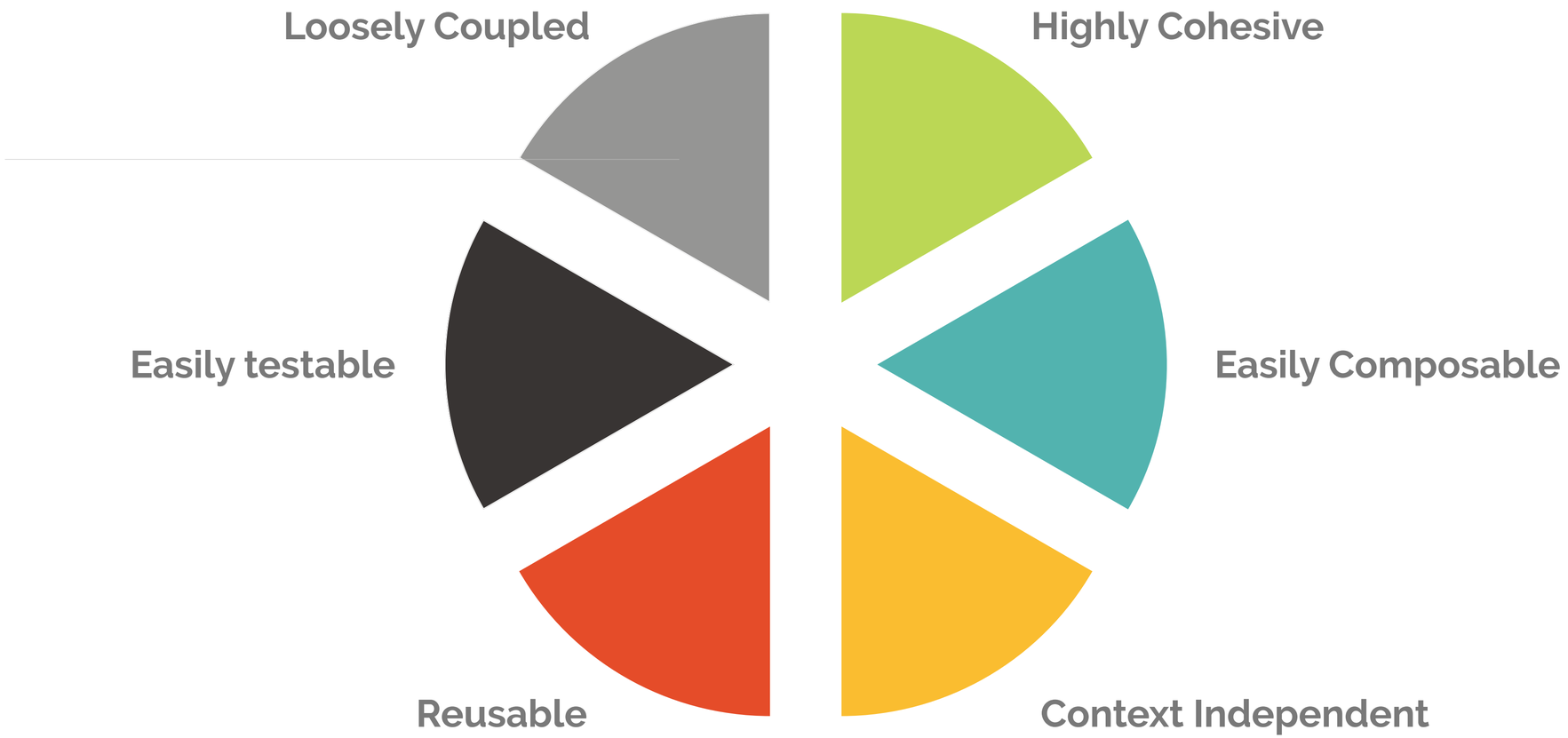
A SOLID APPROACH TO TEST AUTOMATION


Welcome!
Everything is fine.

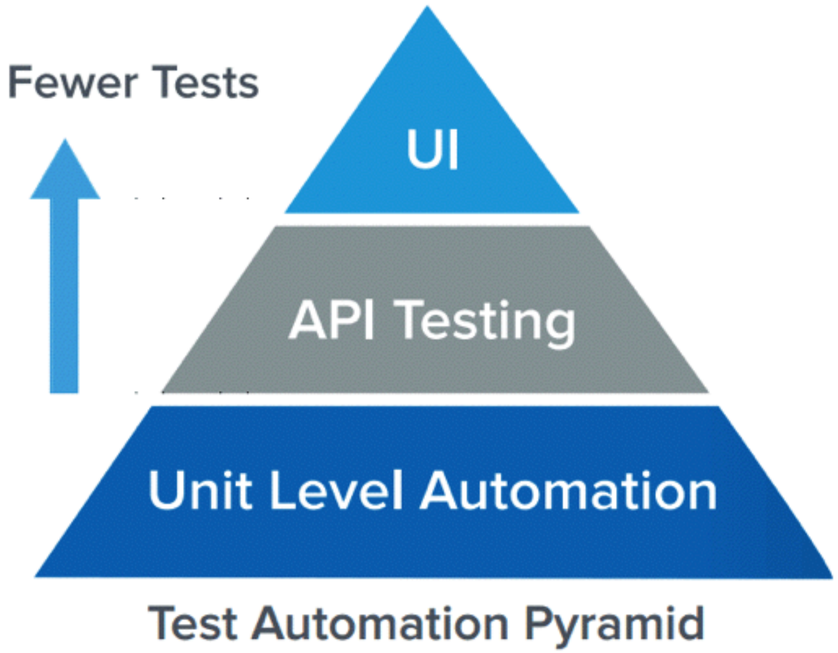
12 MONTHS LATER...

