Automation Testing

What is Automation Testing?

- Automation Testing is a software testing technique that performs using special automated testing software tools to execute a test case suite.
- On the contrary, Manual Testing is performed by a human sitting in front of a computer carefully executing the test steps.
- The automation testing software can also enter test data into the System Under Test, compare expected and actual results and generate detailed test reports. Software Test Automation demands considerable investments of money and resources.

Why Test Automation?

- Test Automation is the best way to increase the effectiveness, test coverage, and execution speed in software testing. Automated software testing is important due to the following reasons:
  - Manual Testing of all workflows, all fields, all negative scenarios is time and money consuming
  - It is difficult to test for multilingual sites manually
  - Test Automation in software testing does not require Human intervention. You can run automated test unattended (overnight)
  - Test Automation increases the speed of test execution
  - Automation helps increase Test Coverage
  - Manual Testing can become boring and hence error-prone.

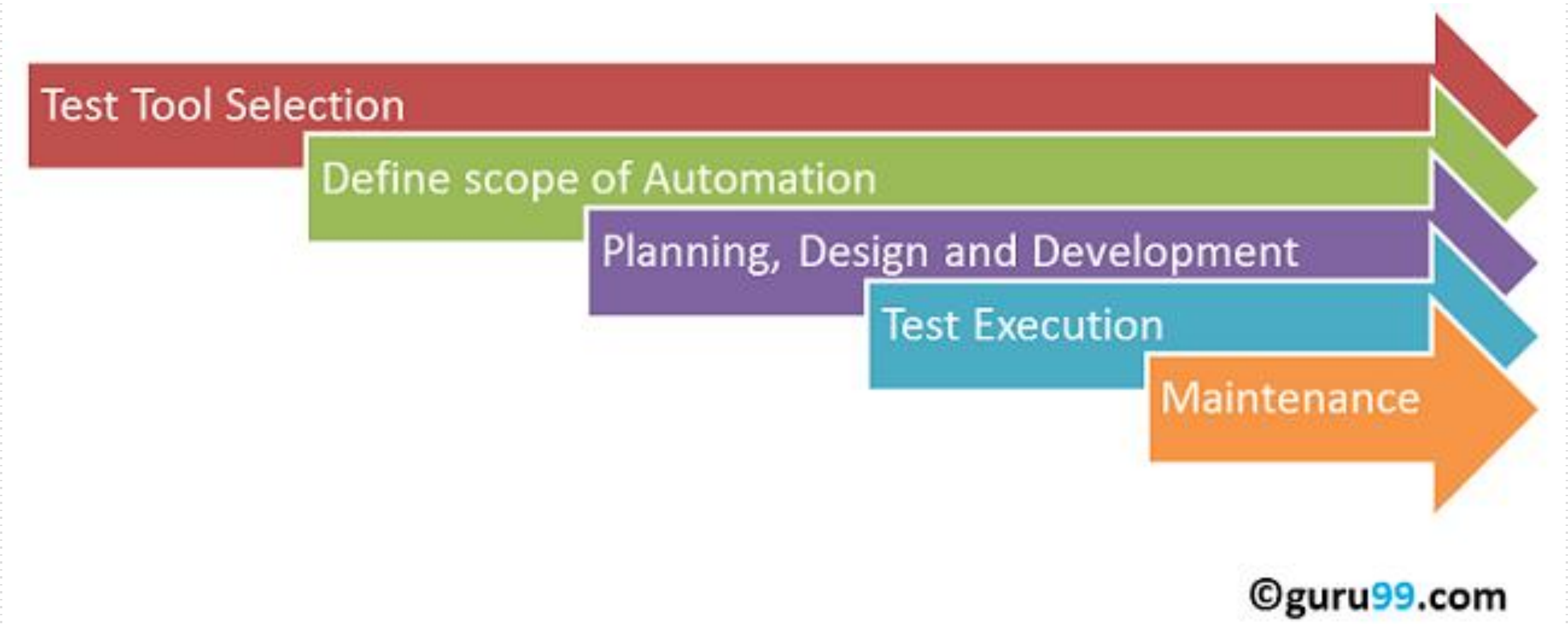
Which Test Cases to Automate?

- Test cases to be automated can be selected using the following criterion to increase the automation ROI
  - High Risk – Business Critical test cases
  - Test cases that are repeatedly executed
  - Test Cases that are very tedious or difficult to perform manually
  - Test Cases which are time-consuming

Which Test Cases not to Automate?

- The following category of test cases are not suitable for automation:
  - Test Cases that are newly designed and not executed manually at least once
  - Test Cases for which the requirements are frequently changing
  - Test cases which are executed on an ad-hoc basis.

Automated Testing Process



Test tool selection

- Test Tool selection largely depends on the technology the Application Under Test is built on.
  - Platform
  - Programming languages
  - Devices
  - ….

Define the scope of Automation

- The scope of automation is the area of your Application Under Test which will be automated. Following points help determine scope:
  - The features that are important for the business
  - Scenarios which have **a large amount of data**
  - **Common functionalities** across applications
  - Technical feasibility
  - The extent to which business components are reused
  - **The complexity** of test cases
  - Ability to use the same test cases for cross-browser testing

Planning, Design, and Development

- During this phase, you create an Automation strategy & plan, which contains the following details
  - Automation tools selected
  - Framework design and its features
  - In-Scope and Out-of-scope items of automation
  - Automation testbed preparation
  - Schedule and Timeline of scripting and execution
  - Deliverables of Automation Testing

Test Execution

- Automation Scripts are executed during this phase. The scripts need input test data before there are set to run. Once executed they provide detailed test reports.
- Execution can be performed using the automation tool directly or through the Test Management tool which will invoke the automation tool.

Test Automation Maintenance Approach

- Test Automation Maintenance Approach is an automation testing phase carried out to test whether the new functionalities added to the software are working fine or not.
- Maintenance in automation testing is executed when new automation scripts are added and need to be reviewed and maintained in order to improve the effectiveness of automation scripts with each successive release cycle.

