Download the file to see the comments

Architecture tips for simple(r) system tests

Amit Wertheimer

System tests

- End-to-end
- No mocks*
- Business logic driven
- Multiple checks per test-function
- Brittle by nature
- Long (er)

The Login example

```
 public void testLogin() {
    SignInPage signInPage = new SignInPage(selenium);
    HomePage homePage =
signInPage.loginValidUser("userName",
    "password");
    Assert.assertTrue(selenium.isElementPresent("compose
button" ,
         "Login was unsuccessful");
 }
```

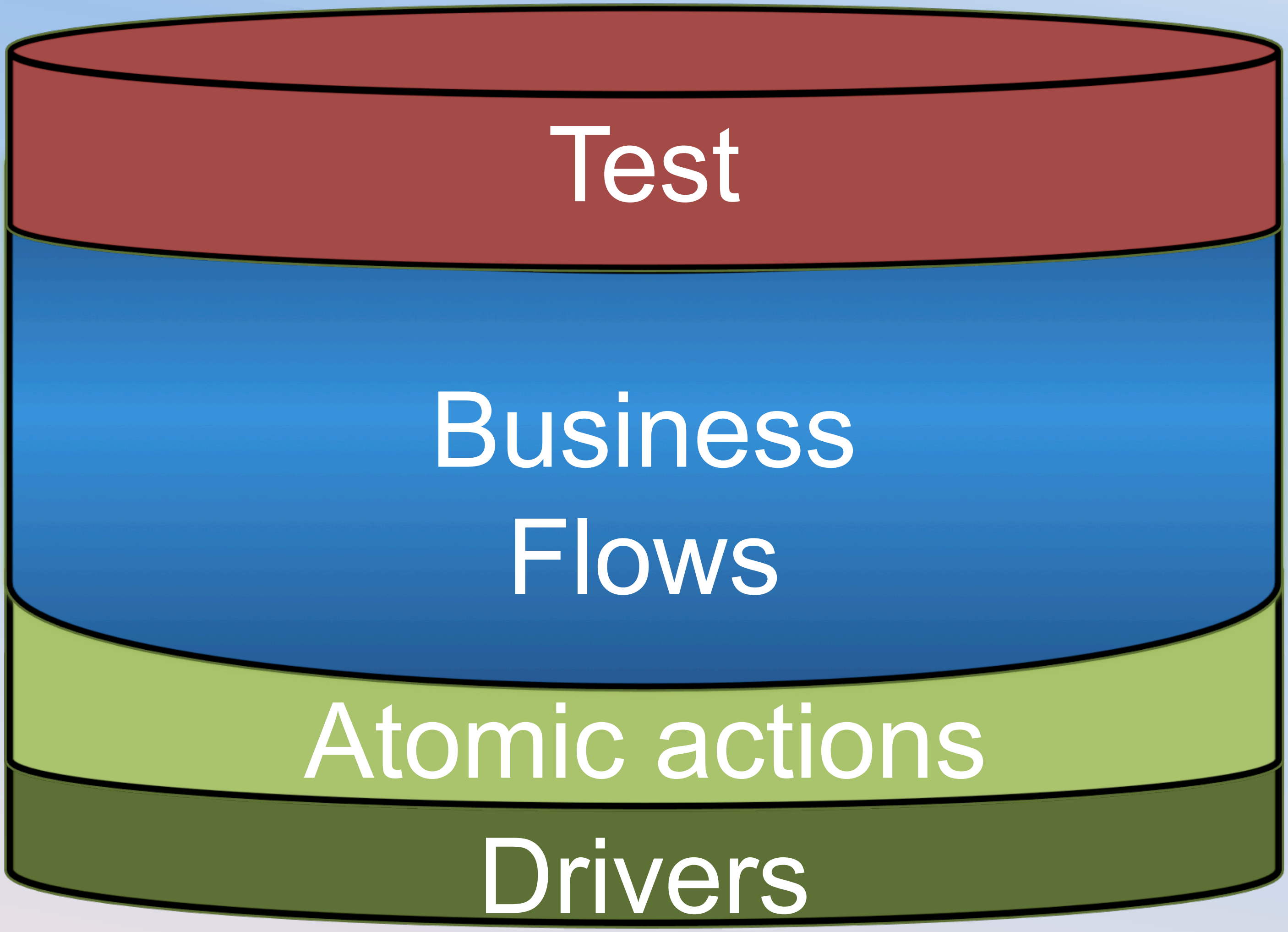
http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-

Meanwhile, in the real world…

```
@Test
@Parameters("UserProvider")
public void testCheckoutNewCustomer(User user) {
    new MainPage(driver).clickCheckOut();
    IsRegisteredPage isRegisteredPage = new IsRegisteredPage(driver);
    RegistrationPage registrationPage =
isRegisteredPage.chooseNewCustomer(user.getEmail());
    assertTrue(registrationPage.isInPage(), "Expecting to be in registration page");
    registrationPage.fillUserDetails(user);
    ShippingAddressPage shippingAddressPage = new ShippingAddressPage(driver);
    ShippingOptionPage shippingOptionsPage =
shippingAddressPage.fillShippingAddress(user);
    PaymentDetailsPage paymentdetailsPage = shippingOptionsPage.standardShipping();
    BillingAddressPage billingAddressPage = paymentdetailsPage.PayWithCreditCard(user);
    ConfirmationPage confirmationPage = billingAddressPage.enterNewBillingAddress(user);
    confirmationPage.confirm();
   assertTrue(DbUtils.isCustomerExist(email), "new customer supposed to exist");
   assertEquals(CartUtils.getNumberOfItemsInCart(), 0, "cart supposed to be empty");
   assertTrue(DbUtils.getOrdersForUser(email).size() > 0, "order should pass to processing");
}
```