Tenants of Good Design



CHANGE IS HARD!

As a Test Engineer, Why do I care?



Test Automation Pyramid

Because Test Automation is an asset!

All software has two critical assets – its code base in a code repository and its test cases or scripts in a test repository.

Building a testing infrastructure is a complex process as it is continuously changing and evolving as the product under development does

Test code should be treated as a first class citizen, in the same manner as the production code

“I have been writing code for more than half a century… I’ve built a lot of systems. And from them all, and by taking them all into consideration, I’ve learned something startling.

The architecture rules are the same! “

Robert “Uncle Bob” Martin

A Design Approach to OOP

Because software development is not a Jenga Game

Single responsibility principle

Open-closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

WARNING

The following slides contain anecdotes and code based on real life cases. These examples (and potential jokes made by the presenter) are terribly written in nature and may offend some with their lack of wit.

Viewer discretion advised.

Single Responsibility Principle

```ruby
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

```
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

```ruby
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

```ruby
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

Single Responsibility Principle

Uncle Bob says...

"A class should only have one reason to change"

A class should do the smallest possible useful thing!


NEVER HALF-ASS TWO THINGS...
WHOLE-ASS ONE THING.
- RON SWANSON
Citytv

Why Does It Matter?

An application that is easy to change is like a box of building blocks

Combining two entities that change for different reasons at different times can lead to problems when changes are made to either entity

Coupled responsibilities leads to duplication. Duplicated code leads to additional maintenance and increases bugs.

Increase your application's chance of breaking unexpectedly if you depend on classes that do too much.

Automation Perspective

- Every class in your automation should only have a single responsibility and that all of its methods should be aligned with that responsibility
- The Page Object Pattern implements SRP very well; One class is responsible for only one web page in the application.
- Very big PO classes with responsibilities that don't explicitly address the page should be avoided

```ruby
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

```ruby
class ToDoListPage
   include PageObject

   element(:new_todo, id: ....)
   element(:todo_items, id: ....)
   element(:todos_remaining, id: ....)
   element(:filterTodos, id: ....)

end
```

```ruby
class ToDoListService

   def setup_list_test_data
     # Database interactions to setup data
   end

end
```

```ruby
class ToDoListActions

   def add_todo_item(item)
     #Stuff...
   end

   def add_todo_items(items)
     #Stuff and Things...
   end

   def toggle_all_completed
    # Stuff and Things again...
   end

   def filter_items
     # Stuff etc...
   end

end
```

Open-Closed Principle

```ruby
@new_user = AdminActions.create_new_user('Jon', 'Snow')

@new_case = CaseManagementActions.create_new_case_for_user(@new_user)
```

```ruby
After do

  if @new_case
    #Code to cleanup new case data
  end

  if @new_user
    #Code to cleanup new user data
  end

end
```

```ruby
@new_user = AdminActions.create_new_user('Jon', 'Snow')

@new_case = CaseManagementActions.create_new_case_for_user(@new_user)

@task_for_case = TaskPage.create_new_task_for_case(@new_case, 'He knows nothing…')

@order_for_case = UserPage.add_new_order_for_user(@new_user)
```

```ruby
After do

  if @new_user
    #Code to cleanup new user data
  end

  if @new_case
    #Code to cleanup new case data
  end

  if @task_for_case
    #Code to cleanup new task data
  end

  if @order_for_case
    #Code to cleanup new order data
  end

end
```

```ruby
After do

  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
end
```

```ruby
After do

  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
end
```

```
After do

  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
  if @new_case
    #Code to cleanup new case data
  end
  if @task_for_case
    #Code to cleanup new task data
  end
  if @order_for_case
    #Code to cleanup new order data
  end
  if @new_user
    #Code to cleanup new user data
  end
end
```

Open-Closed Principle

Uncle Bob says...

"Entities should be open for extension but closed for modification."

Extend the behavior by adding new code, not by changing old code that already works!

Why Does it Matter?

If it ain't broke, don't change it

Minimize the impact and potential errors from changing existing code by extending classes, not modifying them

Applications change as they are developed; our code should make it easy to response to business needs and new features

We (some of us) are human, we make mistakes there is no reason to risk breaking working functionality

Automation Perspective

- Classes within your automation should be expandable without risking broken or unexpected behavior
- Like production code, your classes will change as new requirements come up
- Avoiding huge if statements (or worse case statements) will allow for you automation to be more maintainable and less prone to error

Strategy Pattern

```ruby
class Grill
  attr_accessor :food

  def initialize food
    @food = food
  end

  def grilling
    "Grilling the #{food.type}!"
  end
end
```

```ruby
class HotDog
  def type
    'hot dogs'
  end
end

class Hamburger
  def type
    'hamburgers'
  end
end

class VeggiePatty
  def type
    'veggie patties'
  end
end
```

```ruby
@new_user = AdminActions.create_new_user('Jon', 'Snow')