Types of automation frameworks

- Data Driven Automation Framework
- Keyword Driven Automation Framework
- Modular Automation Framework
- Hybrid Automation Framework

Test Automation benefits

- 70% faster than the manual testing
- Wider test coverage of application features
- Reliable in results
- Ensure Consistency
- Saves Time and Cost
- Improves accuracy
- Human Intervention is not required while execution
- Increases Efficiency
- Better speed in executing tests
- Re-usable test scripts
- Test Frequently and thoroughly
- More cycle of execution can be achieved through automation
- Early time to market

Choose an Automation Tool

- Environment Support
- Ease of use
- Testing of Database
- Object identification
- Image Testing
- Error Recovery Testing
- Object Mapping
- Scripting Language Used
- Support for various types of test – including functional, test management, mobile, etc…
- Support for multiple testing frameworks
- Easy to debug the automation software scripts
- Ability to recognize objects in any environment
- Extensive test reports and results
- Minimize training cost of selected tools

Types of Automated Testing

- Smoke Testing
- Unit Testing
- Integration Testing
- Functional Testing
- Keyword Testing
- Regression Testing
- Data Driven Testing
- Black Box Testing
- API testing

Automation Based on the Type of Testing

- Automation of Functional Tests:
  - Automating these mean writing scripts to validate the business logic and the functionality expected from the application.
- Automation of Non-Functional Tests:
  - These are the requirements related to performance, security, databases, etc. These requirements can remain constant or can be scaled as per the size of the software.

Automation Based on the Phase/Level of Testing

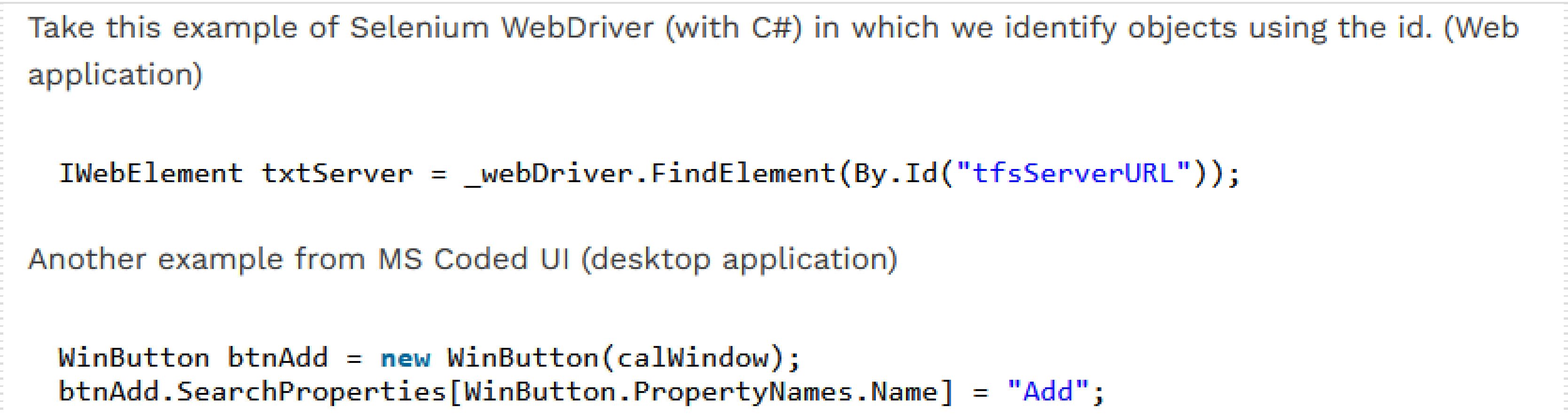
- **Automation of Unit Tests:**
  - These tests are run during the development phase itself, ideally by the dev after the completion of development and before handing over the system to the testers for testing.
- **Automation of API Tests:**
  - API tests are run during the integration phase. These may be run by the development or testing team and can be run before or after the UI layer is built for the application. These tests target the testing based on the request and response on which the application is built.
- **Automation of UI based tests:**
  - UI Based tests are run during the test execution phase. These are specifically run by the testers and are run only once before the UI of the application is handed over to them. These test the functionality and business logic of the application from the front end of the application.

3A’s

- In test automation, we write scripts. Scripting is basically about three ‘A’s:
  1. ARRANGEMENT
  2. ACTION
  3. ASSERTION

#1. ARRANGEMENT or Object Identification

- We identify objects (buttons, dropdowns etc.) either by their ids, names or by their Window Titles etc…
- In case of web application, we identify by user ID, or By XPath or By CSS or By Class Name etc.
- If nothing works, we then identify objects by using mouse coordinates (But it is not a reliable method of object identification)


Take this example of Selenium WebDriver (with C#) in which we identify objects using the id. (Web application)

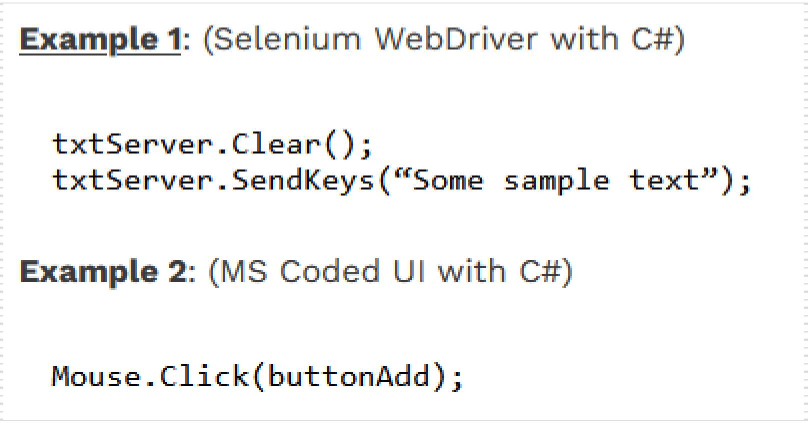
```
IWebElement txtServer = _webDriver.FindElement(By.Id("tfsServerURL"));
```

Another example from MS Coded UI (desktop application)

```
WinButton btnAdd = new WinButton(calWindow);
btnAdd.SearchProperties[WinButton.PropertyNames.Name] = "Add";
```

#2. ACTION on the Identified Object

- When the objects are identified, we perform some kind of actions on it either by mouse or by keyboard.
- For example, either we click, or we double-click, or we mouse hover over it or sometimes we drag-drop.
- Sometimes we write on text boxes. So any kind of action we perform on these objects are covered in this second step.