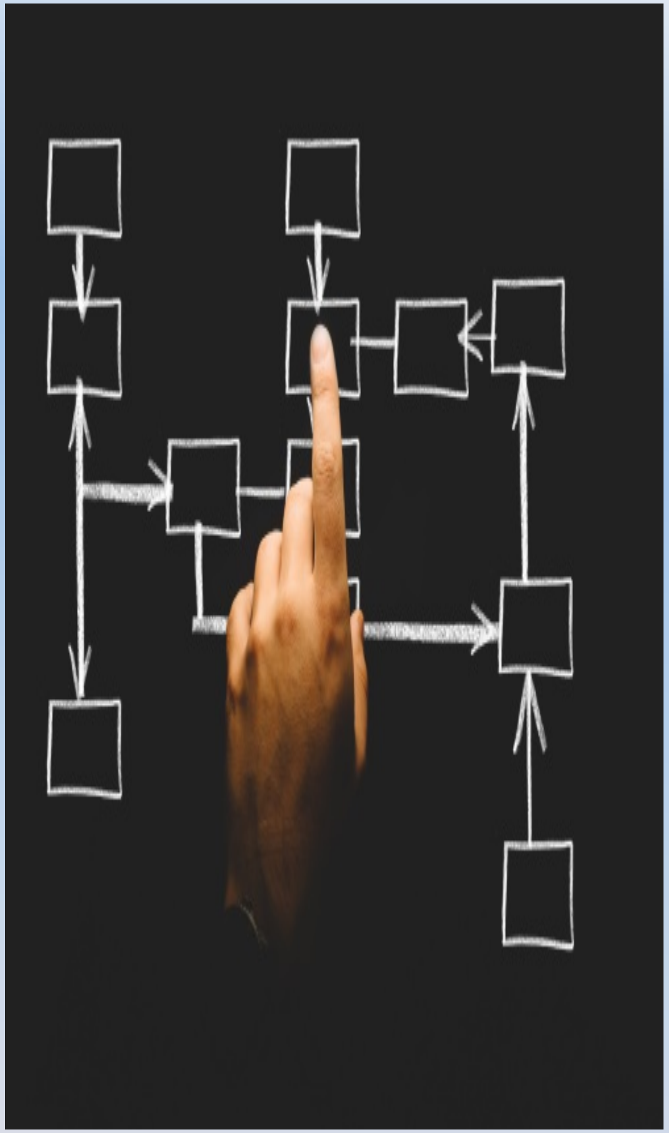
Ogres have layers

The system automation cake



RSA

3 Ways to organise your business flows



I. Utility functions

- Easy!
- Short to implement!
- You are probably doing this already

How does it look?

```
@Test
@Parameters("userProvider")
public void testCheckoutNewCustomer(User user) {
   CheckoutFlows.checkOutAndRegisterNewUser(user,
driver);
   DbAssertions.assertCustomersExist(user.getEmail());
}
```

The mess is still under the carpet

```
public static void checkoutAndRegisterNewUser(User user, WebDriver driver) {
    new MainPage(driver).clickCheckOut();
    IsRegisteredPage isRegisteredPage = new IsRegisteredPage(driver);
    RegistrationPage registrationPage = isRegisteredPage.chooseNewCustomer(user.getEmail());
    if (!registrationPage.isInPage() {
        throw new IncorrectPageException("Expecting to be in registration page");
    }
    registrationPage.fillUserDetails(user);
    ShippingAddressPage shippingAddressPage = new ShippingAddressPage(driver);
    ShippingOptionPage shippingOptionsPage =     shippingAddressPage.fillShippingAddress(user);
    PaymentDetailsPage paymentdetailsPage = shippingOptionsPage.standardShipping();
    BillingAddressPage billingAddressPage = paymentdetailsPage.PayWithCreditCard(user);
    ConfirmationPage confirmationPage = billingAddressPage.enterNewBillingAddress(user);
    confirmationPage.confirm();
}
```

Thumb rules

- Keep the functions independent
- Only test functions should fail a test
- Minimize the number of parameters sent to each function
- Increase discoverability – split functions according to business logic rules