@new_case = CaseManagementActions.create_new_case_for_user(@new_user)
```

```ruby
After do

  if @new_case
    #Code to cleanup new case data
  end

  if @new_user
    #Code to cleanup new user data
  end

end
```

```ruby
@new_user = AdminPage.create_new_user('Jon', 'Snow')
user_mess = UserMess.new(@new_user)
Janitor.add_mess(user_mess)

@new_case = CaseManagementPage.create_new_case_for_user(@new_user)
case_mess = CaseMess.new(@new_case)
Janitor.add_mess(case_mess)
```

```ruby
class UserMess

  def initialize(user)
   @user = user
  end

  def clean
    #Code to cleanup new user data
  end

end

class CaseMess

  def initialize(case)
   @case = case
  end

  def clean
    #Code to cleanup new case data
  end

end
```

```ruby
class Janitor
  class << self

    def add_mess(mess)
      @messes ||= []
      @messes << mess
    end

    def clean_messes
      unless @messes.nil?
        @messes.each { |mess| mess.clean }
        @messes.clear
      end
    end
  end
end
```

```ruby
After do

  #...

  Janitor.clean_messes

  #...

end
```

```ruby
@new_user = AdminPage.create_new_user('Jon', 'Snow')
user_mess = UserMess.new(@new_user)
Janitor.add_mess(user_mess)

@new_case = CaseManagementPage.create_new_case_for_user(@new_user)
case_mess = CaseMess.new(@new_case)
Janitor.add_mess(case_mess)
```

```ruby
class UserMess

  def initialize(user)
   @user = user
  end

  def clean
    #Code to cleanup new user data
  end

end

class CaseMess

  def initialize(case)
   @case = case
  end

  def clean
    #Code to cleanup new case data
  end

end
```

```ruby
class Janitor
  class << self

    def add_mess(mess)
      @messes ||= []
      @messes << mess
    end

    def clean_messes
      unless @messes.nil?
        @messes.each { |mess| mess.clean }
        @messes.clear
      end
    end
  end
end
```

```ruby
After do

  #...

  Janitor.clean_messes

  #...

end
```

```ruby
@new_user = AdminPage.create_new_user('Jon', 'Snow')
user_mess = UserMess.new(@new_user)
Janitor.add_mess(user_mess)

@new_case = CaseManagementPage.create_new_case_for_user(@new_user)
case_mess = CaseMess.new(@new_case)
Janitor.add_mess(case_mess)
```

```ruby
class UserMess

  def initialize(user)
   @user = user
  end

  def clean
    #Code to cleanup new user data
  end

end

class CaseMess

  def initialize(case)
   @case = case
  end

  def clean
    #Code to cleanup new case data
  end

end
```

```ruby
class Janitor
  class << self

    def add_mess(mess)
      @messes ||= []
      @messes << mess
    end

    def clean_messes
      unless @messes.nil?
        @messes.each { |mess| mess.clean }
        @messes.clear
      end
    end
  end
end
```

```ruby
After do

  #...

  Janitor.clean_messes

  #...

end
```

```ruby
@new_user = AdminPage.create_new_user('Jon', 'Snow')
user_mess = UserMess.new(@new_user)
Janitor.add_mess(user_mess)

@new_case = CaseManagementPage.create_new_case_for_user(@new_user)
case_mess = CaseMess.new(@new_case)
Janitor.add_mess(case_mess)
```

```ruby
class UserMess

  def initialize(user)
   @user = user
  end

  def clean
    #Code to cleanup new user data
  end

end

class CaseMess

  def initialize(case)
   @case = case
  end

  def clean
    #Code to cleanup new case data
  end

end
```

```ruby
class Janitor
  class << self

    def add_mess(mess)
      @messes ||= []
      @messes << mess
    end

    def clean_messes
      unless @messes.nil?
        @messes.each { |mess| mess.clean }
        @messes.clear
      end
    end
  end
end
```

```ruby
After do

  #...

  Janitor.clean_messes

  #...

end
```

```ruby
@new_user = AdminPage.create_new_user('Jon', 'Snow')
user_mess = UserMess.new(@new_user)
Janitor.add_mess(user_mess)

@new_case = CaseManagementPage.create_new_case_for_user(@new_user)
case_mess = CaseMess.new(@new_case)
Janitor.add_mess(case_mess)
```

```ruby
class UserMess

  def initialize(user)
   @user = user
  end

  def clean
    #Code to cleanup new user data
  end

end

class CaseMess

  def initialize(case)
   @case = case
  end

  def clean
    #Code to cleanup new case data
  end

end
```

```ruby
class Janitor
  class << self

    def add_mess(mess)
      @messes ||= []
      @messes << mess
    end

    def clean_messes
      unless @messes.nil?
        @messes.each { |mess| mess.clean }
        @messes.clear
      end
    end
  end
end
```

```ruby
After do

  #...