Example 1: (Selenium WebDriver with C#)

```
txtServer.Clear();
txtServer.SendKeys("Some sample text");
```

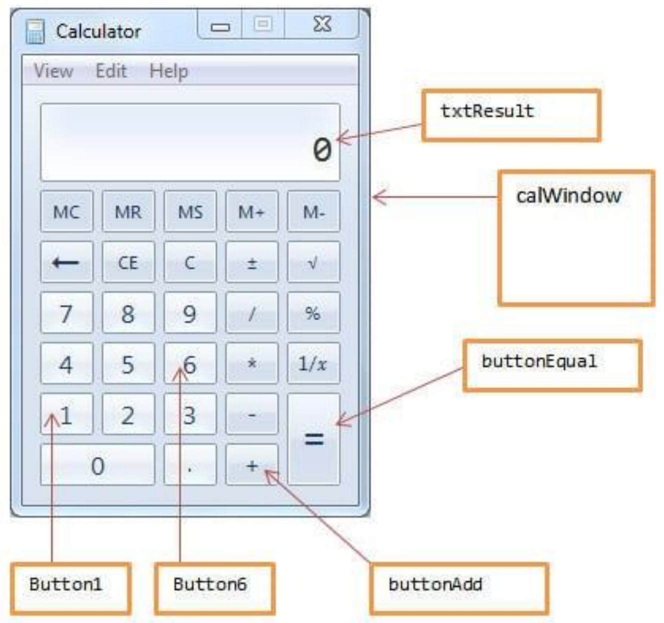
Example 2: (MS Coded UI with C#)

```
Mouse.Click(buttonAdd);
```

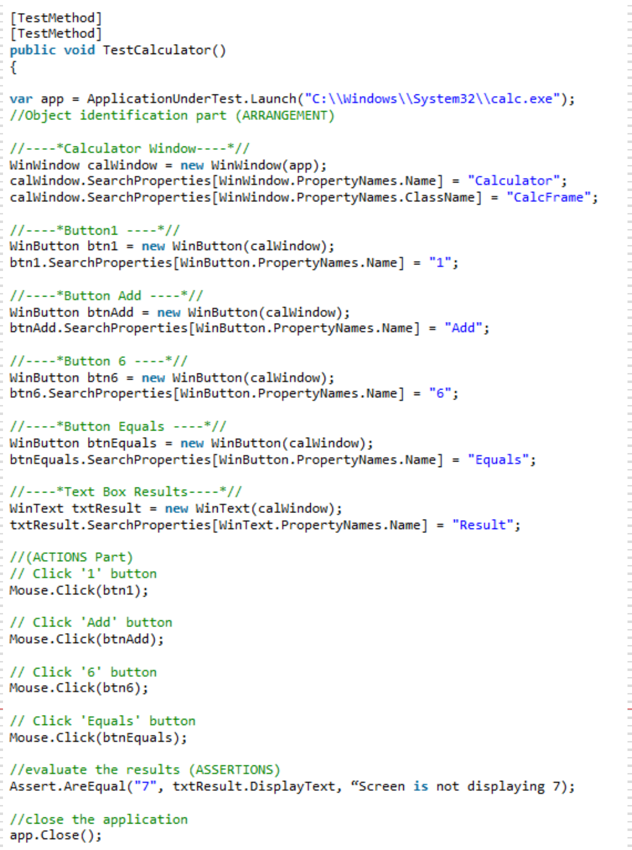
#3. ASSERTION

- The assertion is basically checking the object with some expected result.

Now take a look at a complete script which contains all these steps. The script will open a calculator, press 1 + 6 and then check whether screen shows 7 or not.



Example to learn Test Automation Frameworks



```
[TestMethod]
[TestMethod]
public void TestCalculator()
{

var app = ApplicationUnderTest.Launch("C:\\Windows\\System32\\calc.exe");
//Object identification part (ARRANGEMENT)

//----*Calculator Window----*//
WinWindow calWindow = new WinWindow(app);
calWindow.SearchProperties[WinWindow.PropertyNames.Name] = "Calculator";
calWindow.SearchProperties[WinWindow.PropertyNames.ClassName] = "CalcFrame";

//----*Button1 ----*//
WinButton btn1 = new WinButton(calWindow);
btn1.SearchProperties[WinButton.PropertyNames.Name] = "1";

//----*Button Add ----*//
WinButton btnAdd = new WinButton(calWindow);
btnAdd.SearchProperties[WinButton.PropertyNames.Name] = "Add";

//----*Button 6 ----*//
WinButton btn6 = new WinButton(calWindow);
btn6.SearchProperties[WinButton.PropertyNames.Name] = "6";

//----*Button Equals ----*//
WinButton btnEquals = new WinButton(calWindow);
btnEquals.SearchProperties[WinButton.PropertyNames.Name] = "Equals";

//----*Text Box Results----*//
WinText txtResult = new WinText(calWindow);
txtResult.SearchProperties[WinText.PropertyNames.Name] = "Result";

//(ACTIONS Part)
// Click '1' button
Mouse.Click(btn1);

// Click 'Add' button
Mouse.Click(btnAdd);

// Click '6' button
Mouse.Click(btn6);

// Click 'Equals' button
Mouse.Click(btnEquals);

//evaluate the results (ASSERTIONS)
Assert.AreEqual("7", txtResult.DisplayText, “Screen is not displaying 7);

//close the application
app.Close();
```