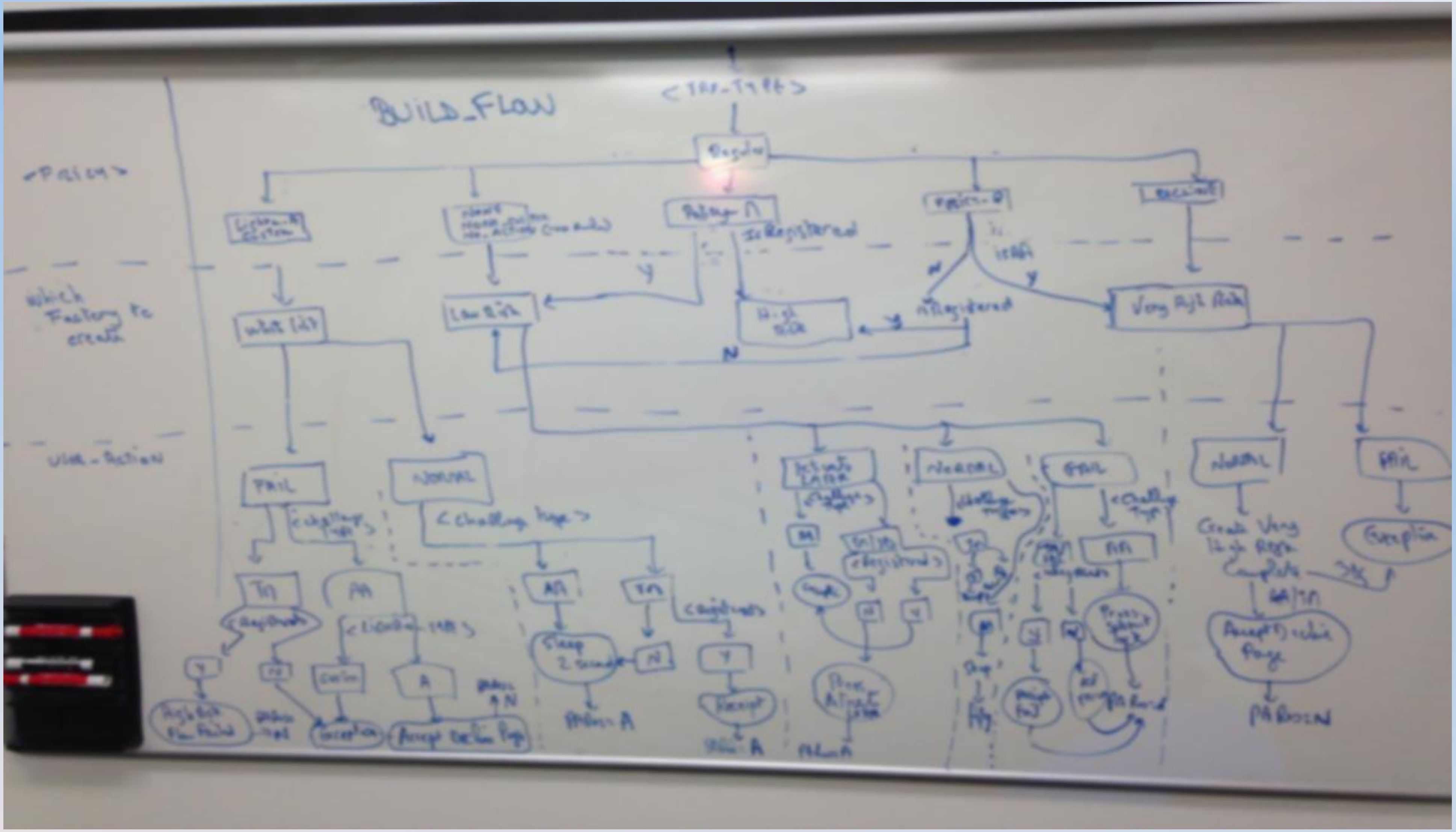
Yes, But…

- Easy to do without thinking (and getting it painfully wrong)
- Decision fatigue (trying to find the correct function)
- Function explosion over time
- Expensive maintenance

Abort? wait?

RSA

Back to the drawing board



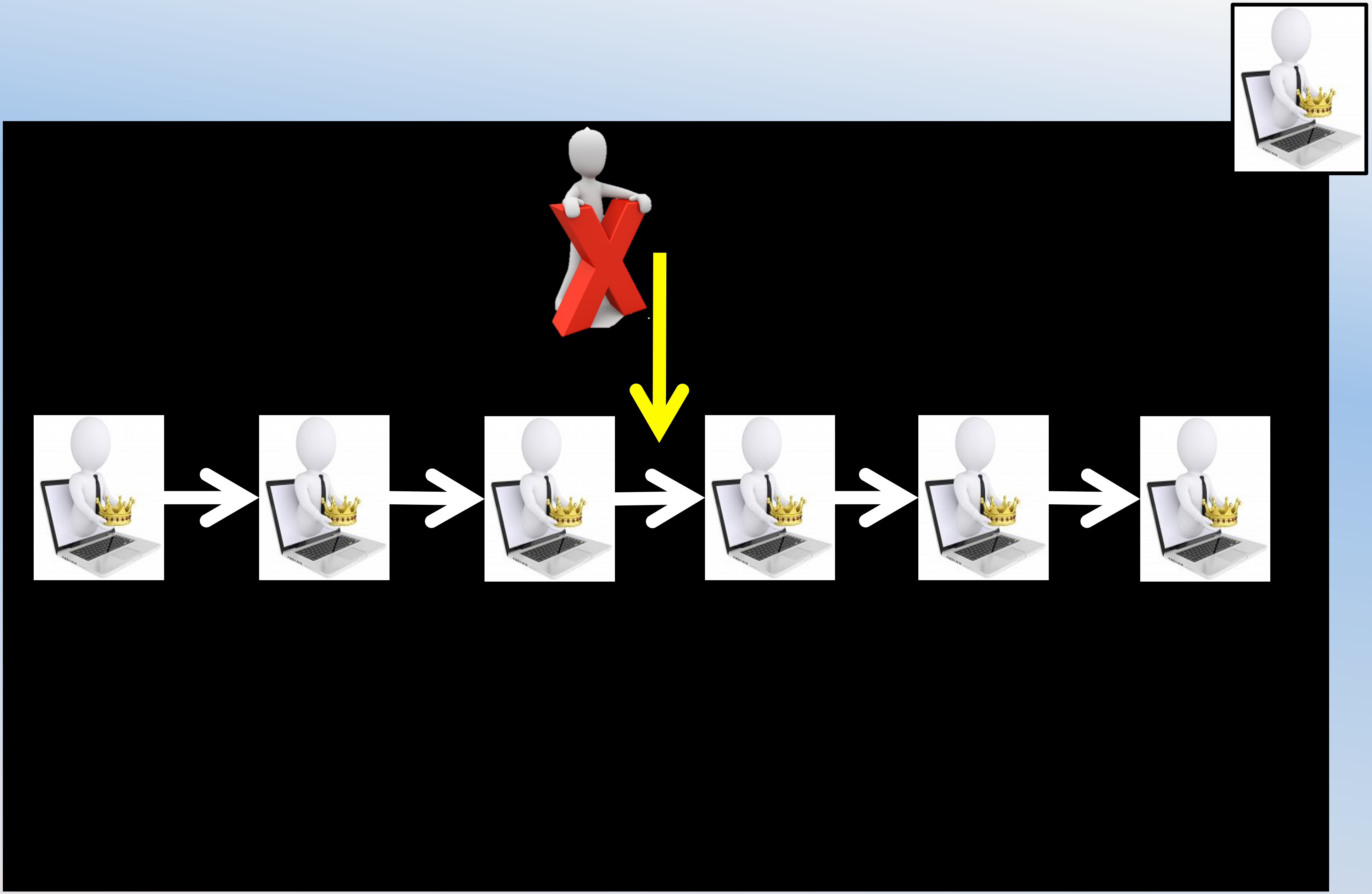
II. Command chaining

Encapsulation for the rescue

The command

```
public interface TestCommand {
    void run();
    void runCancel();

    CommandType getType();

    enum CommandType{
        API,
        UI,
        DB,
        CLI,
    }
}
```

