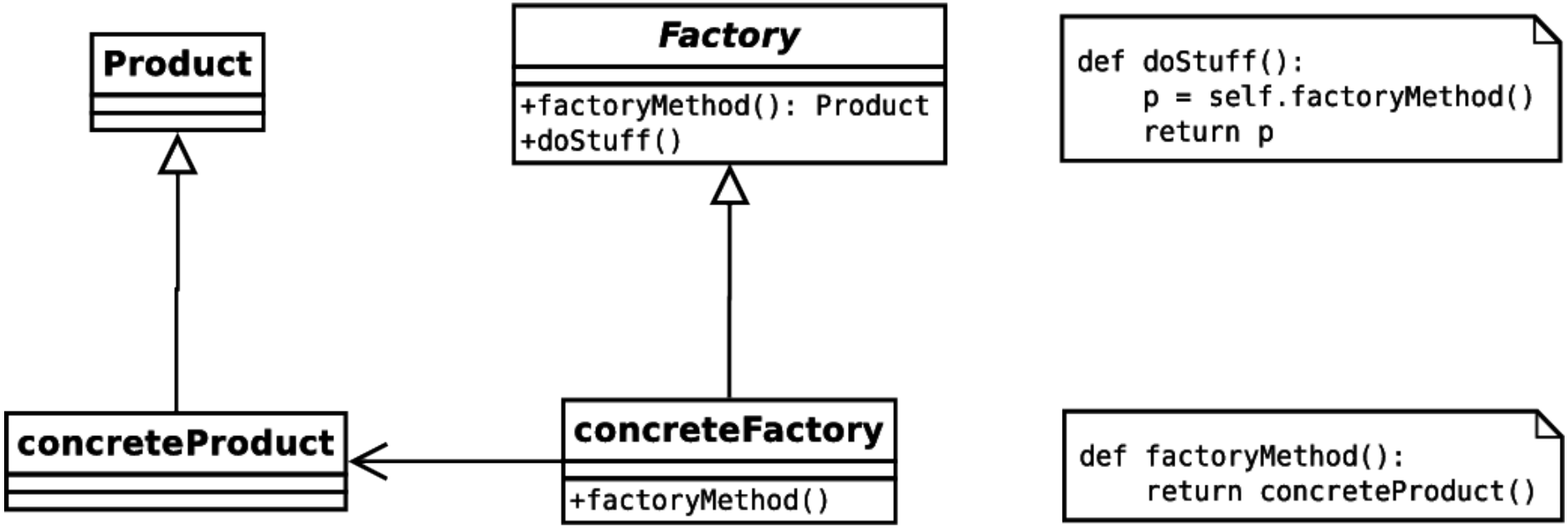






Factory Method Pattern

Define an interface for creating an object, but let subclasses define which class to instantiate.



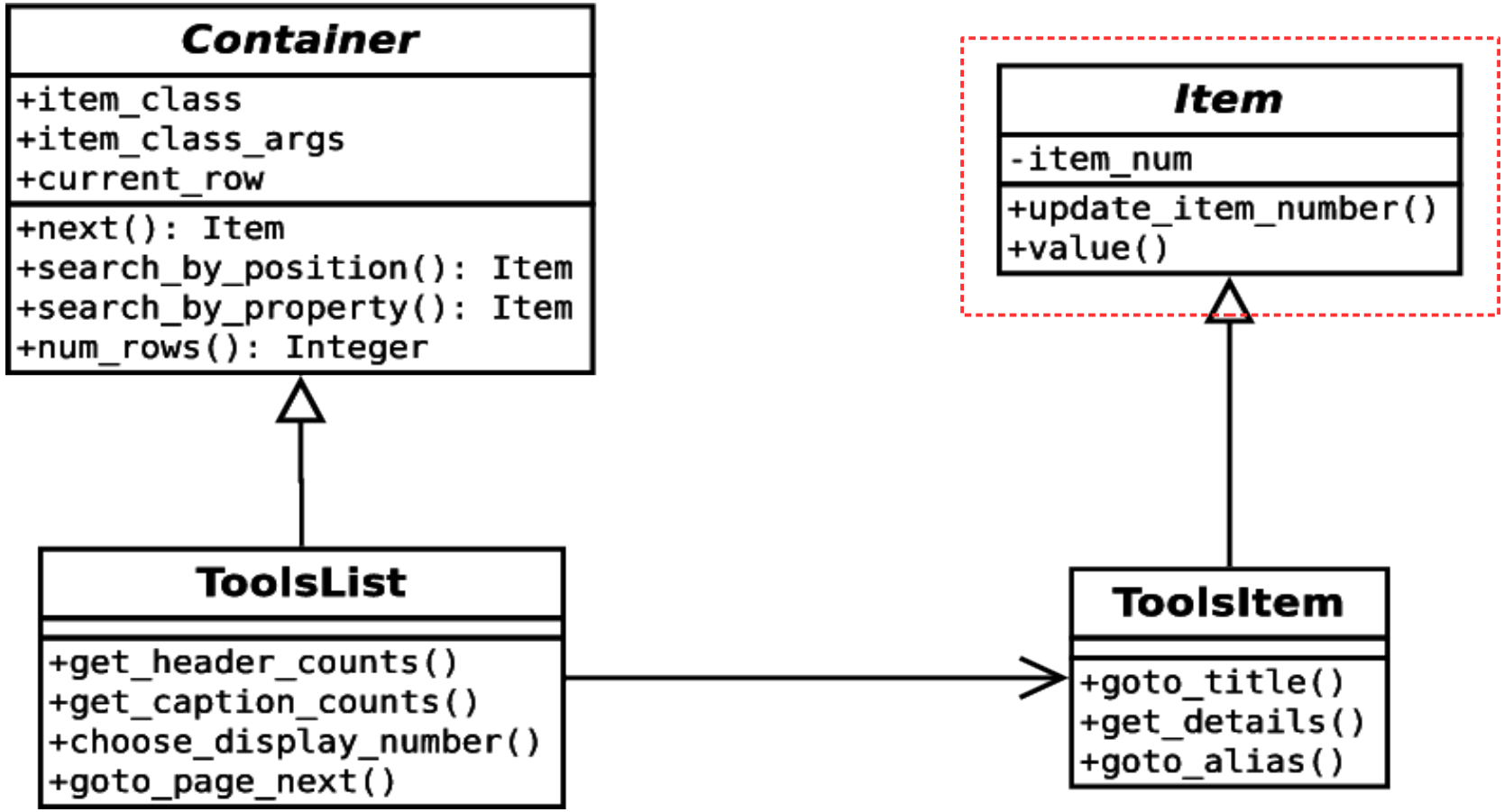
Searching for items

```
class Container(object):
    def __init__(self, locatordict):
        ...
        self.item_class = None
        self.item_class_args = None
        …

    def get_item_by_position(self,item_number):
        ...
        result = self.item_class(
                        *self.item_class_args,
                        item_number=item_number )
        ...
        return result
```

ToolsList implements Container

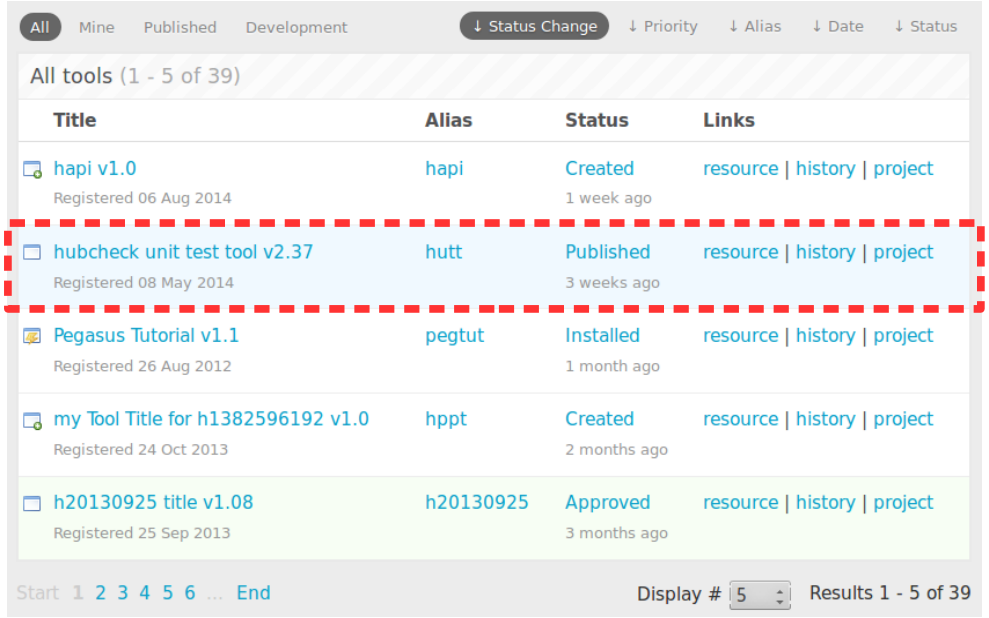
```
class Container(object):
    def __init__(self, locatordict):
        …
        self.item_class = None
        self.item_class_args = None
        …

class ToolsList(Container):
    def __init__(self, locatordict):
        ...
        self.item_class = ToolsItem
        self.item_class_args = [{...}]


class ToolsItem(Item):
    def __init__(self, locator_templates):
        ...
```