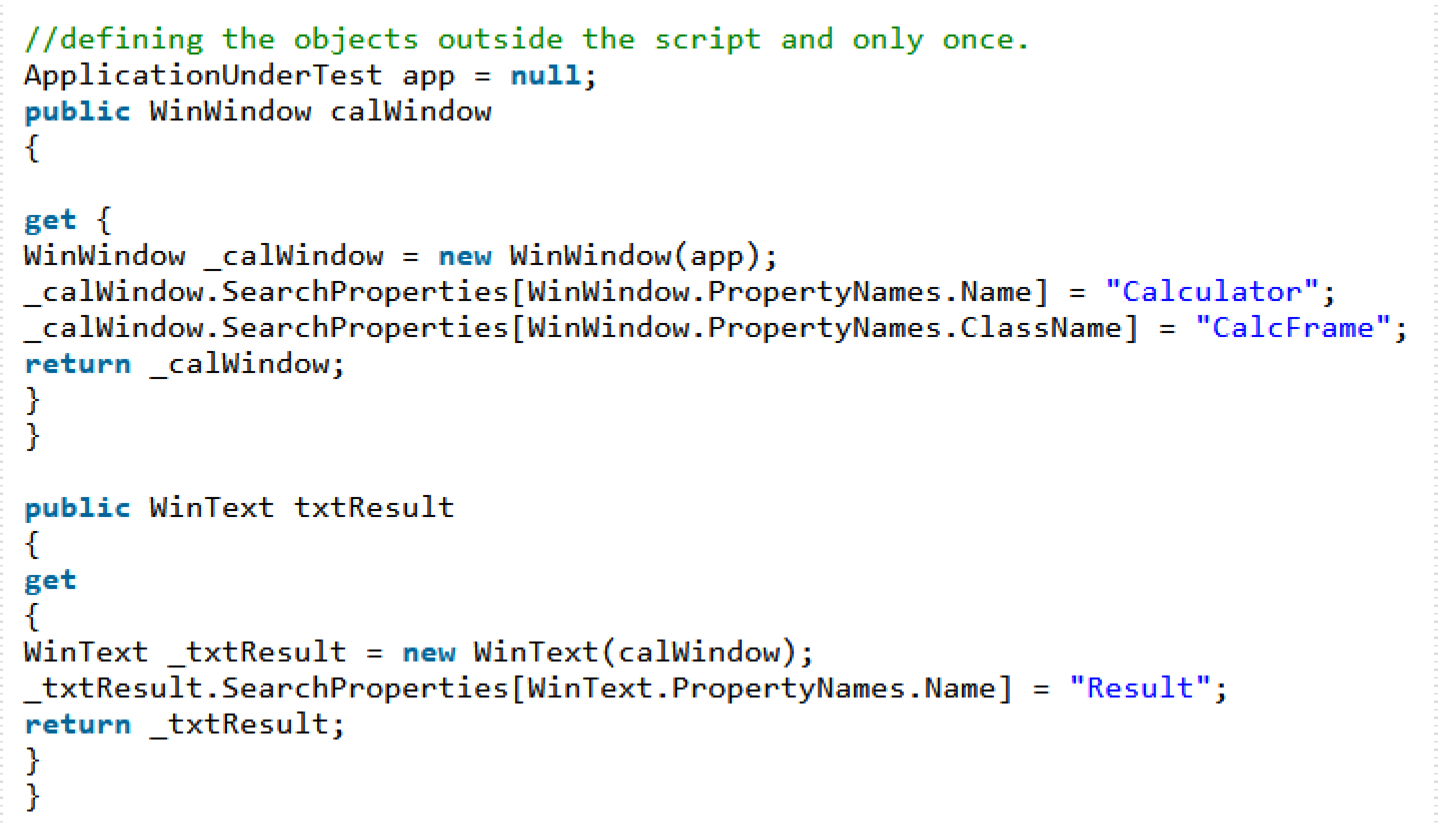
What’s wrong with that script?

- This script does not allow easy maintenance.
  - Take the example of calculator again, if we have to write test cases of each function of the calculator, there will be many test cases.
  - If there are 10 test cases and in each test, we have to define the same object, then if any change occurs in the name or id of the object, we have to change the object identification part in 10 test cases.

DRY principle

- So we have to improve this test case. We should follow the famous DRY principle in our coding.
- DRY stands for “Do not Repeat Yourself”. We should write the object identification part in such a manner that the object should be identified only in one place and should be called everywhere.

Define the objects



```
//defining the objects outside the script and only once.
ApplicationUnderTest app = null;
public WinWindow calWindow
{

get {
WinWindow _calWindow = new WinWindow(app);
_calWindow.SearchProperties[WinWindow.PropertyNames.Name] = "Calculator";
_calWindow.SearchProperties[WinWindow.PropertyNames.ClassName] = "CalcFrame";
return _calWindow;
}
}

public WinText txtResult
{
get
{
WinText _txtResult = new WinText(calWindow);
_txtResult.SearchProperties[WinText.PropertyNames.Name] = "Result";
return _txtResult;
}
}
```

```
//making functions for similar kind of tasks
public void ClickButton(string BtnName)
{
WinButton button = new WinButton(calWindow);
button.SearchProperties[WinButton.PropertyNames.Name] = BtnName ;
Mouse.Click(button);
}

public void AddTwoNumbers(string number1, string number2)
{
ClickButton(number1);
ClickButton("Add");
ClickButton(number2);
ClickButton("Equals");
}
```

```
//Test case becomes simple and easy to maintain.
[TestMethod]

public void TestCalculatorModular()
{
app = ApplicationUnderTest.Launch("C:\\Windows\\System32\\calc.exe");

//do all the operations
AddTwoNumbers("6", "1");

//evaluate the results
Assert.AreEqual("7", txtResult.DisplayText, “screen is not displaying 7”);

//close the application
app.Close();

}
```

5 popular frameworks in test automation:

1. Linear
2. Modularity
3. Data Driven
4. Keyword Driven
5. Hybrid

#1. Linear Framework:

- Characteristics
  - Everything related to a script is defined inside the scripts.
  - Does not care about abstraction and code duplication
  - Record and playback normally generate linear code
  - Easy to get started
  - Maintenance Nightmare.
- This framework can be used in small-scale projects where there are not many UI screens.