http://static5.bigstockphoto.com/thumbs/3/6/4/small2/46370575.jpg

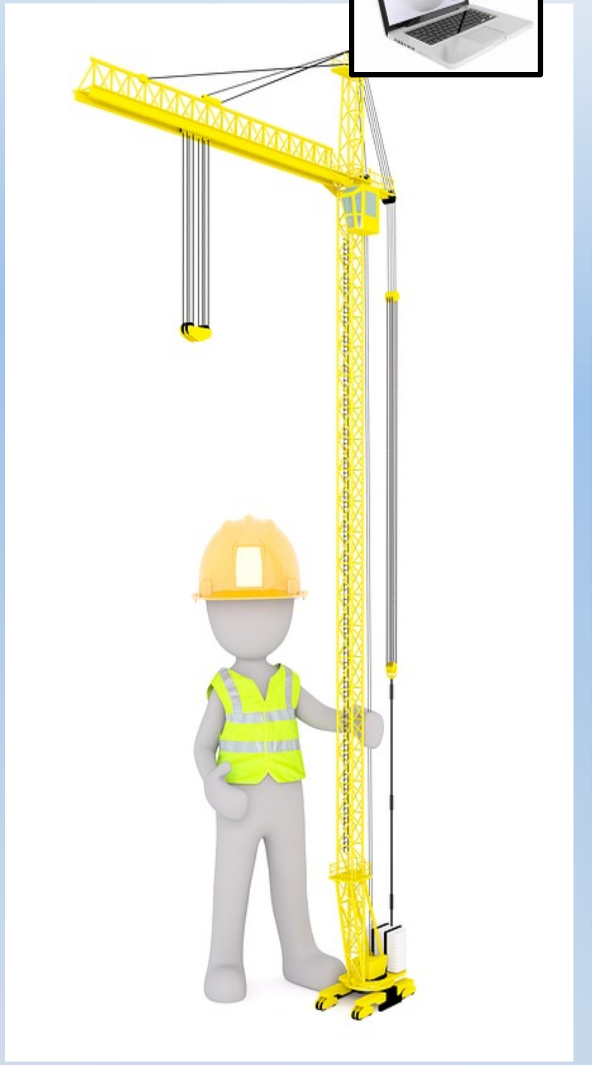


The Runner

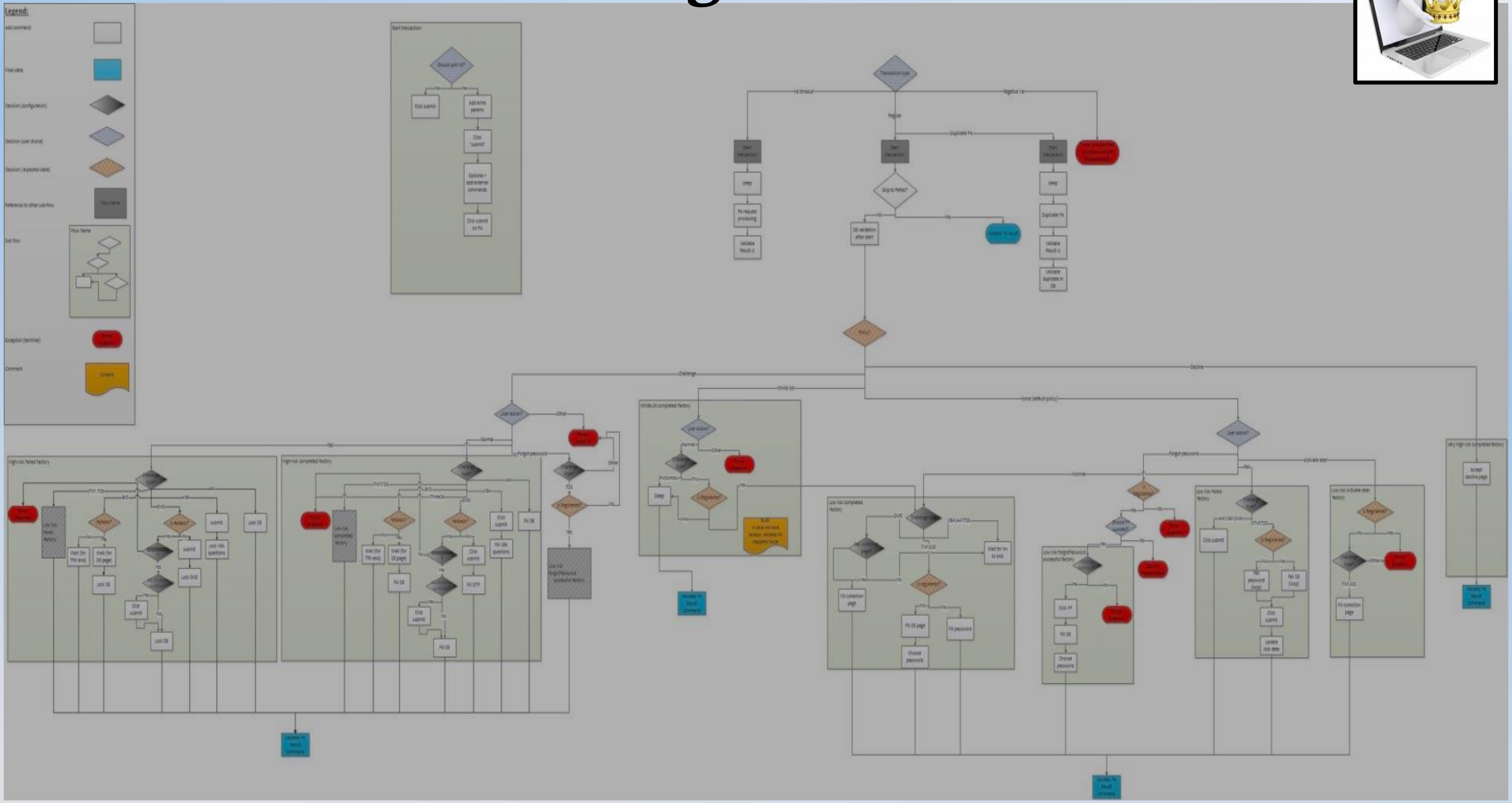
```
public class FlowRunner {
   public void run() {
      Boolean isCancelled = Boolean FALSE;
       for (ITestCommand command commands){
         if (isCancelled ||
      ShouldCancel(command.getType())) {
                  command.runCancel();
                  isCancelled=true;
                  continue;
         }
         command.run();
       }
   }
}
```

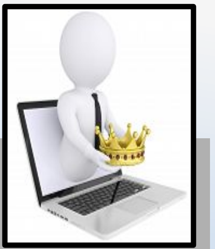
The Factory

```java
public static FlowRunner buildCheckout(TestData testData) {
    FlowRunner runner = testData.getFlowRunner();
    runner.add(new StartCheckoutCmd(testData));
    if (testData.isRegistered()) {
        runner.add(new FillUserNameAndPasswordCmd(testData));
    }
    else{
        runner.add(new ChooseNewCustomerCmd(testData));
        runner.add(new FillUserDetailsCmd(testData));
    }
    runner.add(new FillShippingAddressCmd(testData));
    runner.add(new CompleteBillingCmd(testData));
    return runner;
}
```