ItemList Pattern Participants



Item Class Overview

```
class Item (object):

    def __init__ (self, locatordict, item_number):
        …

    def update_item_number (self, item_number):
        …

    def value (self):
        …
```



Items represent a single item

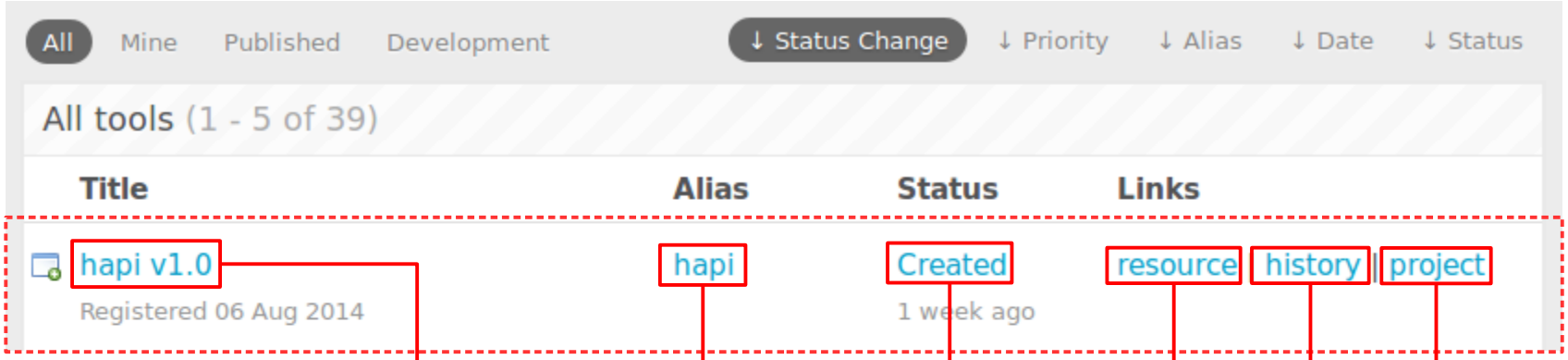
```
class Item (object):

    def __init__ (self, locatordict, item_number):
        self.locators = locatorsdict
        self.__item_number = item_number
        …
```

Items can be updated

```
class Item (object):

    def __init__ (self, locatordict, item_number):
        self.locators = locatorsdict
        self.__item_number = item_number
        …

    def update_item_number (self, item_number):
        …
```

```
locators = {
    'title'        : "tr:nth-of-type(1) .title",
    'details'      : "tr:nth-of-type(1) .details",
    'alias'        : "tr:nth-of-type(1) .alias",
    'status'       : "tr:nth-of-type(1) .status",
    'time'         : "tr:nth-of-type(1) .time",
    'resource'     : "tr:nth-of-type(1) .page",
    'history'      : "tr:nth-of-type(1) .history",
    'wiki'         : "tr:nth-of-type(1) .wiki",
}
```

Interacting with ItemLists



tr:nth-of-type(1) .title

tr:nth-of-type(1) .alias

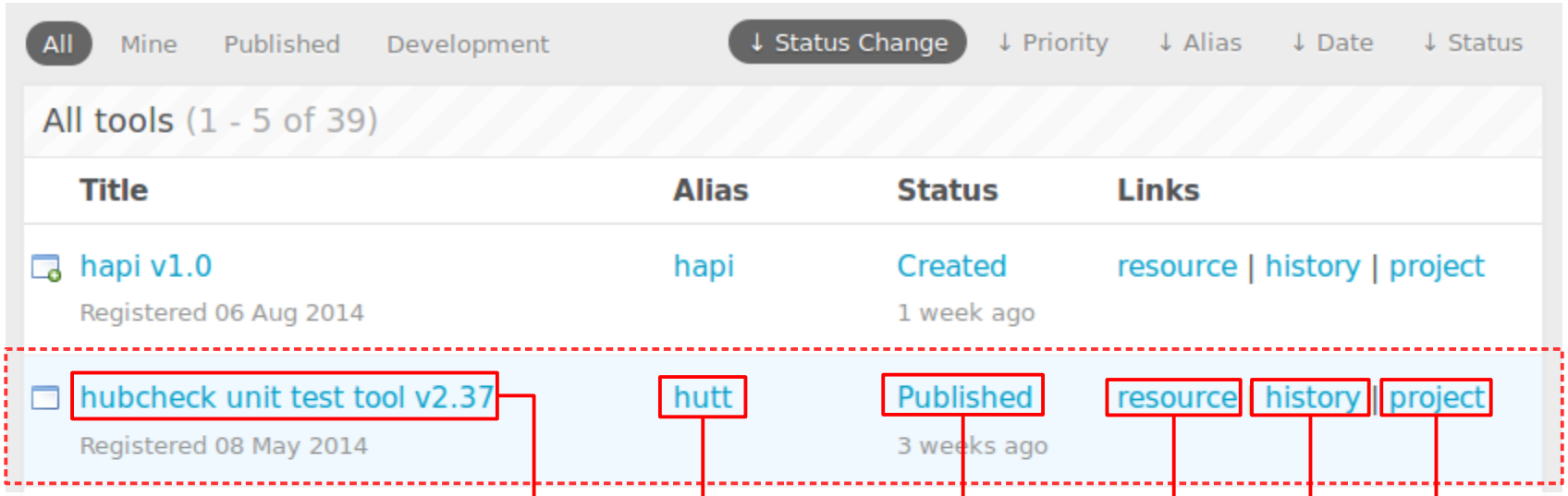
tr:nth-of-type(1) .status

tr:nth-of-type(1) .page

tr:nth-of-type(1) .history

tr:nth-of-type(1) .wiki

Interacting with ItemLists



tr:nth-of-type(2) .title

tr:nth-of-type(2) .alias

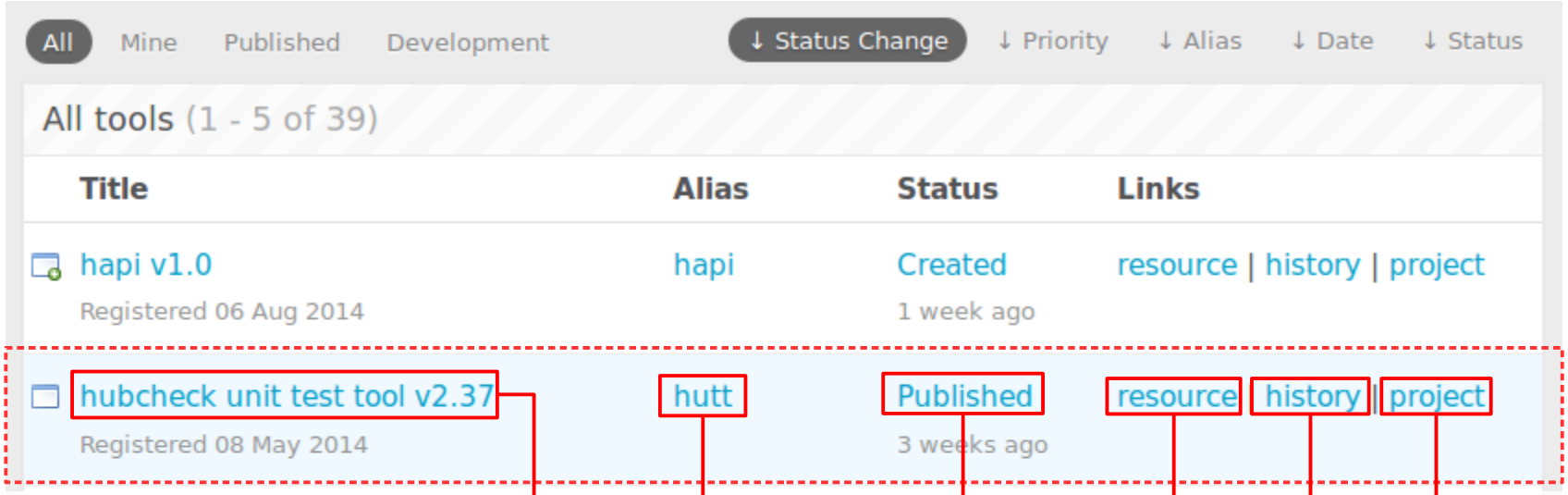
tr:nth-of-type(2) .status

tr:nth-of-type(2) .page

tr:nth-of-type(2) .history

tr:nth-of-type(2) .wiki

Interacting with ItemLists



tr:nth-of-type(n) .title

tr:nth-of-type(n) .alias

tr:nth-of-type(n) .status

tr:nth-of-type(n) .page

tr:nth-of-type(n) .history

tr:nth-of-type(n) .wiki

Template Locators

```
>>> locator = "tr:nth-of-type({item_num}) .title"
>>>
>>> locator.format(item_num=3)
```

tr:nth-of-type(3) .title

Items can be updated

```
class Item (object):

    def __init__ (self, locatordict, item_number):
        self.locators = locatordict
        self.__item_number = item_number
        …

    def update_item_number (self, item_number):
        …
```