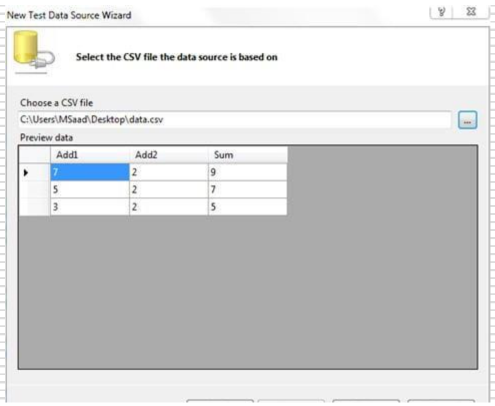
#2. Modularity Framework

- Characteristics
  - The objects are defined once and reusable in all test methods.
  - Small and to-the-point methods are created for individual functionalities
  - The test case is the collection of these small methods and reusable objects
  - This allows us to write maintainable code.
- This results in easier maintenance. If any change occurs in the calculator UI, we have to change only in functions. Our scripts will remain intact. This framework is highly used in automation.

#3. Data Driven Framework

- Characteristics:
  - Test Data (input and output values) are separated from the script and stored in External Files. This could be a.CSV file, an Excel spreadsheet or a Database.
  - When the script is executed, these values are picked from external files, stored in variables and replace the hard-coded values if present.
  - Really useful in places where the same test case has to be run with different inputs.

Examples


New Test Data Source Wizard

Select the CSV file the data source is based on

Choose a CSV file

C:\Users\MSaad\Desktop\data.csv

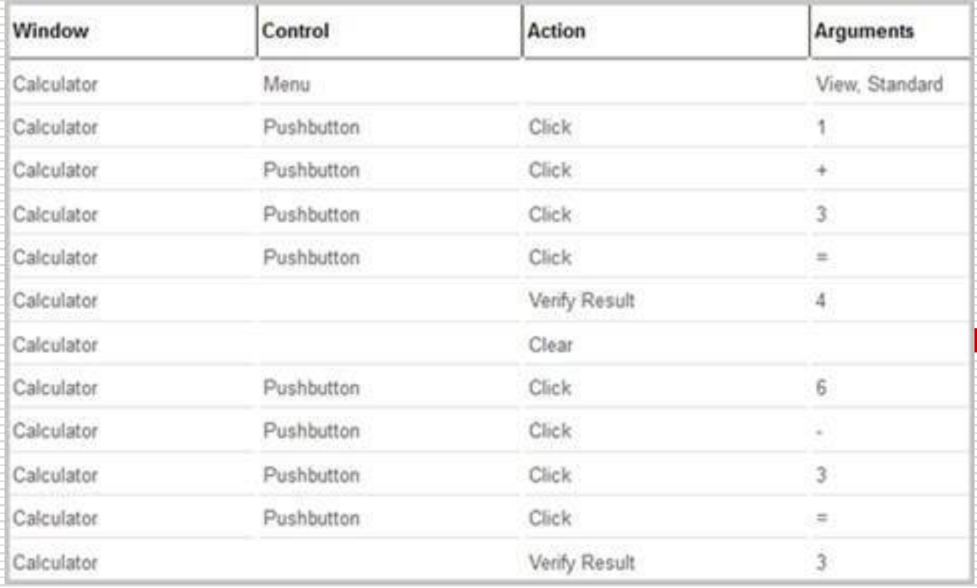
Preview data

Add1	Add2	Sum
7	2	9
5	2	7
3	2	5

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data

public void TestCalculatorDataDrivsen()
{
app = ApplicationUnderTest.Launch("C:\\Windows\\System32\\calc.exe");

//do all the operations
AddTwoNumbers(FromCSV.ADD1, FromCSV.ADD2);

//evaluate the results
Assert.AreEqual(FromCSV.Sum, txtResult.DisplayText);

//close the application
app.Close();

}
```

#4. Keyword-Driven Framework

- Characteristics:
  - Both data and actions are defined outside the script.
  - It required the development of keywords for different types of actions.
  - The functionality which we have to test is written in a step-by-step manner in tabular form using the keywords we develop and the test data. We store this table in external files just like data driven framework.
  - The script will parse this table and perform the corresponding actions.
  - Allows manual tester who does not know about coding to be part of automation to some extent.



Window	Control	Action	Arguments
Calculator	Menu		View, Standard
Calculator	Pushbutton	Click	1
Calculator	Pushbutton	Click	+
Calculator	Pushbutton	Click	3
Calculator	Pushbutton	Click	=
Calculator		Verify Result	4
Calculator		Clear	
Calculator	Pushbutton	Click	6
Calculator	Pushbutton	Click	-
Calculator	Pushbutton	Click	3
Calculator	Pushbutton	Click	=
Calculator		Verify Result	3

```
[TestMethod]
public void TestCalculator()
{
app = ApplicationUnderTest.Launch("C:\\Windows\\System32\\calc.exe");

Table tb = ReadFromExcel();
Foreach(WinRow row in tb)
{

WinCell Window = row.Cells[“Window”];
WinCell Control = row.Cells[“Control”];
WinCell Action = row.Cells[“Action”];
WinCell Arguments = row.Cells[“Arguments”];
UITestControl c = GetControl(Control.Text,Window.Text);