  Janitor.clean_messes

  #...

end
```

Liskov Substitution Principle

```ruby
class NewAccountForm
  include PageObject

  # Shared Elements across all account pages

  def populate_form_with(account_data_hash)
    # Do things specific to forms for existing
      Customers
  end

end
```

```ruby
class NewAccountForm
  include PageObject

    #...

  def populate_form_with(account_data_hash)
    # Do things specific to forms for existing
      Customers

    # Retrieve Account data from page and store
      in hash

    page_account_data_hash
  end

end
```

```ruby
class NewAccountForNewCustomerForm < NewAccountForm

  #...

  def populate_form_with(account_data_hash)

    # Do things specific to forms for New Customers

    # Retrieve Account data from page and store in hash

    page_account_data_hash

  end

end
```

```ruby
class NewAccountForm
  include PageObject

  #...


  def populate_form_with(account_data_hash)

    # Do things specific to forms for existing
      Customers

    # Retrieve Account data from page and store
      in hash

    page_account_data_hash

  end

end
```

```ruby
class NewAccountForExistingCustomerForm < NewAccountForm

  #...

  def populate_form_with(account_data_json)

    account_data_hash = JSON.parse(account_data_json)

    # Do things specific to forms for existing Customers

    # Retrieve Account data from page and save as json

   page_account_data_json

  end

end
```

```ruby
class NewAccountForm
  include PageObject

  #...

  def populate_form_with(account_data_hash)

    # Do things specific to forms for existing
      Customers

    # Retrieve Account data from page and store
      in hash

    page_account_data_hash

  end

end
```

```ruby
class NewAccountForExistingCustomerForm < NewAccountForm

  #...

  def populate_form_with(account_data_json)

    account_data_hash = JSON.parse(account_data_json)

    # Do things specific to forms for existing Customers

    # Retrieve Account data from page and save as json

   page_account_data_json

  end

end
```

Liskov Substitution Principle

Barbara Liskov Says...

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

Liskov Substitution Principle

Uncle Bob says...

"Derived classes must be substitutable for their base classes."

Liskov Substitution Principle

Twitter says...

"Sort of like when they changed the actor for Daario Naharis in Game of Thrones."

LSP is all about Contracts

What does the superclass method **require**?

What does the superclass method **promise**?

“Substitutability is possible only when objects behave as expected and subclasses are expected to conform to their superclass’s interface. “

Why Does It Matter?

honor your contract!

For subclasses in a hierarchy to be easy to use, they must agree to a “contract” with their superclass.

A superclass places restrictions on the “required” inputs and “promised” result of a methods

Subclasses violating this contract are not truly a “kind-of” their super class; this casts doubt on the whole hierarchal structure

Automation Perspective

- Contractually guaranteeing that your methods work for existing and new items will prevent unexpected behavior in your system
- Allowing your PageObject hierarchies to be substitutable will grant you more flexibility as pages evolve
- Changing the behavior of a PageObject subclasses will make your framework harder to use; don't change what other people in the code expect the behavior to be!

```ruby
class NewAccountForm
  include PageObject

  #...


  def populate_form_with(account_data_hash)

    # Do things specific to forms for existing
      Customers

    # Retrieve Account data from page and store
      in hash

    page_account_data_hash

  end

end
```

```ruby
class NewAccountForExistingCustomerForm < NewAccountForm

  #...

  def populate_form_with(account_data_json)

    account_data_hash = JSON.parse(account_data_json)

    # Do things specific to forms for existing Customers

    # Retrieve Account data from page and save as json

   page_account_data_json

  end

end
```

```ruby
class NewAccountForExistingCustomerForm < NewAccountForm
  #...
  def populate_form_with(account_data_hash)
    # Do things specific to forms for existing Customers
    # Retrieve Account data from page and save as json
    page_account_data_hash
  end
end

class NewAccountActions
  def create_account_for_existing_customer
    #...
    account_data_hash = JSON.parse(account_data_json)
    NewAccountForExistingCustomerForm.populate_form_with(account_data_hash)