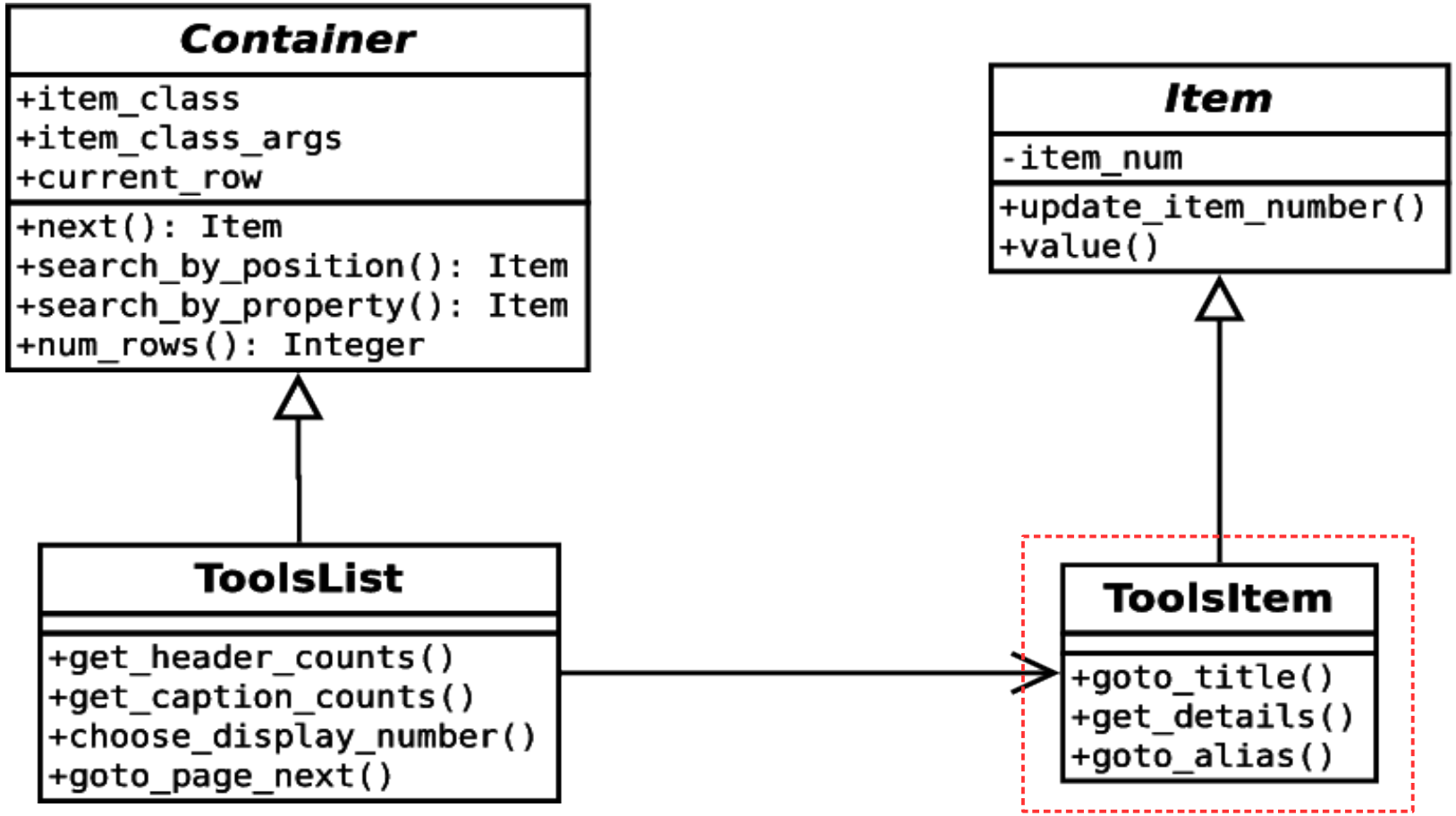
```
locators = {
    'title'        : "tr:nth-of-type(1) .title",
    'details'      : "tr:nth-of-type(1) .details",
    'alias'        : "tr:nth-of-type(1) .alias",
    'status'       : "tr:nth-of-type(1) .status",
    'time'         : "tr:nth-of-type(1) .time",
    'resource'     : "tr:nth-of-type(1) .page",
    'history'      : "tr:nth-of-type(1) .history",
    'wiki'         : "tr:nth-of-type(1) .wiki",
}
```

Items can be updated

```
class Item (object):

    def __init__ (self, locator_templates, item_number):
        self.locators = {}
        self.locator_templates = locator_templates
        self.__item_number = item_number

        self.update_item_number(item_number)
        …

    def update_item_number (self, item_number):
        self.__item_number = item_number
        for k,v in self.locators_templates.items():
            self.locators[k] = v.format(item_num=self.__item_number)
        …
```

```
locators = {
    'title'        : "tr:nth-of-type({item_num}) .title",
    'details'      : "tr:nth-of-type({item_num}) .details",
    'alias'        : "tr:nth-of-type({item_num}) .alias",
    ...
}
```

ItemList Pattern Participants



Items have a value

```
class ToolsItem (Item):

    def __init__ (self, locator_templates, item_number):
        ...
        self.title = Link(...)
        self.details = Text(...)
        self.alias = Link(...)
        ...

    def value(self):
        properties = {
            'title'     : self.title.text(),
            'details' : self.details.value,
            'alias'     : self.alias.text(),
            ...
        }
        return properties

    ...
```

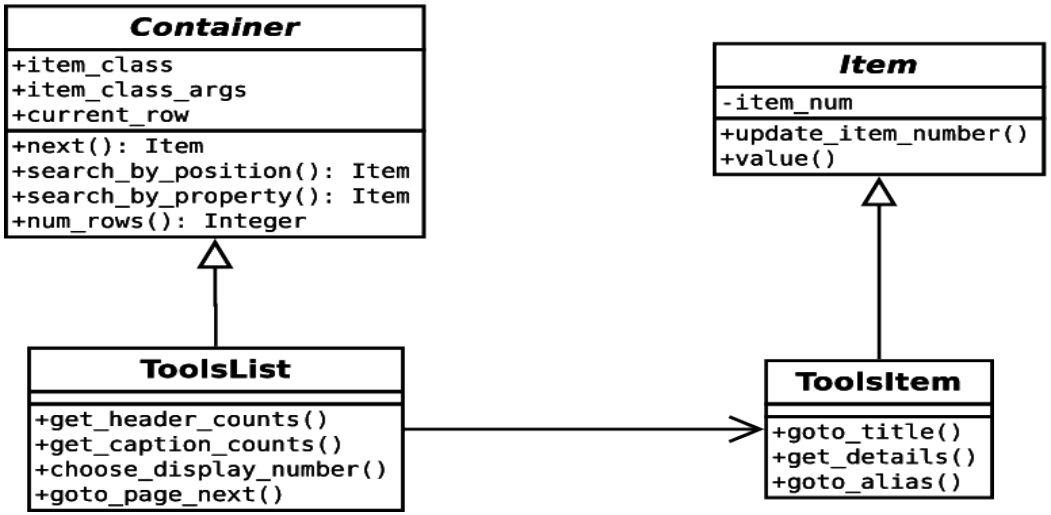
value() method returns a dictionary of properties from the item.

Items can be interacted with

```
class ToolsItem (object):
    def __init__ (self, locator_templates, item_number):
        ...
        self.title = Link(...)
        self.details = Text(...)
        self.alias = Link(...)
        ...

    def goto_title(self):
        self.title.click()

    def get_details(self):
        return self.details.value

    def goto_alias(self):
        self.alias.click()

    ...
```

Web page services are implemented as methods of the page object.