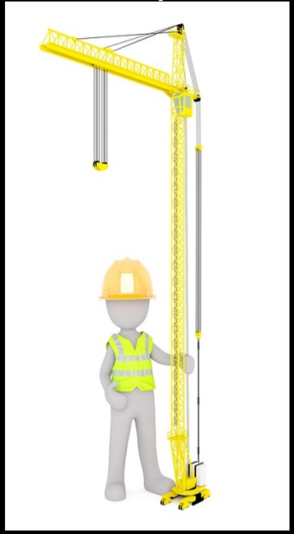
A word of warning





And together...

```
@Test
@Parameters({"userProvider","paymentMethod"})
public void testAuthorizationWithCommand(User user, PaymentMethod
paymentMethod) {
   TestData testData = defaultTestData(user);
   testData.setPaymentMethod(paymentMethod);
   CheckoutBuilder.build(testData).run();
  assertTrue(authorizationIsValid(paymentMethod));
}
```



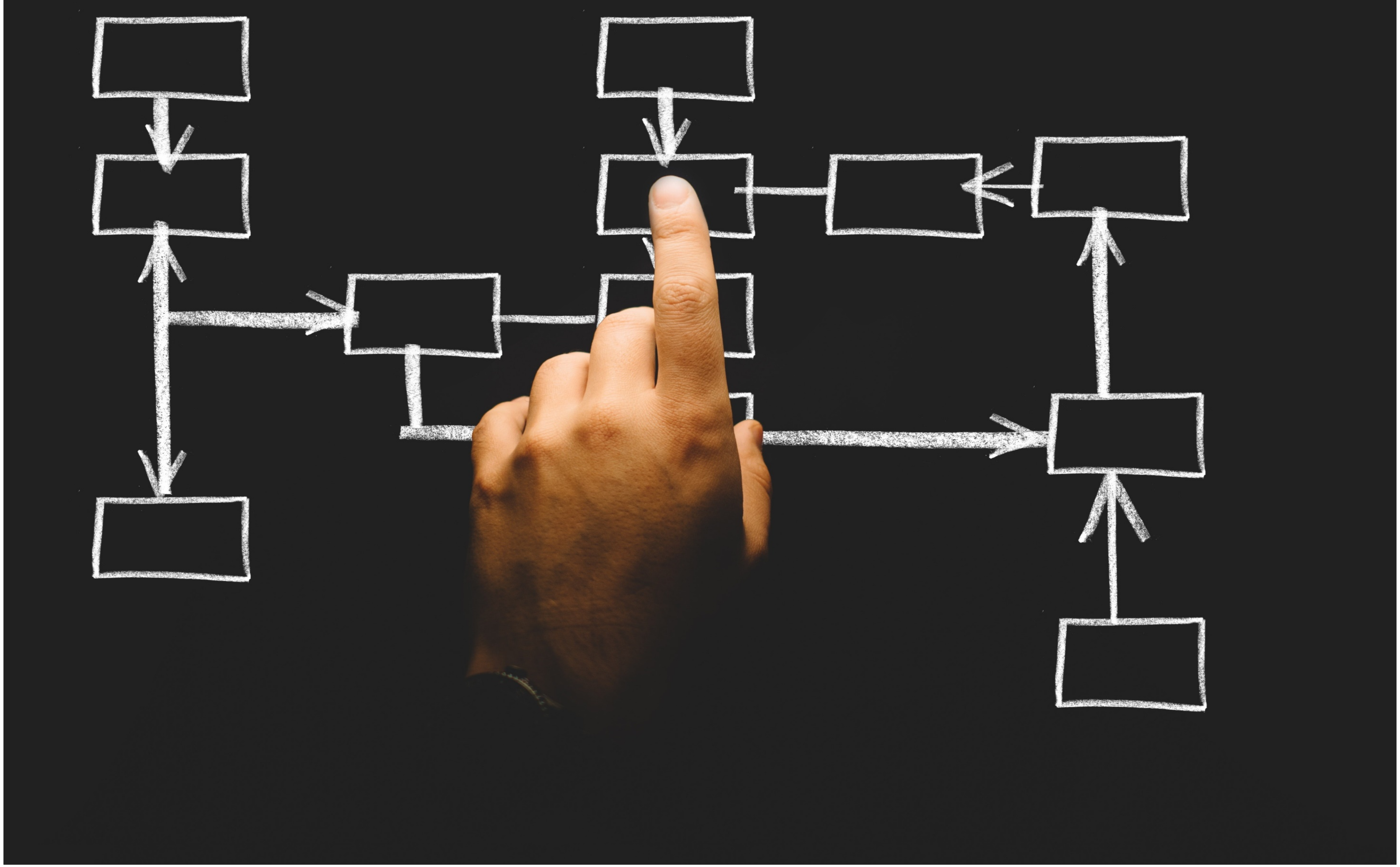
Clear Skies and rainbows

- Tests focus only on the data that matters
- Writing new tests got 3 times faster
- Easy to adapt the code to new functionality