    # ...
  end
end
```

Interface Segregation Principle

```
when /^I add a product with name (.*) to the cart$/ do |product_name|
on(HomePage).search = product_name
@result = on(ResultsPage).find_result(product_name)
@result.click
@purchase_price = on(ProductPage).purchase_price
on(ProductPage).add_to_cart
end
```

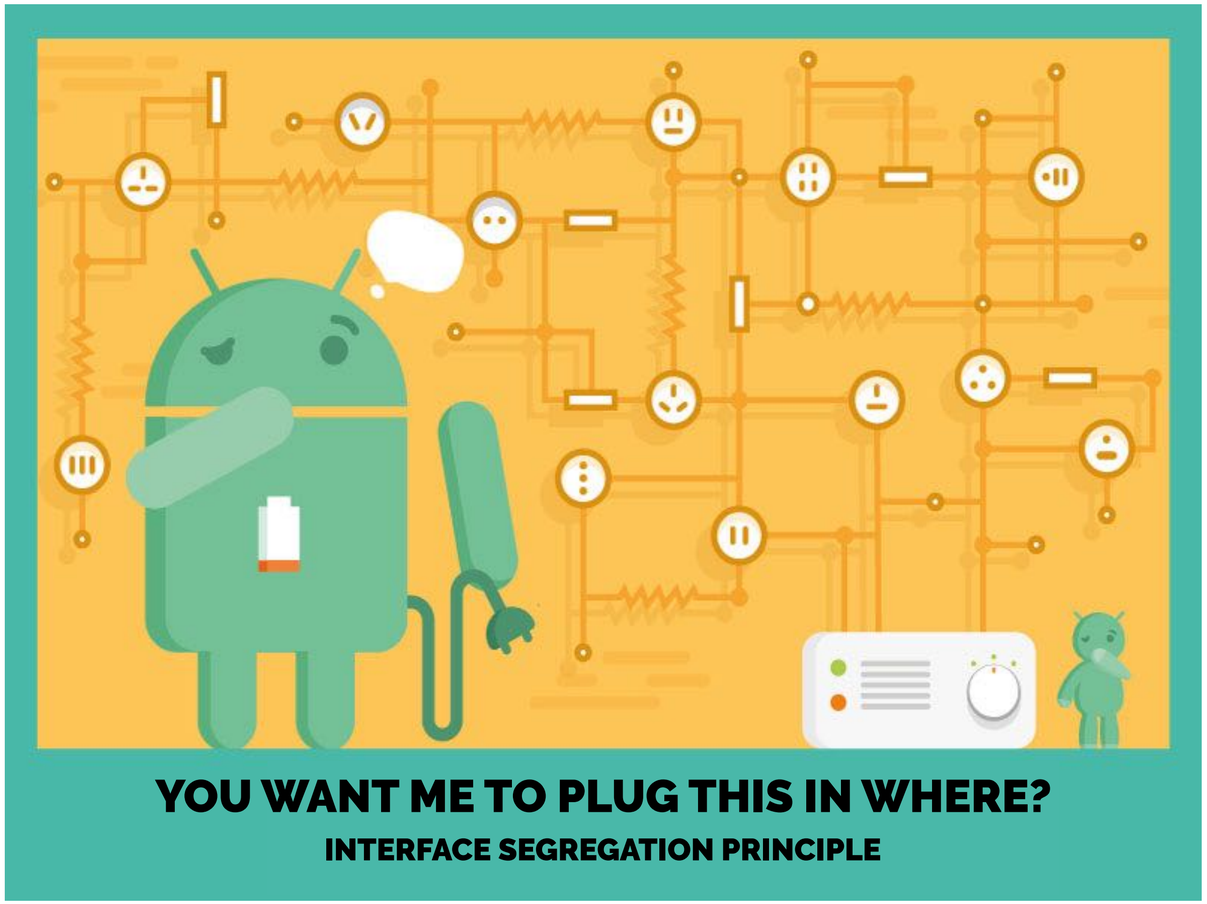
```
when /^I add a product with name (.*) to the cart$/ do |product_name|
on(HomePage).search = product_name
@result = on(ResultsPage).find_result(product_name)
@result.click
@purchase_price = on(ProductPage).purchase_price
on(ProductPage).add_to_cart
on(ProductPage).open_cart
end
```

Interface Segregation Principle

Uncle Bob says...

"Many client specific interfaces are better than general purpose interface."

Why Does It Matter?



Divide and Conquer