ItemList Pattern



Problem

1. Can't hard code locators
2. Don't want to pre-allocate
3. Easy access to items
4. Sequential access to items
5. Searching for items

Solution

1. Template Locators
2. Factory Method Pattern
3. Item objects are updatable
4. Iterator Pattern
5. Item objects have a value

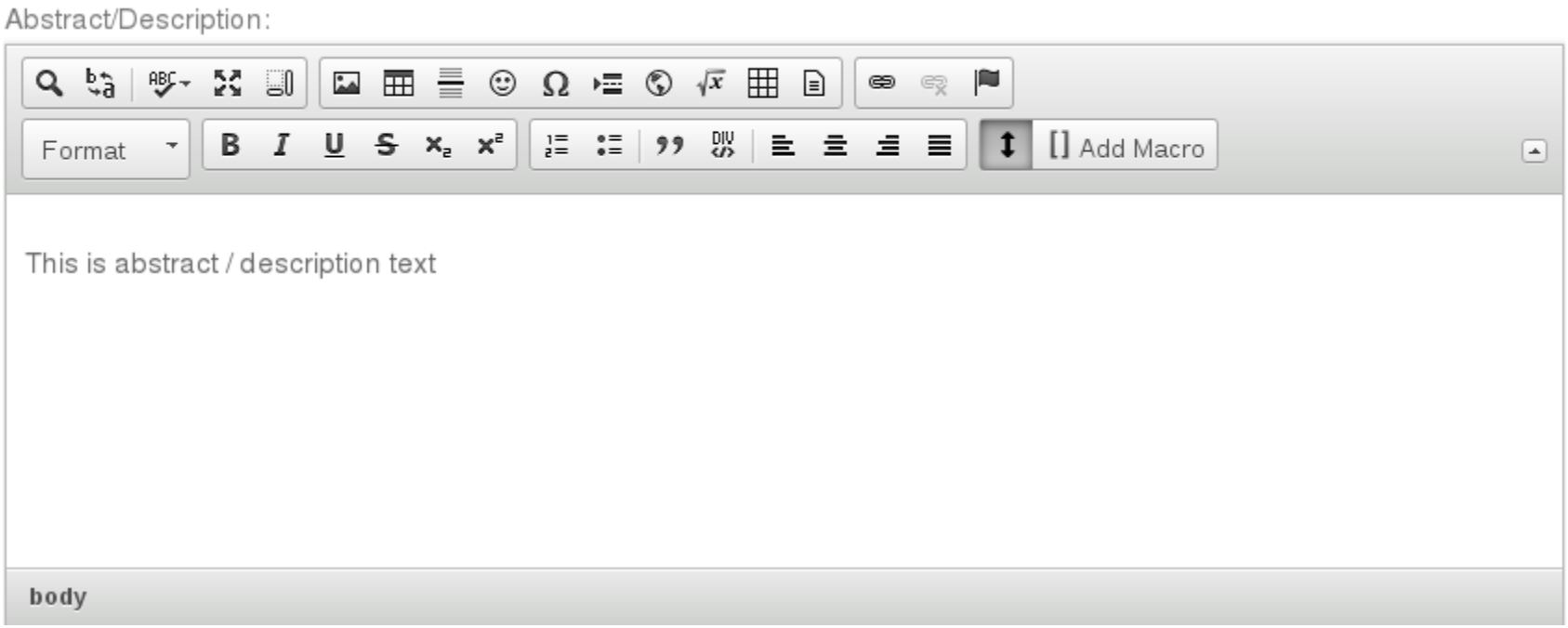
Interacting with Iframes

Abstract/Description:

This is abstract / description text

```
<label for="field-fulltxt">
    Abstract/Description:
    <textarea id="field-fulltxt">This is abstract / description text</textarea>
</label>
```

Interacting with Iframes

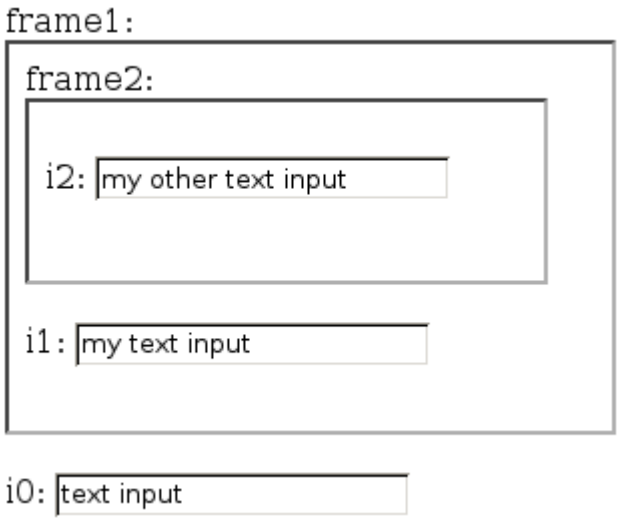



```
<iframe class="cke_wysiwg_frame">
    <html>
        <body class="ckeditor-body">
            <p>This is abstract / description text </p>
        </body>
    </html>
</iframe>
```

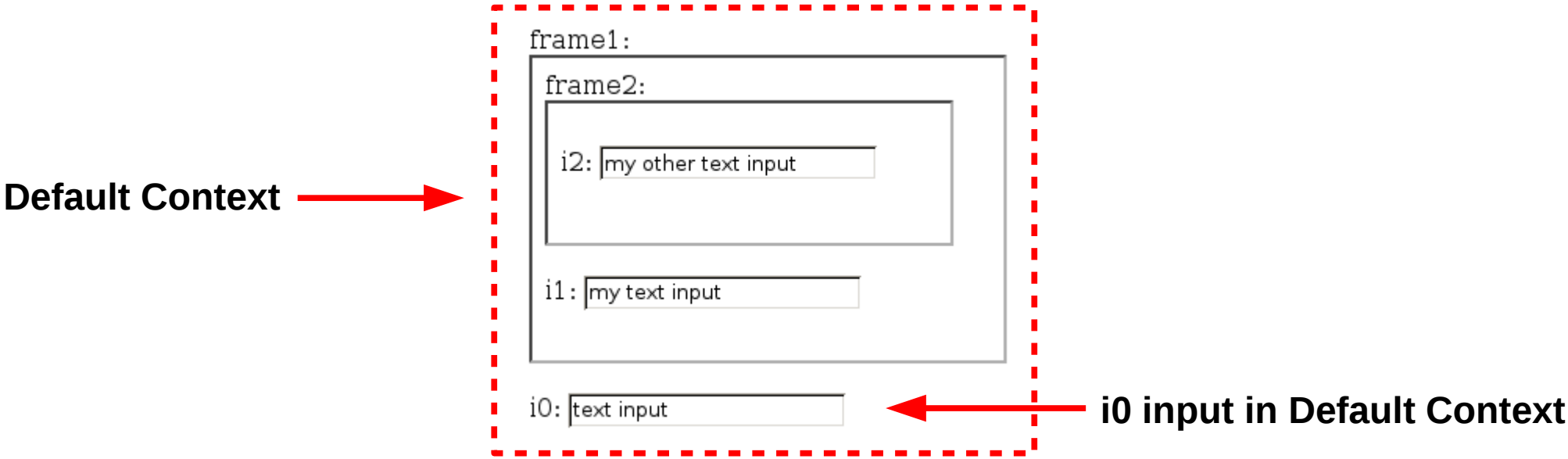
Question:

Do I need to write a new page object for a widget embedded in an <iframe>, if I already have a class that works except for entering and exiting the <iframe>?