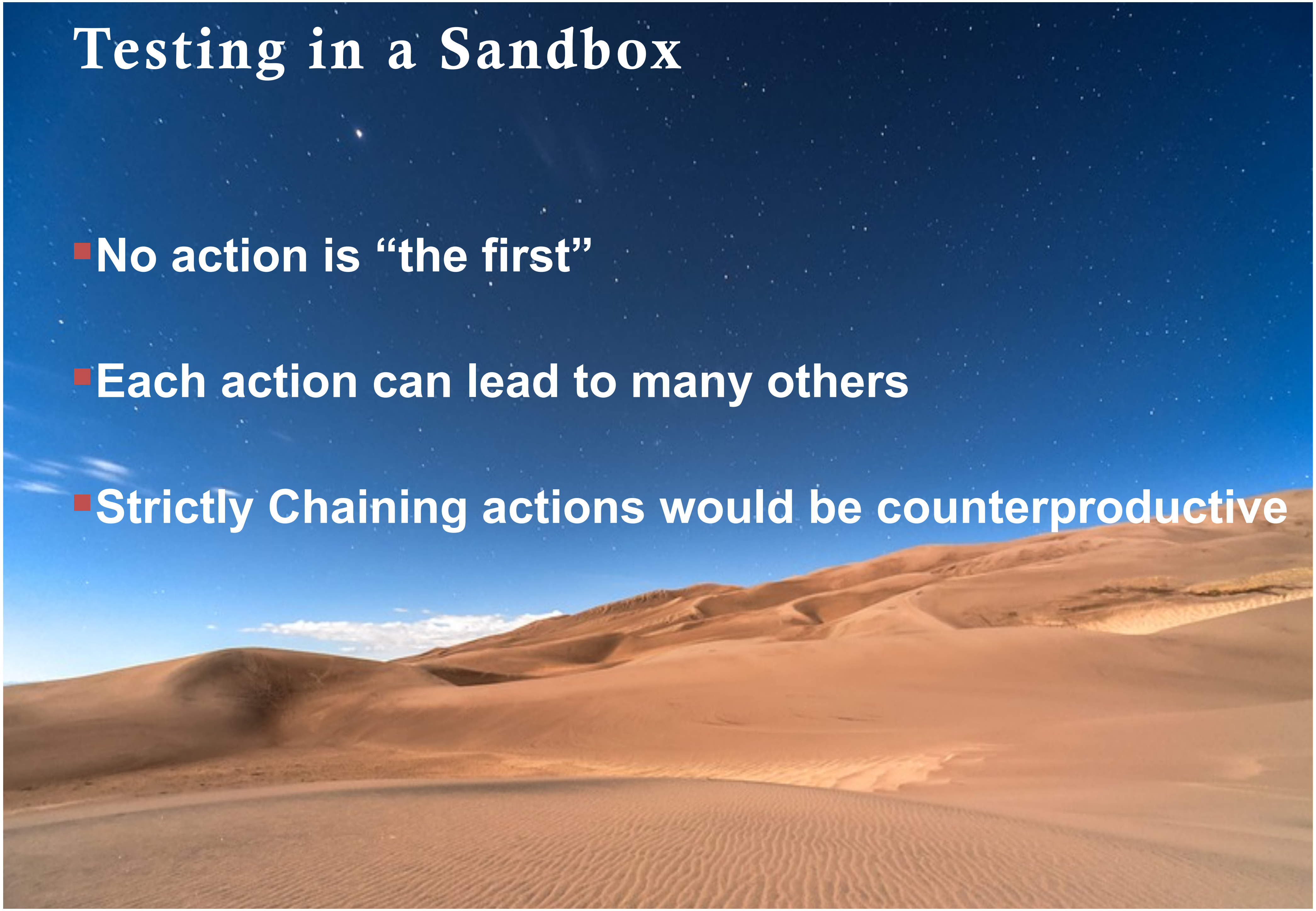
Yes, but…

- Debugging is more difficult
- The builder code is as complicated as your business logic
- Writing checks as “set”, ”build”, “run” is not the way we are used to think
- Not as easy to explain the infrastructure to new team members
- Real code is never as simple or clean as the examples.
- Only matches Linear flows