Simplifies the interface that any one client will use and removes dependencies that might cause others to develop on parts of the interface that they don't need

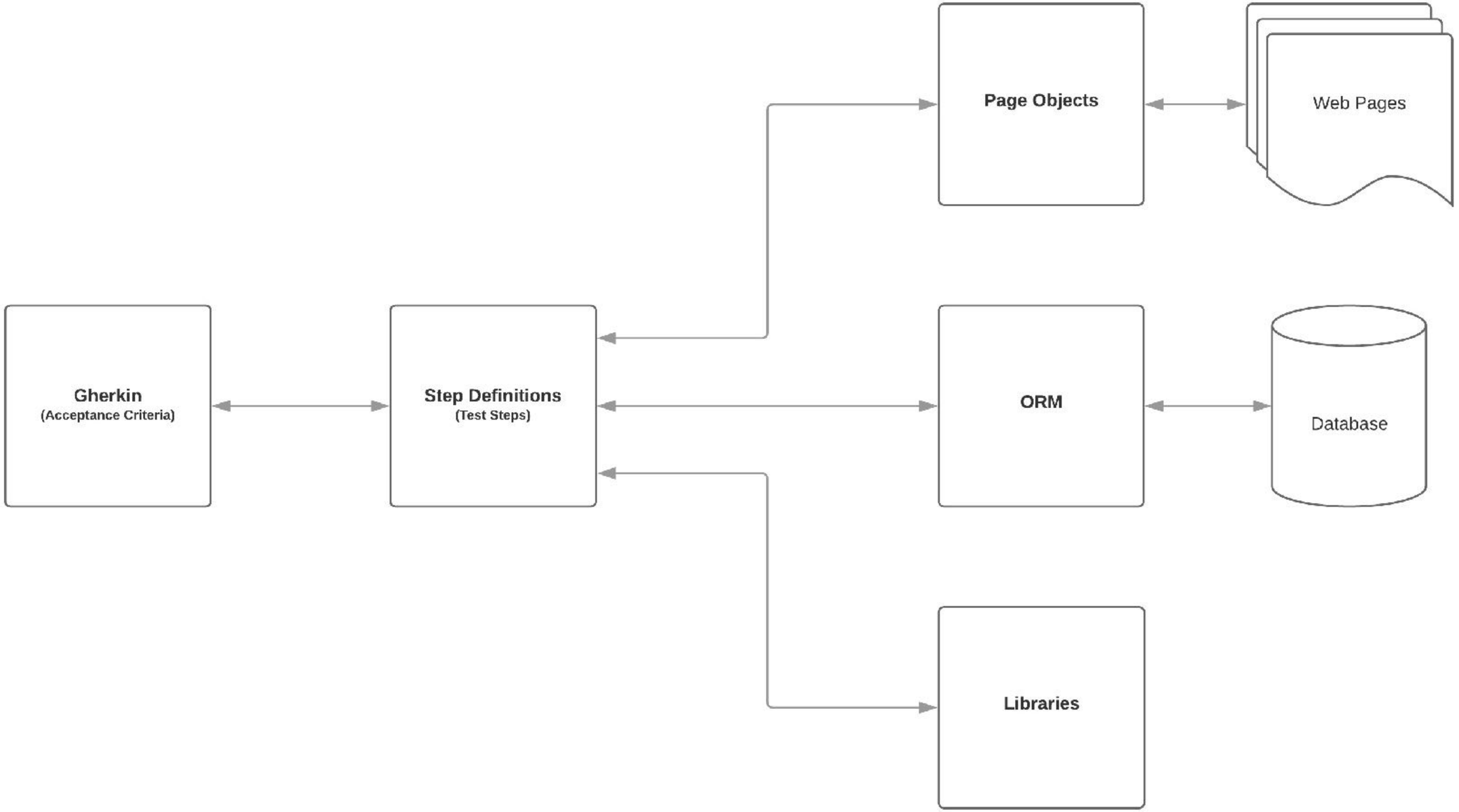
Provides flexibility by reducing one generic interface and many classes implementing it to many interfaces being consumed by highly specialized classes

Automation Perspective

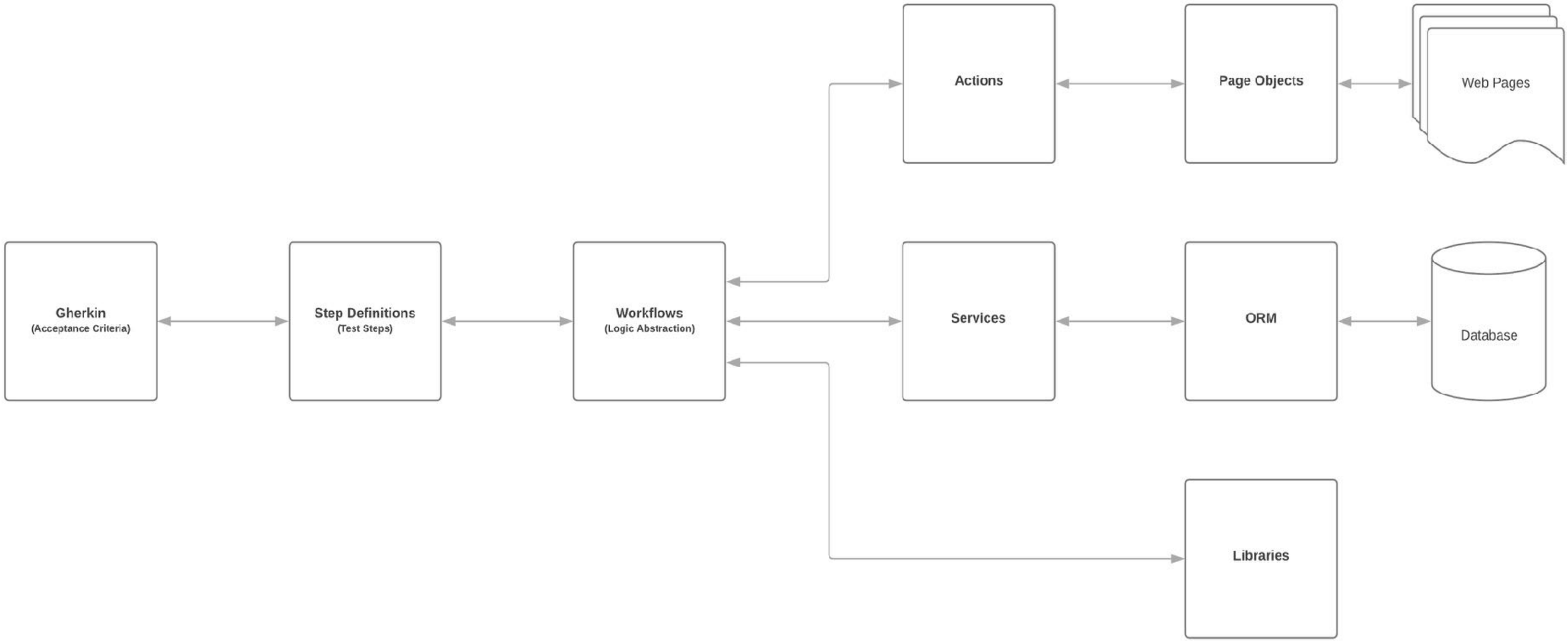
- General use step definitions can lead to issues as the system behavior changes; defining many step, but with specific meaning will create more cohesive cases (and make your gherkin more consumable!)

- Separating the “implementation” of your automation (Page Objects, Models, Actions, etc) from the test execution will allow test cases to be more maintainable and readable

An Aside...


Page Objects
Web Pages
Gherkin
(Acceptance Criteria)
Step Definitions
(Test Steps)
ORM
Database
Libraries

Utilizing Workflows