If(Action.Text == “Click”)
Mouse.Click (c);

If (Action.Text == “Clear”)
c.Clear();

if(Action.Text == “Verify Result”)
Assert.AreEqual(c.Text, Arguments.Text)

//….and so on
}
```

#5. Hybrid Test Automation Framework

- Characteristics:
  - The combination of two or more of the above techniques, taking from their strengths and minimizing their weaknesses.
  - The framework can use the modular approach along with either data-driven or keyword-driven framework.
  - The framework can use scripts to perform some tasks that might be too difficult to implement in a pure keyword driven approach.

GUI Automation Testing & Regresstion Testing

Regresstion

“Repeat testing after changes.”

Regresstion Testing – What ?

- Tag line: *"Repeat testing after changes."*
  - But scratch the surface and you find three fundamentally different visions:
    1. *Procedural:* Run the same tests again
    2. *Risk-oriented:* Expose errors caused by change
    3. *Refactoring support*: Help the programmer discover implications of her code changes.
- Fundamental question or goal
  - Good regression testing gives clients confidence that they can change the product (or product environment).

Why Are Regression Testing Weak?

- Does the same thing over and over
- Most defects are found during test creation
- Software doesn’t break or wear out
- Any other test is equally likely to stumble over unexpected side effects
- Automation reduces test variability
- Only verifies things programmed into the test

1.Procedural regression testing

- Tag line
  - *"Run the same tests again"*
- Paradigmatic cases
  - Manual, scripted regression testing
  - Automated GUI regression testing
  - Smoke testing (manual or automated)

Procedural RT - Benefits

- The tests exist already (no need for new design, or new implementation, but there will be maintenance cost)
- Many regulators and process inspectors like them
- Because we are investing in re-use, we can afford to take the time to craft each test carefully, making it more likely to be powerful in future use
- This is the dominant paradigm for automated testing, so it is relatively easy to justify and there are lots of commercial tools
- Implementation of automated tests is often relatively quick and easy (though maintenance can be a nightmare)

Procedural regression testing

- Blind spots / weaknesses
  - Anything not covered in the regression series.