How do Iframes work

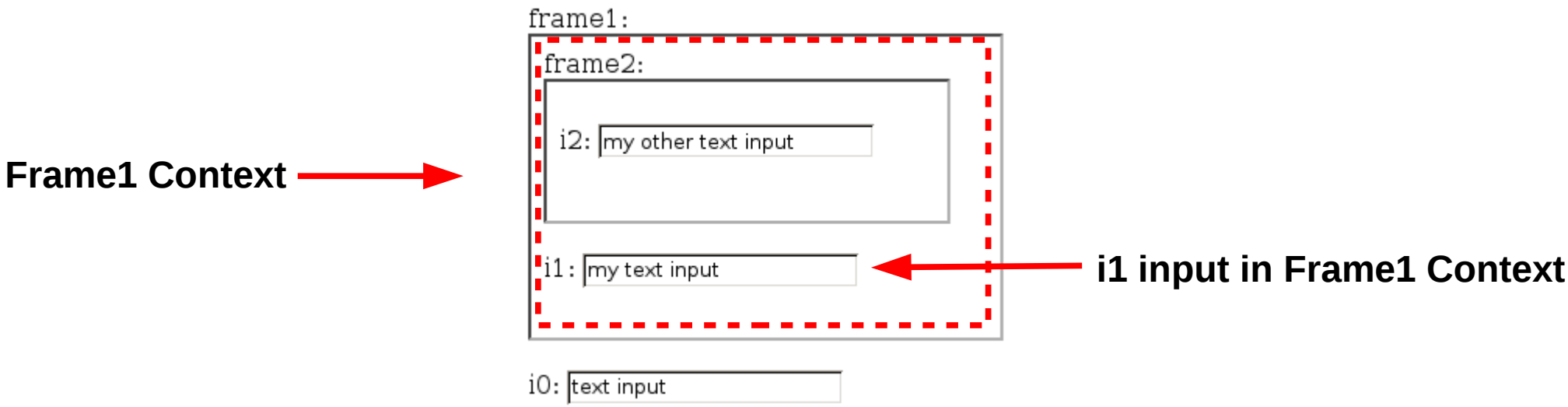


How do Iframes work



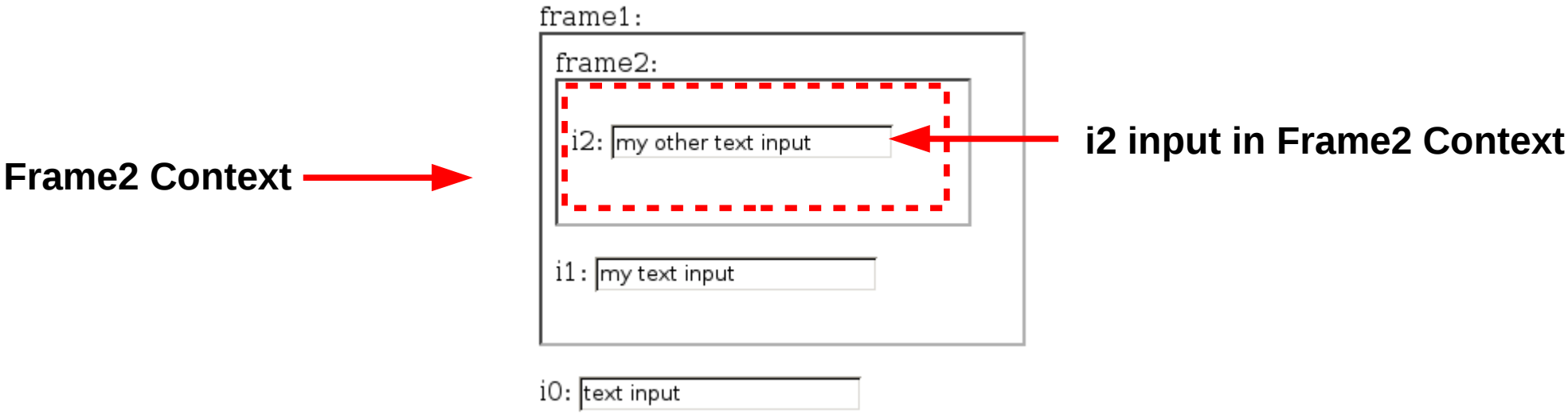

```
<html>
  <body>
    <label for="frame1">frame1: </label>
    <iframe id="frame1" src="inner_page.html"></iframe>
    <br/><br/>
    <label for="i0">i0: </label>
    <input type="text" id="i0" value="text input"></input>
  </body>
</html>
```

How do Iframes work




```
<html>
  <body>
    <label for="frame2">frame2: </label>
    <iframe id="frame2" src="another_page.html"></iframe>
    <br/><br/>
    <label for="i1">i1: </label>
    <input type="text" id="i1" value="my text input"></input>
  </body>
</html>
```

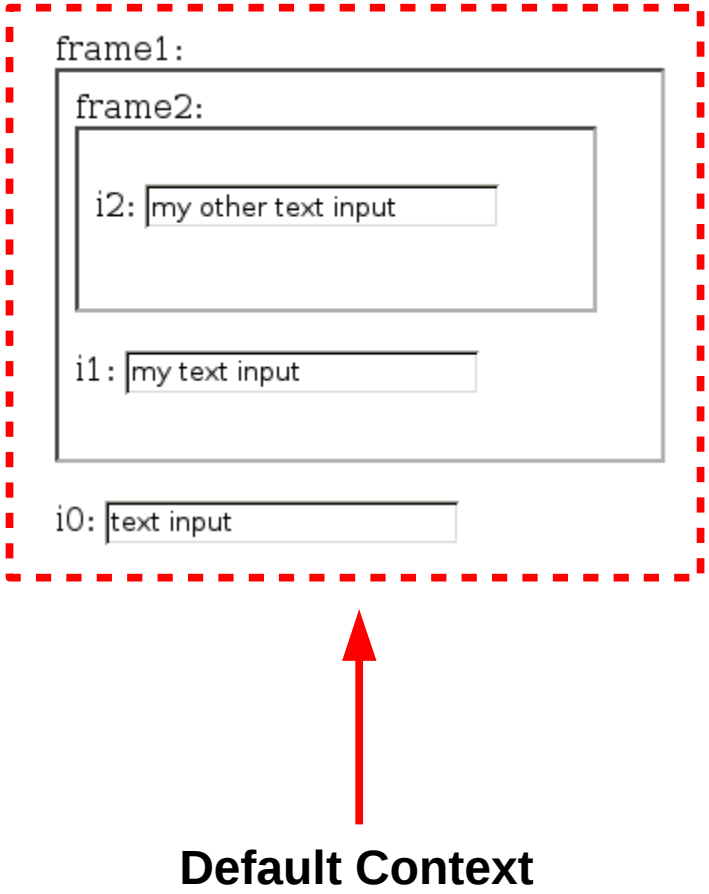
How do Iframes work




```
<html>
   <body>
      <label for="i2">i2: </label>
      <input type="text" id="i2" value="my other text input"></input>
   </body>
</html>
```

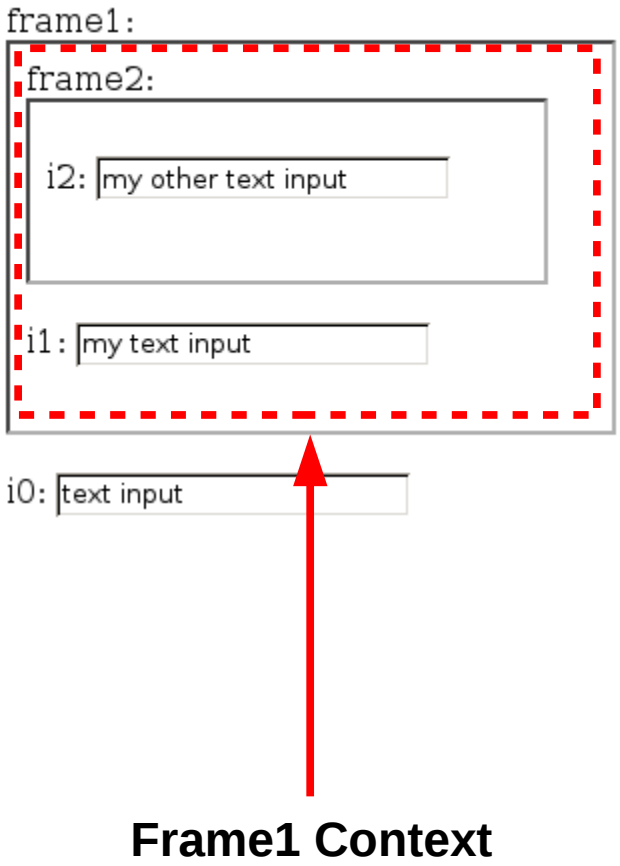
Page Object for i0

```
class Text(BasePageWidget):
    def __init__(self, locator):
        ...

    # getter
    def value(self):
        e = self.find_element(self.locator)
        return e.get_attribute('value')

    # setter
    def value(self, text):
        e = self.find_element(self.locator)
        e.clear()
        e.send_keys(text)

    def append(self, text):
        e = self.find_element(self.locator)
        e.send_keys(text)
```



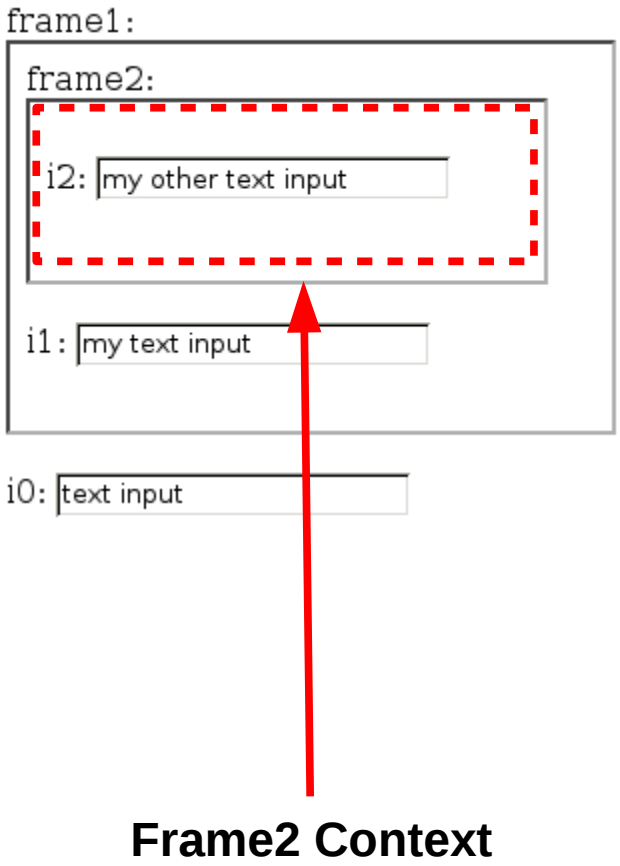
Page Object for i1

```
class Text1Frame(BasePageWidget):
    def __init__(self, locator):
        ...

    # getter
    def value(self):
        ...

    # setter
    def value(self, text):
        frame = self.find_element('#frame1')
        self._browser.switch_to_frame(frame)
        e = self.find_element(self.locator)
        e.clear()
        e.send_keys(text)
        self._browser.switch_to_default_content()

    def append(self, text):
        ...
```