```
when /^I add a product with name (.*) to the cart$/ do |product_name|
on(HomePage).search = product_name
@result = on(ResultsPage).find_result(product_name)
@result.click
@purchase_price = on(ProductPage).purchase_price
on(ProductPage).add_to_cart
on(ProductPage).open_cart
end
```

```ruby
class SearchWorkflow
  def search_for_book_with_title(search_criteria)
      #...
  end
end

class ProductWorkflow
  def view_product_details_for_result_with_name(product_result_name)
      #...
  end
  def open_shopping_cart
      #...
  end
end


when /^I add a textbook to the cart$/ do
    SearchWorkflow.search_for_book_with_title('Preparing for a Conference Talk')
    ProductWorkflow.view_product_details_for_result_with_name('Conference Speaking for Dummies')
    ProductWorkflow.open_shopping_cart
end
```

Dependency Inversion Principle

```ruby
class ProductCatalogWorkflow

  def initialize(product_catalog_actions = ProductCatalogActions.new)
    @product_catalog_actions = product_catalog_actions
  end

  def configure_product_in_cart product
    @product_catalog_actions.select_product_in_cart product
    @product_catalog_actions.open_options_for product
    @product_catalog_actions.product_color = product.color
    @product_catalog_actions.product_height = product.height
    @product_catalog_actions.product_width = product.width
    @product_catalog_actions.save_cart_item
  end

  #...

end
```

Dependency Inversion Principle

Uncle Bob says...

"Depend upon abstractions, do not depend upon concretions."

```java
public class LightBulb {
  public void turnOn() {
    System.out.println("LightBulb: Bulb turned on...");
  }
  public void turnOff() {
    System.out.println("LightBulb: Bulb turned off...");
  }
}
```

```
public class ElectricPowerSwitch {
  public LightBulb lightBulb;
  public boolean on;
  public ElectricPowerSwitch(LightBulb lightBulb) {
    this.lightBulb = lightBulb;
    this.on = false;
  }
  public boolean isOn() {
    return this.on;
  }
  public void press(){
    boolean checkOn = isOn();
    if (checkOn) {
        lightBulb.turnOff();
    this.on = false;
    } else {
        lightBulb.turnOn();
       this.on = true;
     }
  }
}
```

```
public interface Switchable {
  void turnOn();
  void turnOff();
}
```

```
public class LightBulb implements Switchable {
  @Override
  public void turnOn() {
    System.out.println("LightBulb: Bulb turned on...");
  }

  @Override
  public void turnOff() {
    System.out.println("LightBulb: Bulb turned off...");
  }
}
```

```
public class ElectricPowerSwitch {
  public Switchable client;
  public boolean on;
  public ElectricPowerSwitch(Switchable client) {
    this.client = client;
    this.on = false;
  }
  public boolean isOn() {
    return this.on;
  }
  public void press(){
    boolean checkOn = isOn();
    if (checkOn) {
        client.turnOff();
    this.on = false;
    } else {
        client.turnOn();
        this.on = true;
    }

  }
}
```

Why Does It Matter?

Don't build a new car every time you have to drive to work!

Preventing your classes from being ridged (relying on specific implementations)

Provides flexibility by allowing any object that conforms to an "interface" to be used

Automation Perspective

- Classes within your automation framework (Workflows, Actions, Services, etc) should not be limited by concrete reference to its dependencies

- Using Dependency Inversion will allow classes within your framework to be “flexible” as the application under test changes

Before DIP

```ruby
class ProductCatalogWorkflow

  def initialize(product_catalog_actions = ProductCatalogActions.new)
    @product_catalog_actions = product_catalog_actions
  end

  def configure_product_in_cart product
    @product_catalog_actions.select_product_in_cart product
    @product_catalog_actions.open_options_for product
    @product_catalog_actions.product_color = product.color
    @product_catalog_actions.product_height = product.height
    @product_catalog_actions.product_width = product.width
    @product_catalog_actions.save_cart_item
  end

  #...