  - Repeating the same tests means not looking for the bugs that can be found by other tests.
  - Pesticide paradox
  - Low yield from automated regression tests
  - Maintenance of this standard list can be costly and distracting from the search for defects.

2.Risk-oriented regression testing

- Tag line
  - *"Test after changes."*
- Fundamental question or goal
  - Manage the risks that

    (a) a bug fix didn't fix the bug or

    (b) the fix (or other change) had a side effect.
- Paradigmatic case(s)
  - <u>Bug regression</u> (Show that a bug was not fixed)
  - <u>Old fix regression</u> (Show that an old bug fix was broken)
  - <u>General functional regression</u> (Show that a change caused a working area to break.)

Risk-oriented regression testing (cont.)

- In this approach, we might **re-use old tests** or **create new ones**.
- Often, we **retest an area** or function with tests of **increasing power** (perhaps by combining them with other functions).
- The focus of the technique is on testing for **side effects of change**, not the inventory (tóm tắt) of old tests.

Risk-oriented RT - Examples

- Here are examples of a few common ways to test a program more harshly while retesting in the same area:
  - Do more iterations (one test hits the same function many times).
  - Do more combinations (interactions among variables, including the function under test's variables).
  - Do more things (sequences of functions that include the function under test).

Risk-oriented RT – Examples (cont.)

- Methodically cover the code (all N-length sequences that include the function under test; all N-wide combinations that include the function under test's variables and variables that are expected to interact with or constrain or be constrained by the function under test).
- Look for specific errors (such as similar products' problems) that involve the function under test.
- Try other types of tests, such as scenarios, that involve the function under test.
- Try to break it (take a perverse view, get creative).

3.Refactoring support: Change detectors

- Tag line
  - *"Change detectors"*
- Fundamental question or goal
  - Support refactoring: Help the programmer discover implications of her code changes.
- Paradigmatic case(s)
  - ***Test-driven development using glass-box testing tools like junit, httpunit, and fit…***

Note