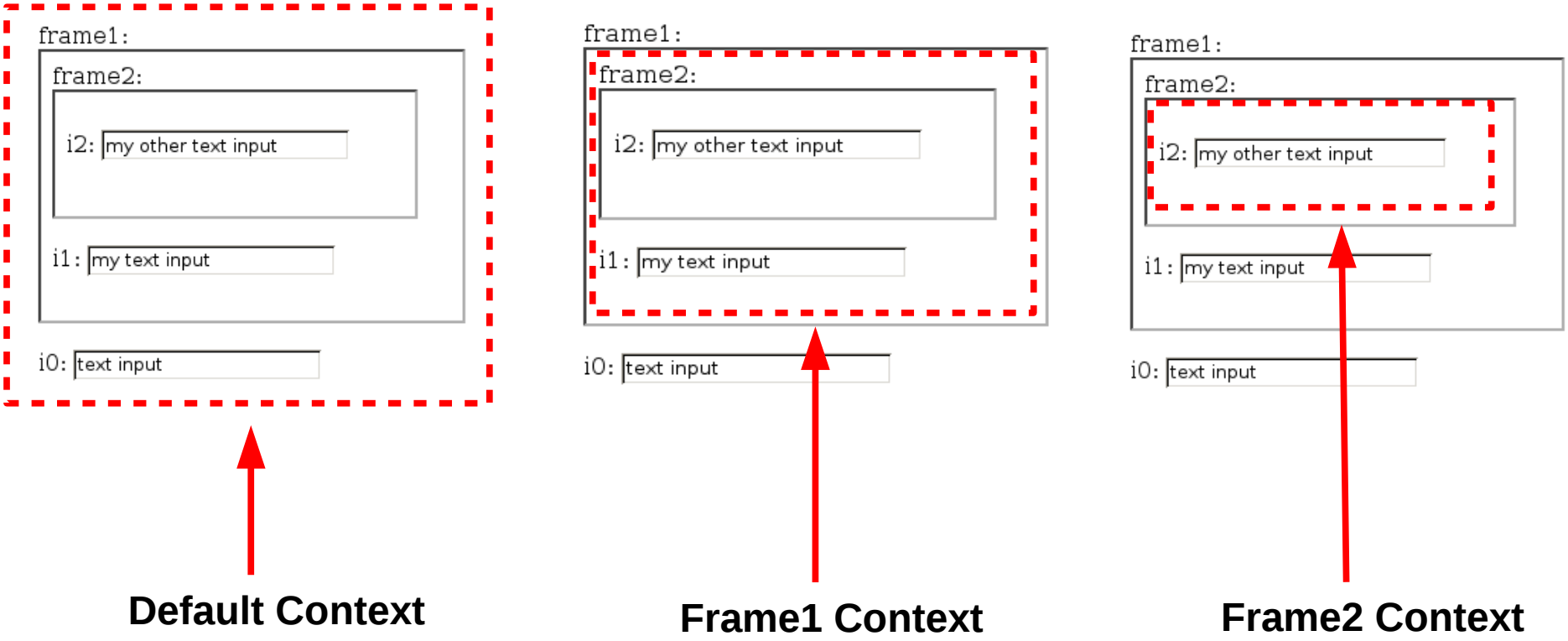
Page Object for i2

```
class Text2Frame(BasePageWidget):
    def __init__(self, locator):
        ...

    # getter
    def value(self):
        ...

    # setter
    def value(self, text):
        frame1 = self.find_element('#frame1')
        self._browser.switch_to_frame(frame1)
        frame2 = self.find_element('#frame2')
        self._browser.switch_to_frame(frame2)
        e = self.find_element(self.locator)
        e.clear()
        e.send_keys(text)
        self._browser.switch_to_default_content()

    def append(self, text):
        ...
```



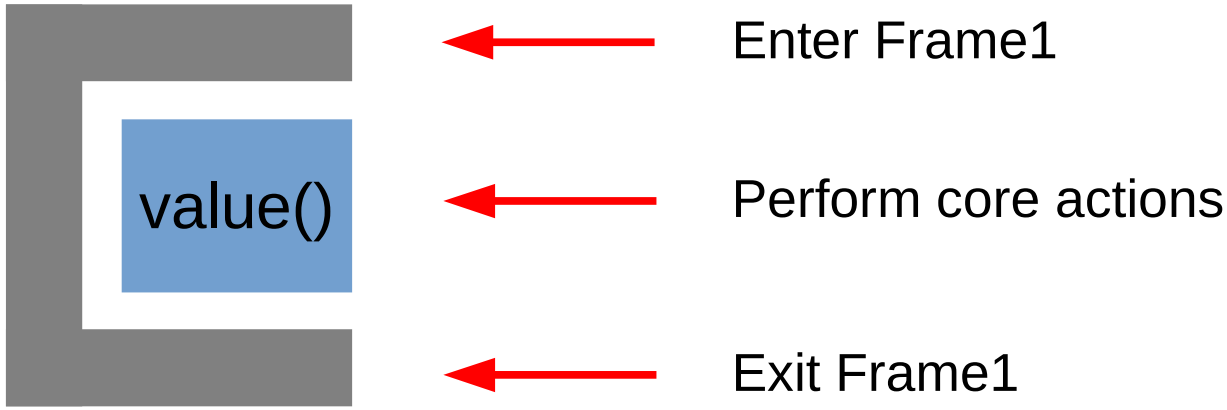
Dealing with Iframes

- Enter / Exit frames to interact with elements
- Must update all methods of page object.
- Create new page objects for Iframes?!?