end
```

After DIP

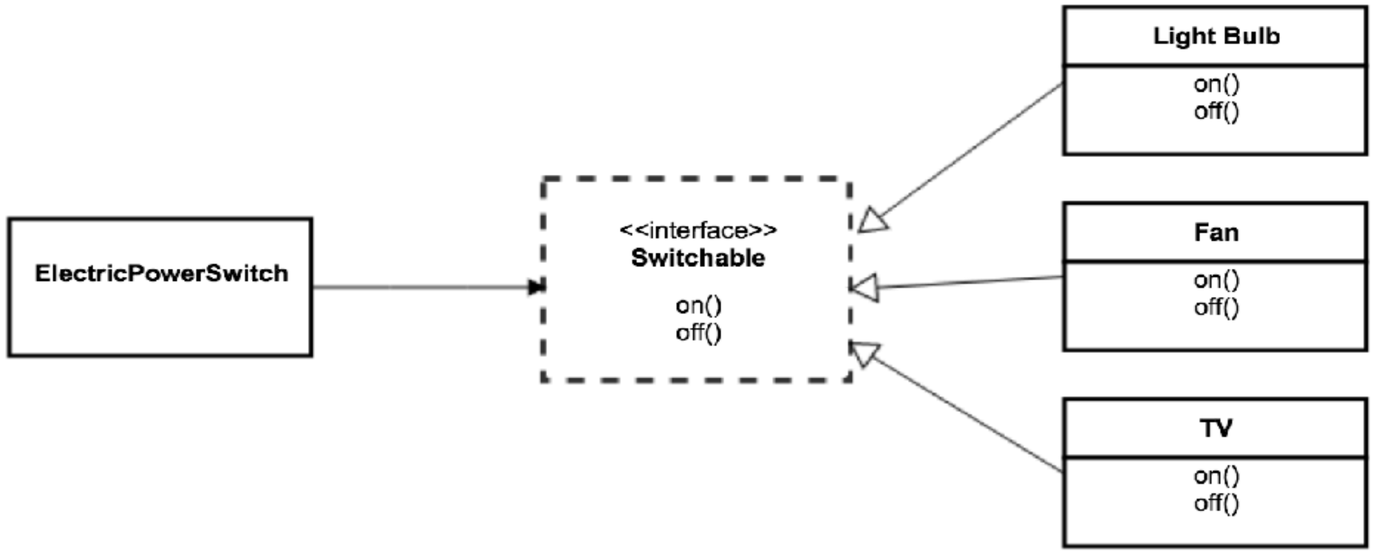
```
class ProductCatalogWorkflow

  def initialize(product_catalog_actions = ProductCatalogActions.new)
    @product_catalog_actions = product_catalog_actions
  end

  def configure_product_in_cart product
    @product_catalog_actions.select_product_in_cart product
    @product_catalog_actions.open_options_for product
    @product_catalog_actions.product_color = product.color
    @product_catalog_actions.product_height = product.height
    @product_catalog_actions.product_width = product.width
    @product_catalog_actions.save_cart_item
  end

  #...

end
```

```ruby
class ProductCatalogWorkflow

  def initialize(product_catalog_actions = ProductCatalogActions.new)
    @product_catalog_actions = product_catalog_actions
  end

  def configure_product_in_cart product
    raise 'Must use a Picture Frame!' unless product.is_a?(PictureFrame)
    @product_catalog_actions.select_product_in_cart product
    @product_catalog_actions.open_options_for product
    @product_catalog_actions.product_color = product.color
    @product_catalog_actions.product_height = product.height
    @product_catalog_actions.product_width = product.width
    @product_catalog_actions.save_cart_item
  end

  #...

end
```

Utilizing Protocols

Ruby doesn't have explicit interfaces, but conceptually has something similar called **Protocols** (coined by Jim Weirich)

Methods are written in a way that allows them to utilize any dependency that conform to the *idea* of how the dependency should respond



```ruby
class ProductCatalogWorkflow

  def initialize(product_catalog_actions = ProductCatalogActions.new)
    @product_catalog_actions = product_catalog_actions
  end

  def configure_product_in_cart product
    raise 'Must respond to color, height, width' unless [:color,:height, :width].all?
      { |message| product.respond_to?(message) }
    @product_catalog_actions.select_product_in_cart product
    @product_catalog_actions.open_options_for product
    @product_catalog_actions.product_color = product.color
    @product_catalog_actions.product_height = product.height
    @product_catalog_actions.product_width = product.width
    @product_catalog_actions.save_cart_item
  end

  #...

end
```

Chicken Typing

```ruby
# Configures a product in the cart with a color, height,
and width
#
# @param product [Object] the product being configured in
the cart.
# product must respond to :color, :height, :width

def configure_product_in_cart product
  @product_catalog_actions.select_product_in_cart product
  @product_catalog_actions.open_options_for product
  @product_catalog_actions.product_color = product.color
  @product_catalog_actions.product_height = product.height
  @product_catalog_actions.product_width = product.width
  @product_catalog_actions.save_cart_item
end
```

Instance Method Details

`#configure_product_in_cart(product) ⇒ Object`

Configures a product in the cart with a color, height, and width

product** must respond to **:color, :height, :width

Parameters:

- **`product`** (`Object`) — the product being configured in the cart.

[View source]

TL;DR

Too Long; Didn't Read

Change is hard, but with good design it is manageable

Test Automation should be held to the same standards as any software

By applying the SOLID Principles and other design patterns, your test automation can be adaptable to the ever evolving system it tries to test

SOLID

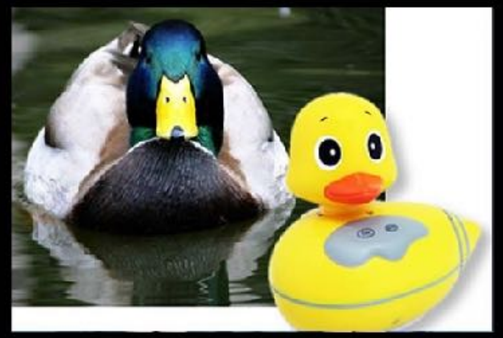
Software Development is not a Jenga game

Interface Segregation Principle

You want me to plug this in *where?*

Single Responsibility Principle

Just because you *can* doesn't mean you *should.*



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

Thanks for coming!

@bsbellanca

/bellanc