III. State transitioning

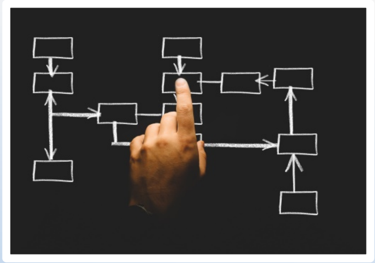


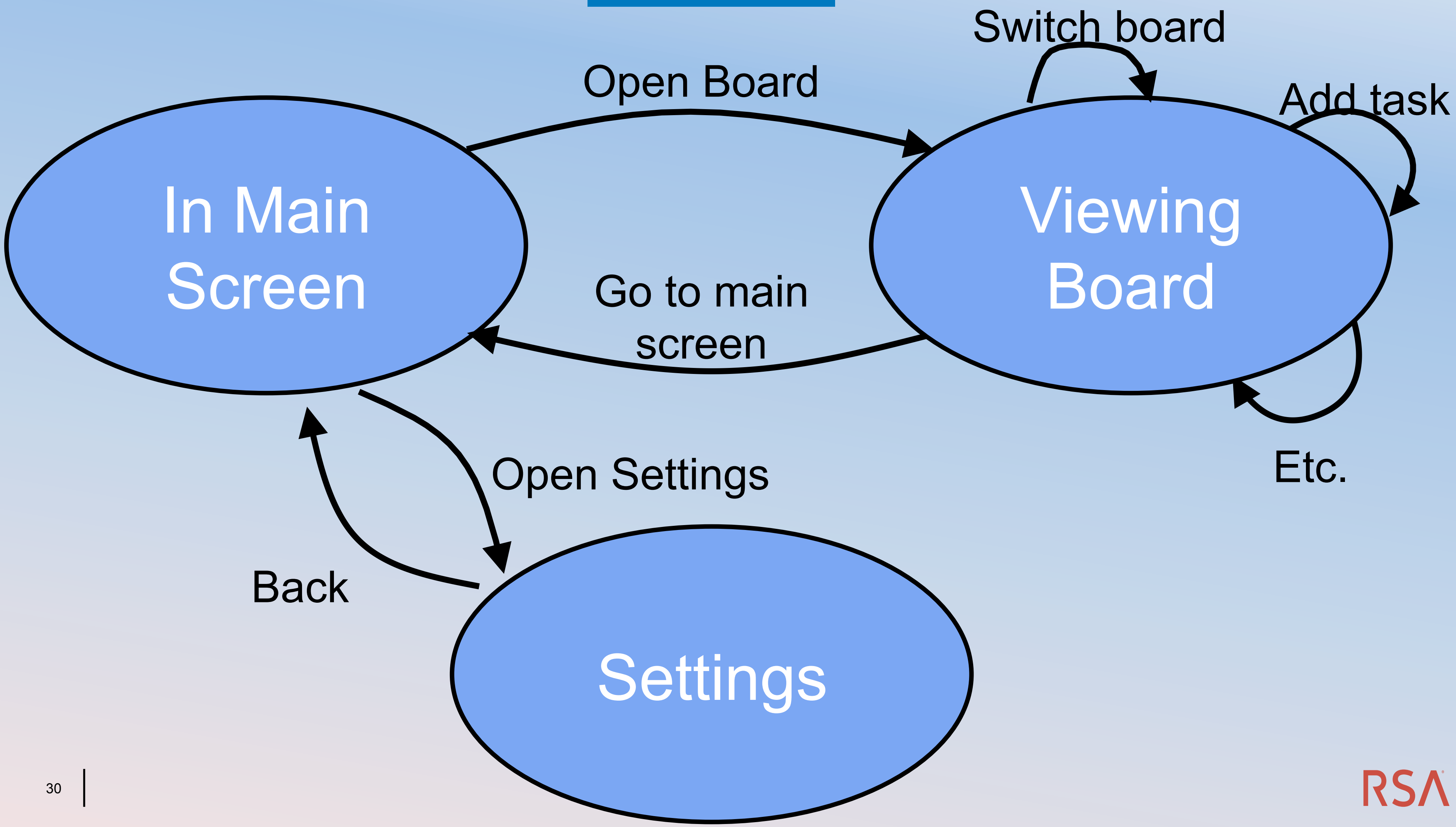
Testing in a Sandbox



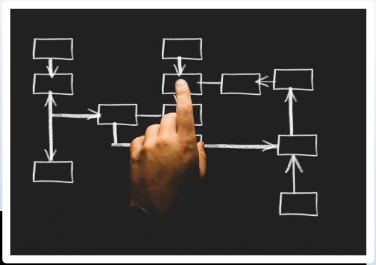
- No action is “the first”
- Each action can lead to many others
- Strictly Chaining actions would be counterproductive

Example



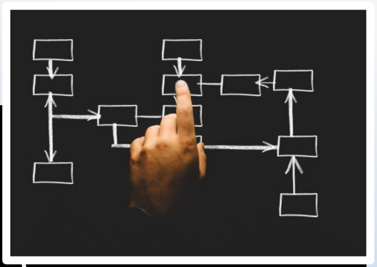

Switch board
Open Board
Add task
In Main Screen
Viewing Board
Go to main screen
Etc.
Open Settings
Back
Settings

State Interface



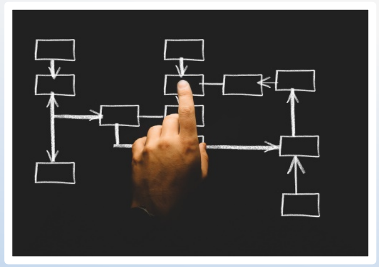
```
public interface ViewBoard {

   ViewBoard createNewTask(String listName, String taskContent);

   ViewBoard switchToBoard(String otherBoard);

   ViewBoard assertTaskExists(String listName, String taskContent);

   TrelloMainScreen goToHomeView();

}
```

And actual implementation



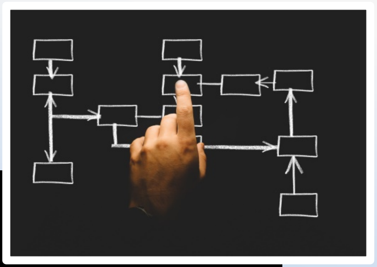
```
public class ViewBoardApiImpl implements ViewBoard {
     public ViewBoard createNewTask (String listName, String taskContent) {
       String taskId = TrelloRestClient.addTask(session, currentBoard, listName,
taskContent);
       Reporter.log("task “ + taskId + " created in list “ + listName ;
       if (null == taskId){
           throw new IllegalStateException("task not created");
       }
       return this;
   }
   public ViewBoard switchToBoard (String otherBoard) {
       null.currentBoard = otherBoard;
       return this;
   }
```

What do we gain?



- Fluent API at the business level
- IDE auto-complete
- Maximal flexibility
- Quickly identifying invalidated tests.
- Easy to add functionality

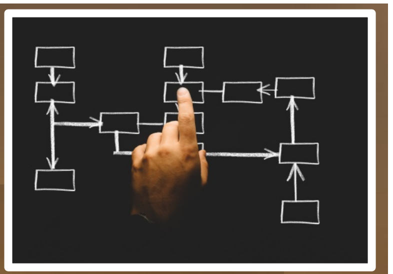
And it looks like this



```
@Test
@Parameters("user")
 public void TrelloTaskPersistence(User user){
   String taskContent = "do something";
   TrelloApp loginWith(user).openBoard(board1)
         .createNewTask("list name", taskContent)
         .switchToBoard(board2)
         .switchToBoard(board1)
         .assertTaskExists("list name", taskContent);
 }
```

RSA

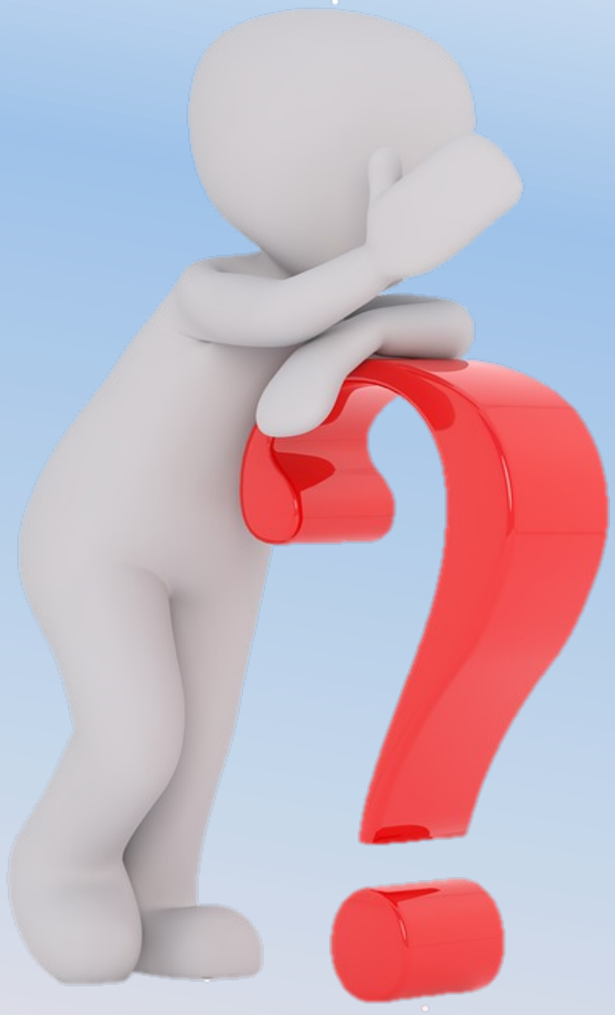
Pitfalls



- Write business actions, not technical ones
- Removing a state transition is expensive
- Tests become quite verbose
- Multiple ways to perform a transition can be challenging to include in the design
- State transitions based on “memory” can complicate interface design