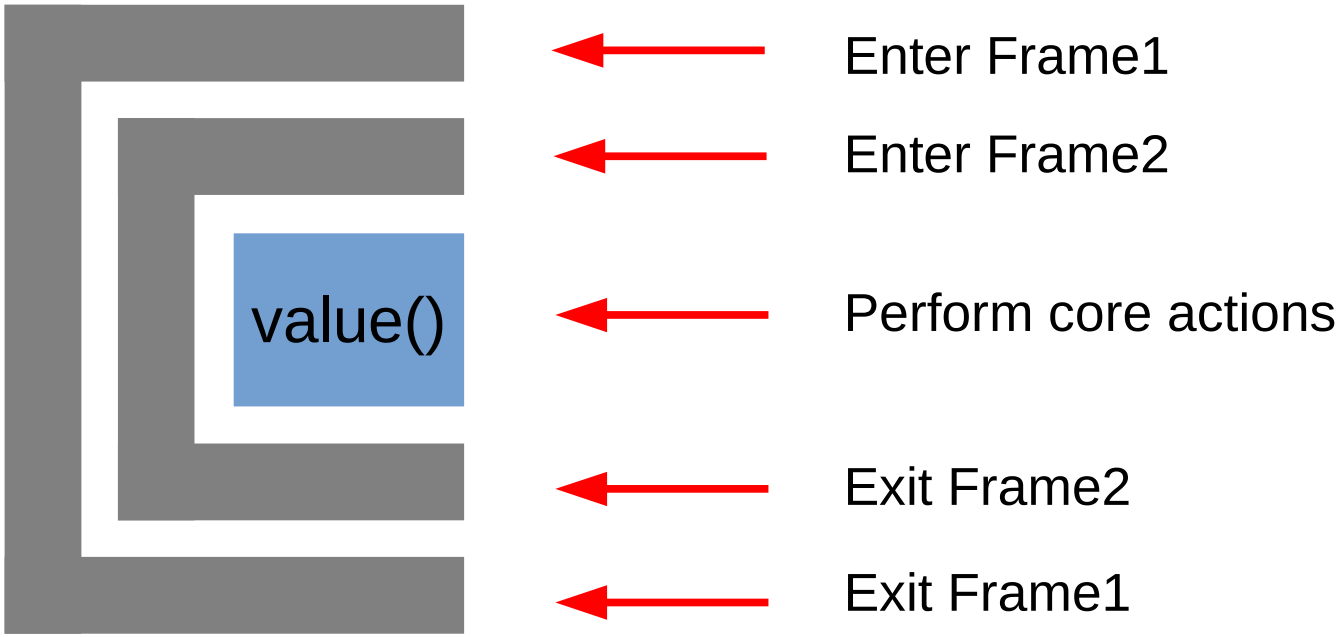


Page object's value: No Iframes

Page object's value: One Iframe

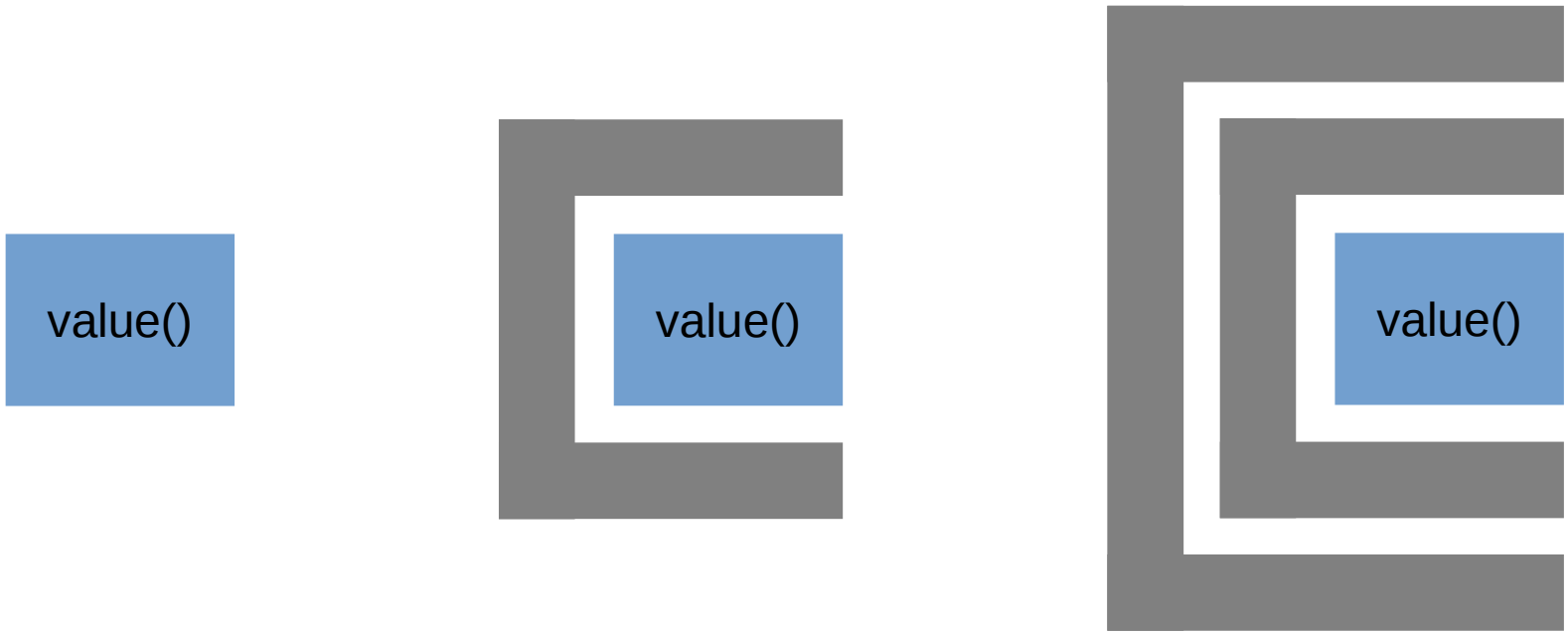


Page object's value: Two Iframes



IframeWrap Design Pattern

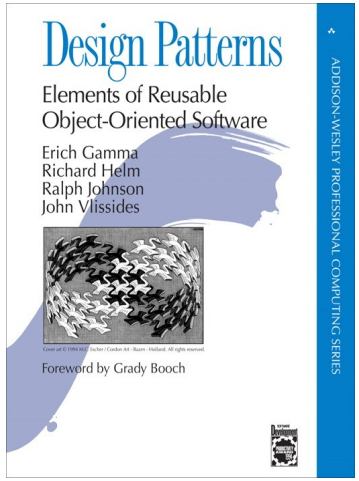
Use Decorator Pattern to wrap attributes of a page object with Enter / Exit iframe calls.



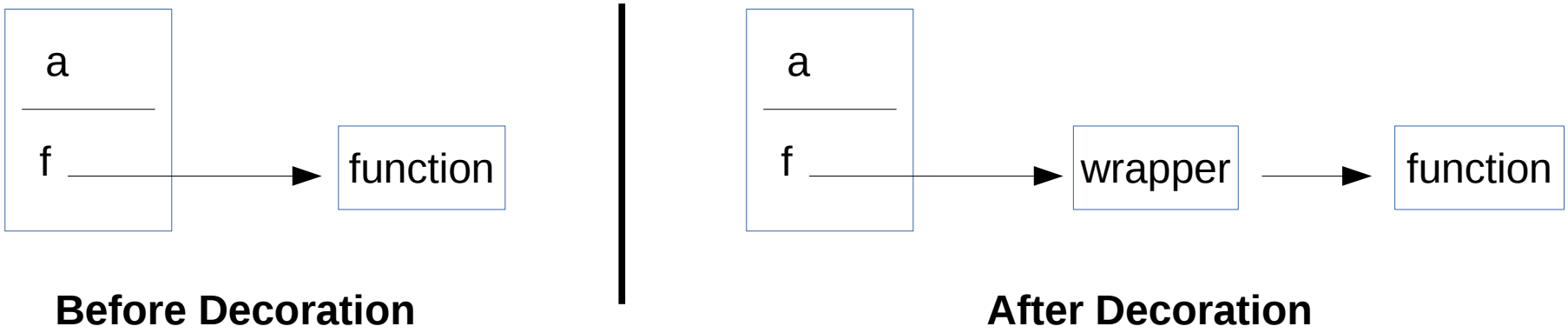
0 Iframes **1 Iframe** **2 Iframes**

Decorator Pattern

Attach additional responsibilities to an object dynamically and transparently.

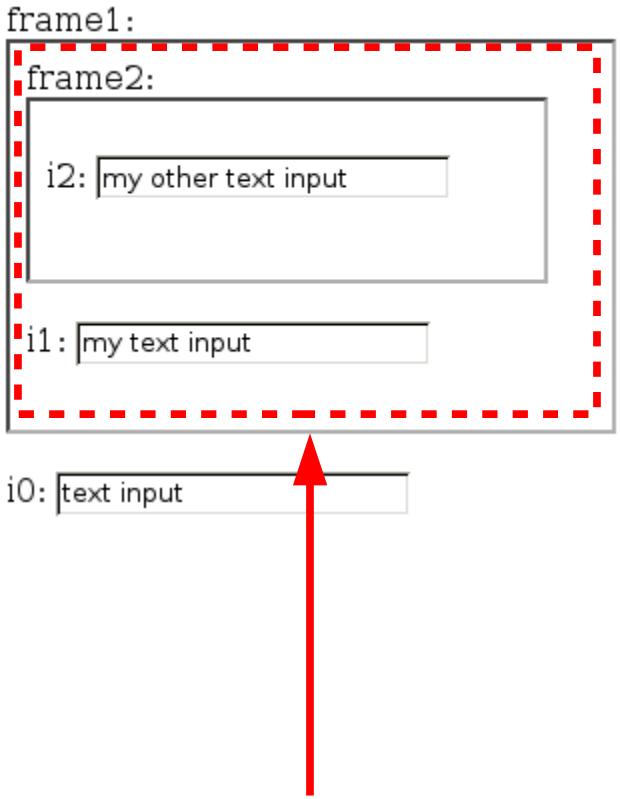



```
class A (object):

    def f (self):
        do_some_stuff()


a = A()
```

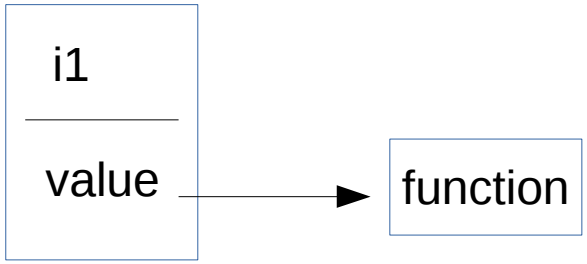


Applying the Decorator Pattern

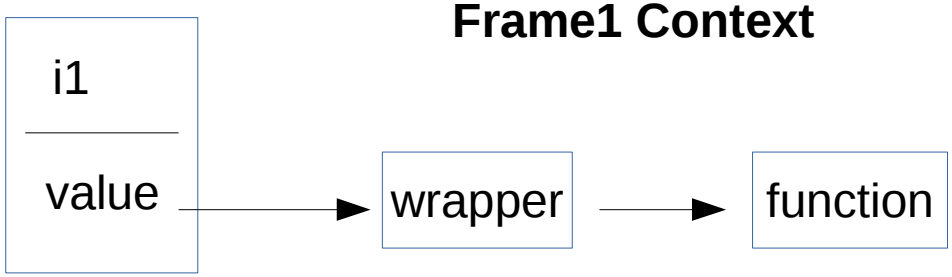
```
class Text(BasePageWidget):

    # setter
    def value(self, text):
        e = self.find_element(self.locator)
        e.clear()
        e.send_keys(text)


i1 = Text('#i1')
i1.value('new i1 text')
```



Frame1 Context



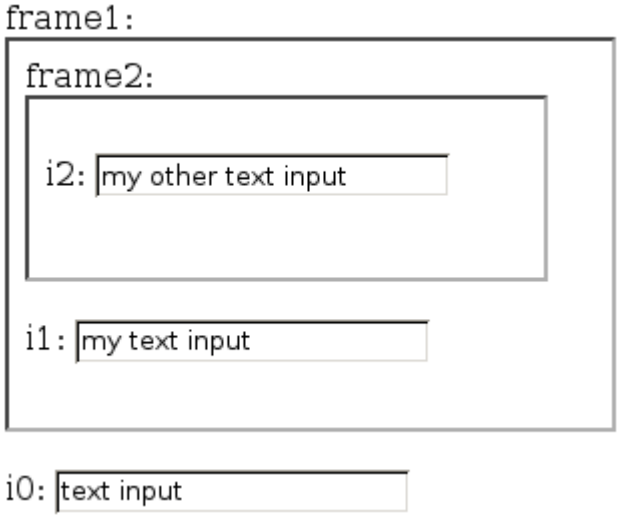
Before Decoration



After Decoration

Create decorated page objects

```
class FramedInputs(BasePageObject):
    def __init__(self):

        self.i0 = Text('#i0')

        self.i1 = IframeWrap( Text('#i1'), ['#frame1'] )

        self.i2 = IframeWrap( Text('#i2'), ['#frame2', '#frame1'] )
```