Next thing tomorrow morning

- Observe:
  - Are your system tests speaking the business language?
  - Are they easy to write (and read)?
- Decide:
  - How you want your test functions to look like?
  - Don’t over-complicate stuff. Simple is better
- Act:
  - Add functionality incrementally
- Learn about design patterns and leverage them to your benefit



Further reading

- Gojko Adzik: How to implement UI testing without shooting yourself in the leg: https://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shooting-yourself-in-the-foot-2/

- Alex Schladebeck: How to stop hating UI tests: https://www.youtube.com/watch?v=iCqjCflx140

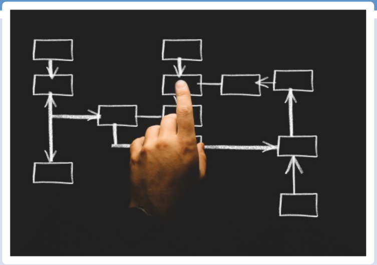
- Head first design patterns: http://shop.oreilly.com/product/9780596007126.do

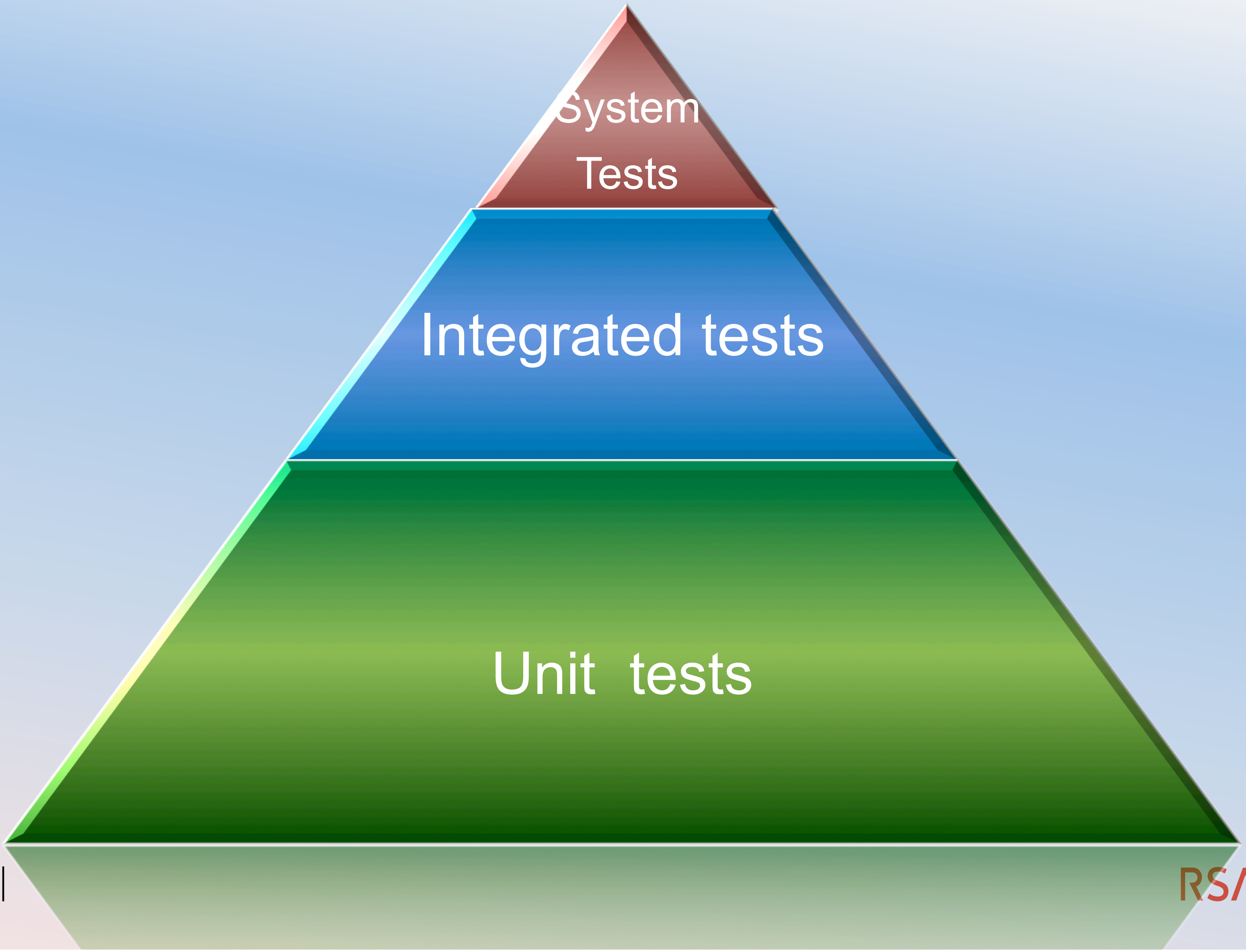
- Screenplay pattern https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay

Thank you

RSA

Which one suites me best?

	Utility functions	State transitions	Command chaining
			
Initial implementation	Trivial	Requires careful design	Complex
Maintenance	High	Changes in interfaces will break the tests	Easy to change, difficult to debug
Coding level required	Basic	Advanced	Advanced
Primary drawback	Copy\paste is the way forward	Very verbose	Difficult to explain
?When to use	I just want something that works	The user can wander freely between states	A main flow in the application with many customizations


System
Tests
Integrated tests
Unit tests

Choosing is difficult

```
@Test
@Parameters({"userProvider","paymentMethod"})
public void testAuthorization(User user,PaymentMethod paymentMethod){
    switch  paymentMethod){
        case CREDIT:
            CheckoutFlows.checkOutWithCreditCard(user,driver);
            break;
        case DEBIT:
            CheckoutFlows.checkoutWithDebitCard(user,driver);
            break;
        case PAYPAL:
            CheckoutFlows checkoutWithPayPal(user,driver);
            break;
    }
assertTrue(authorizationIsValid(paymentMethod , "Authorization was not completed properly");
}
```