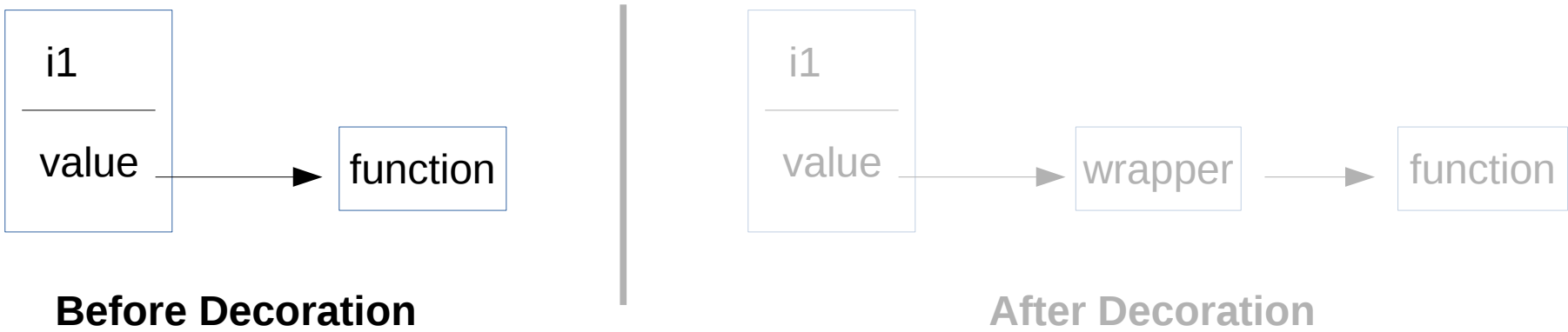


HTML element's page object

List of frames to traverse

Find callable attributes

```
class IframeTracker(object):
    ...
    def wrap_callable_attributes(self,o):
        for attr, item in o.__class__.__dict__.items():
            if callable(item):
                item = getattr(o,attr)
                setattr(o,attr,self.wrap_attribute(item))
```



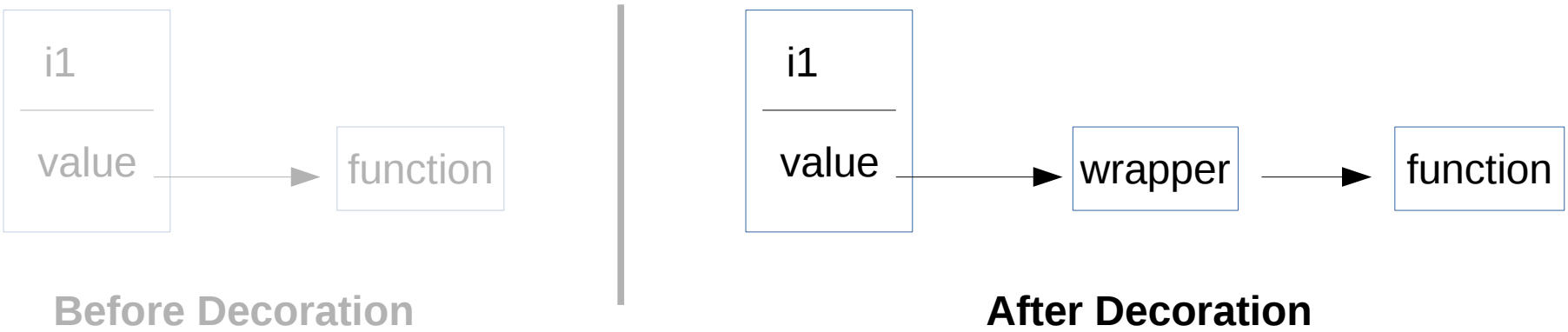
Wrap attributes with iframe calls

```
class IframeTracker(object):
    ...
    def wrap_callable_attributes(self,o):
        for attr, item in o.__class__.__dict__.items():
            if callable(item):
                item = getattr(o,attr)
                setattr(o,attr,self.wrap_attribute(item))

    def wrap_attribute(self,item):
        def wrapper(*args, **kwargs):
            ...
            switched = self._switch_to_iframe_context(final_framelevel)
            result = item(*args, **kwargs)
            self._switch_to_iframe_context(initial_framelevel)

            return result
        return wrapper
```

```
frame = self.find_element('#frame1')
self._browser.switch_to_frame(frame)
e = self.find_element(self.locator)
e.clear()
e.send_keys(text)
self._browser.switch_to_default_content()
```

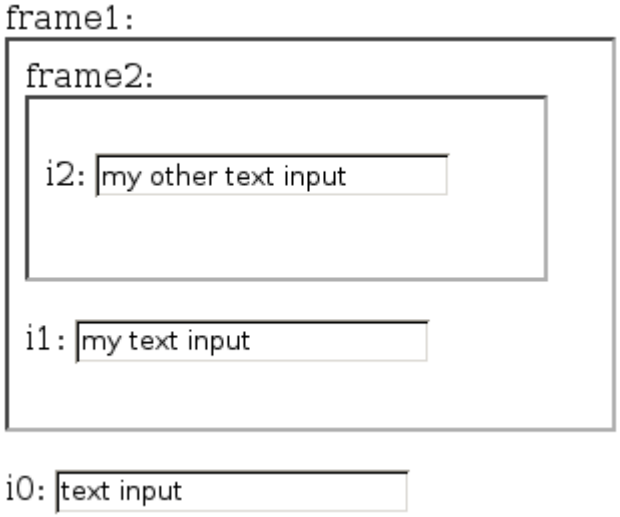
Store the wrapped attribute

```
class IframeTracker(object):
    ...
    def wrap_callable_attributes(self,o):
        for attr, item in o.__class__.__dict__.items():
            if callable(item):
                item = getattr(o,attr)
                setattr(o,attr,self.wrap_attribute(item))
```



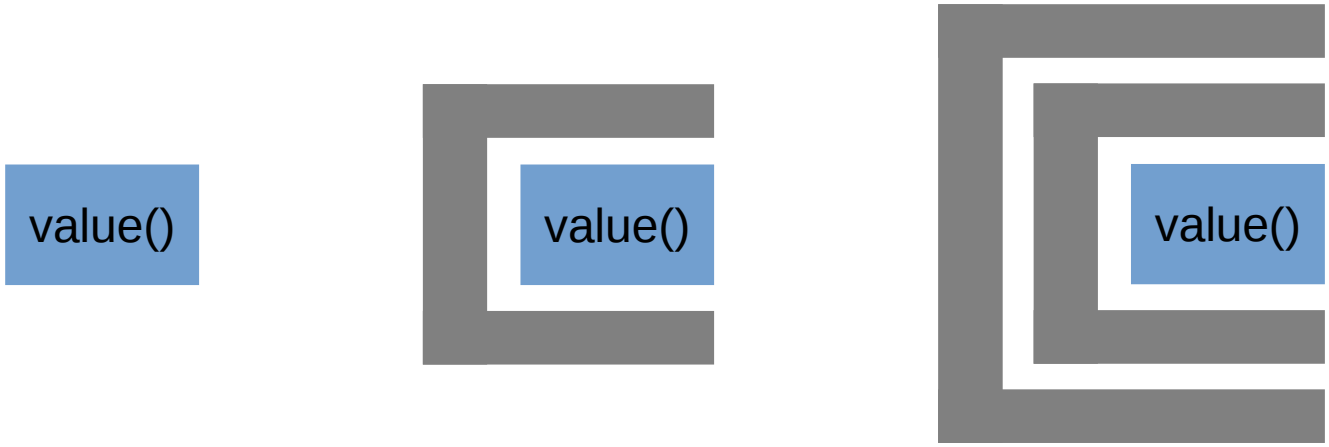
Using an IframeWrap'd page object

```
class FramedInputs(BasePageObject):
    def __init__(self):
        self.i0 = Text('#i0')
        self.i1 = IframeWrap( Text('#i1'), ['#frame1'] )
        self.i2 = IframeWrap( Text('#i2'), ['#frame2', '#frame1'] )


po = FramedInputs()

# print out the current text in the widgets
print "i0.value = %s" % (po.i0.value)
print "i1.value = %s" % (po.i1.value)
print "i2.value = %s" % (po.i2.value)

# update the text in the widgets
po.i0.value = 'i0 text'
po.i1.value = 'new i1 text'
po.i2.value = 'new i2 text too'
```



IframeWrap Pattern Summary

- Use IframeWrap Pattern when you have page objects for HTML elements that work the same inside or outside of an iframe
- Uses Decorator Pattern to wrap page object attributes with calls to enter and exit iframe



IframeWrap Pattern Gotchas

- Not all page object attributes need to be wrapped.
  - Keep a list of specific methods not to wrap.

- Wrapping Python properties can be tricky
  - Create a new class object from old class.
  - Decorate the property functions of new class.
  - Associate new class with page object.

More Info

- Design Patterns Book
- Websites
  - http://www.oodesign.com
  - http://sourcemaking.com
- Special Thanks
  - HUBzero Team
  - Sam Midkiff, Advisor