- The programmer creates these tests and runs them every time she compiles the code.
- If a test suite takes more than 2 minutes, the programmer might split tests into 2 groups (tests run at every compile and tests run every few hours or overnight).
- The intent of the tests is to exercise every function in interesting ways, so that when the programmer refactors code, she can quickly see
  - (a) what would break if she made a change to a given variable, data item or function or
  - (b) what she did break by making the change.

Refactoring support: How ?

- In the unit test situation, the programmer (not an independent tester) writes the tests, typically before she writes the code. The testing focuses the programming, yielding better code in the first place.

Refactoring support: How ?

- In the unit test case, when the programmer makes a change that has a side-effect, she immediately discovers the break and fixes it → There is no communication cost.
- You don't have (as you would in black box testing) a tester who discovers a bug, replicates it, reports it, and then a project manager who reads the report, maybe a triage team who study the bug and agree it should be fixed, a programmer who has to read the report, troubleshoot the problem, fix it, file the fix, a tester who has to retest the bug to determine that the fix really fixed the problem and then close the bug. All labor-hours considered, this can easily cost 4 hours of processing time, compared to a few minutes to fix a bug discovered at the unit level.

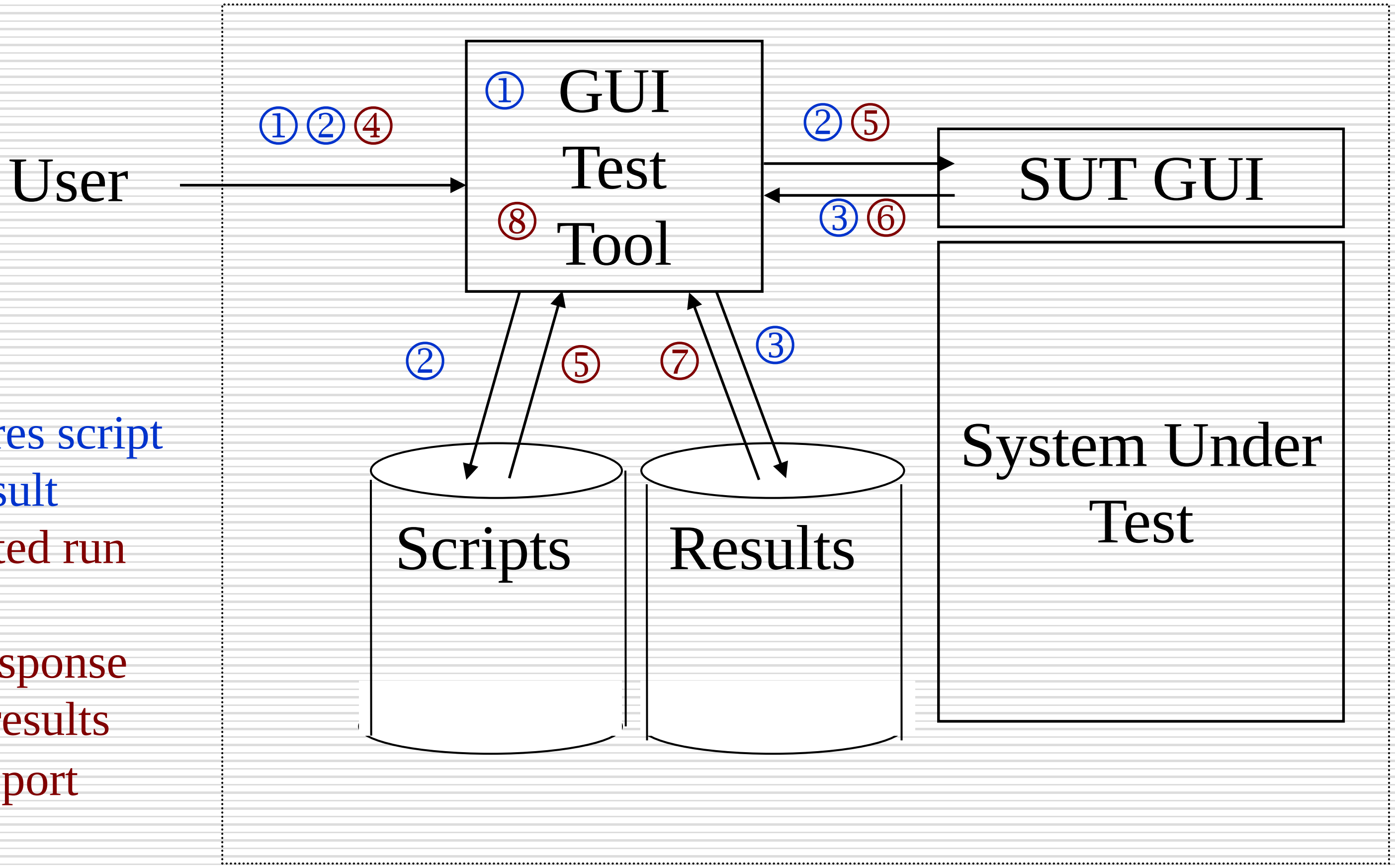
Automating GUI regression testing

Good or bad ?

Automating GUI regression testing

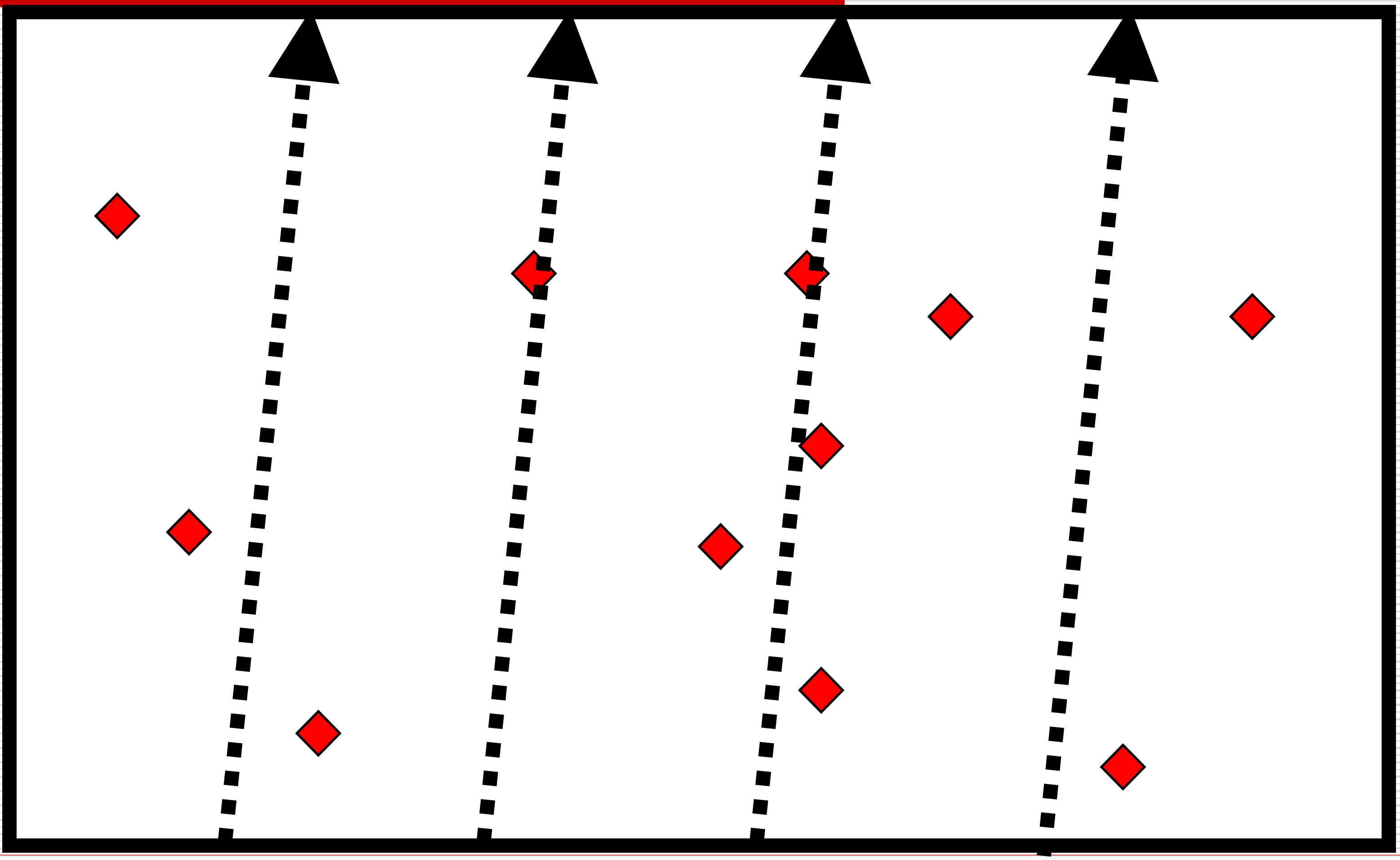
- This is the most commonly discussed automation approach:
  1. conceive and create a test case
  2. run it and inspect the output results
  3. if the program fails, report a bug and try again later
  4. if the program passes the test, save the resulting outputs
  5. in future tests, run the program and compare the output to the saved results
  6. report an exception whenever the current output and the saved output don’t match

A GUI Regression Test Model



① Launch tool
② Test; tool captures script
③ Test; capture result
④ Launch automated run
⑤ Play script
⑥ Capture SUT response
⑦ Read recorded results
⑧ Compare and report

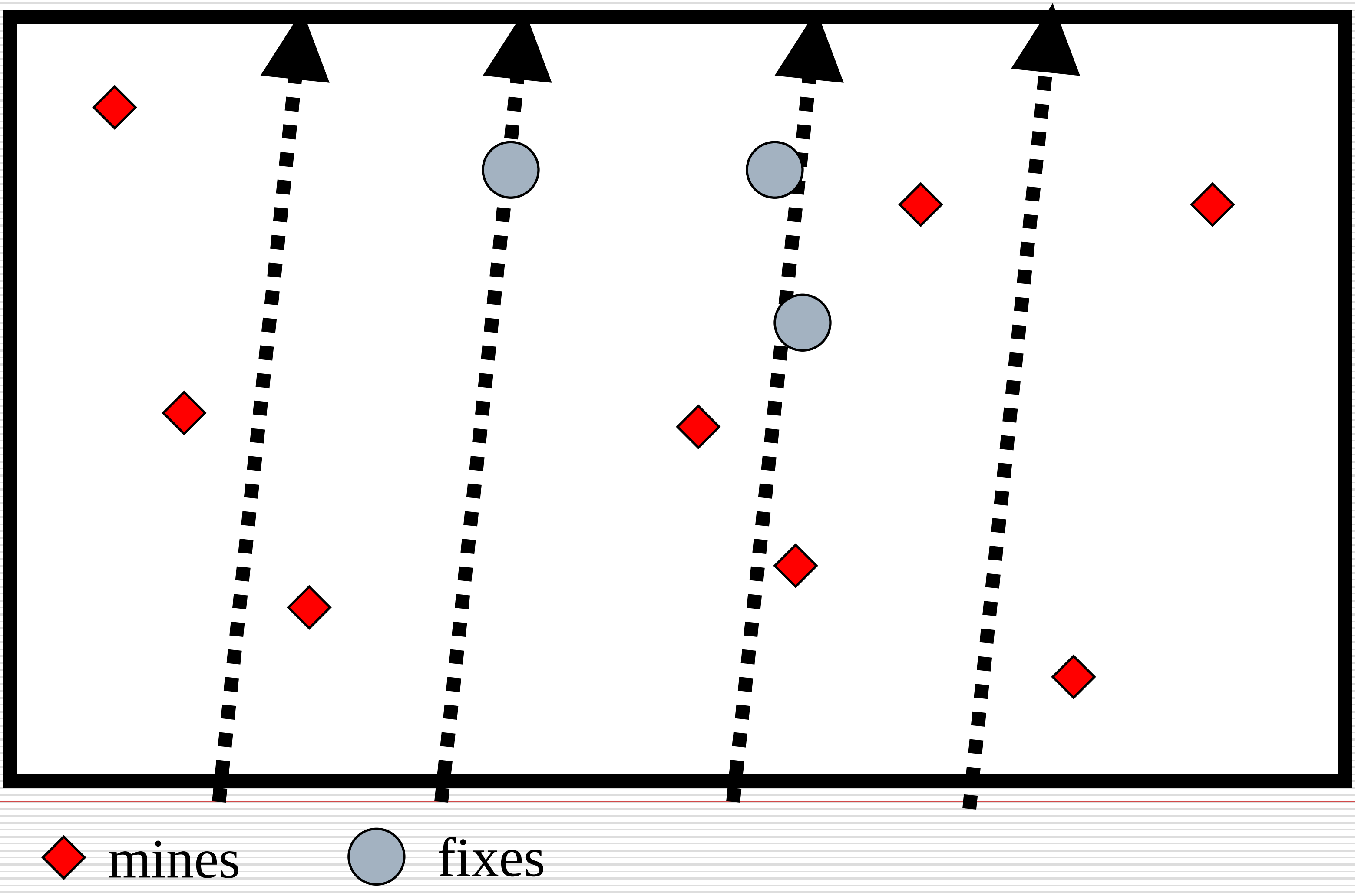
An analogy: Clearing mines



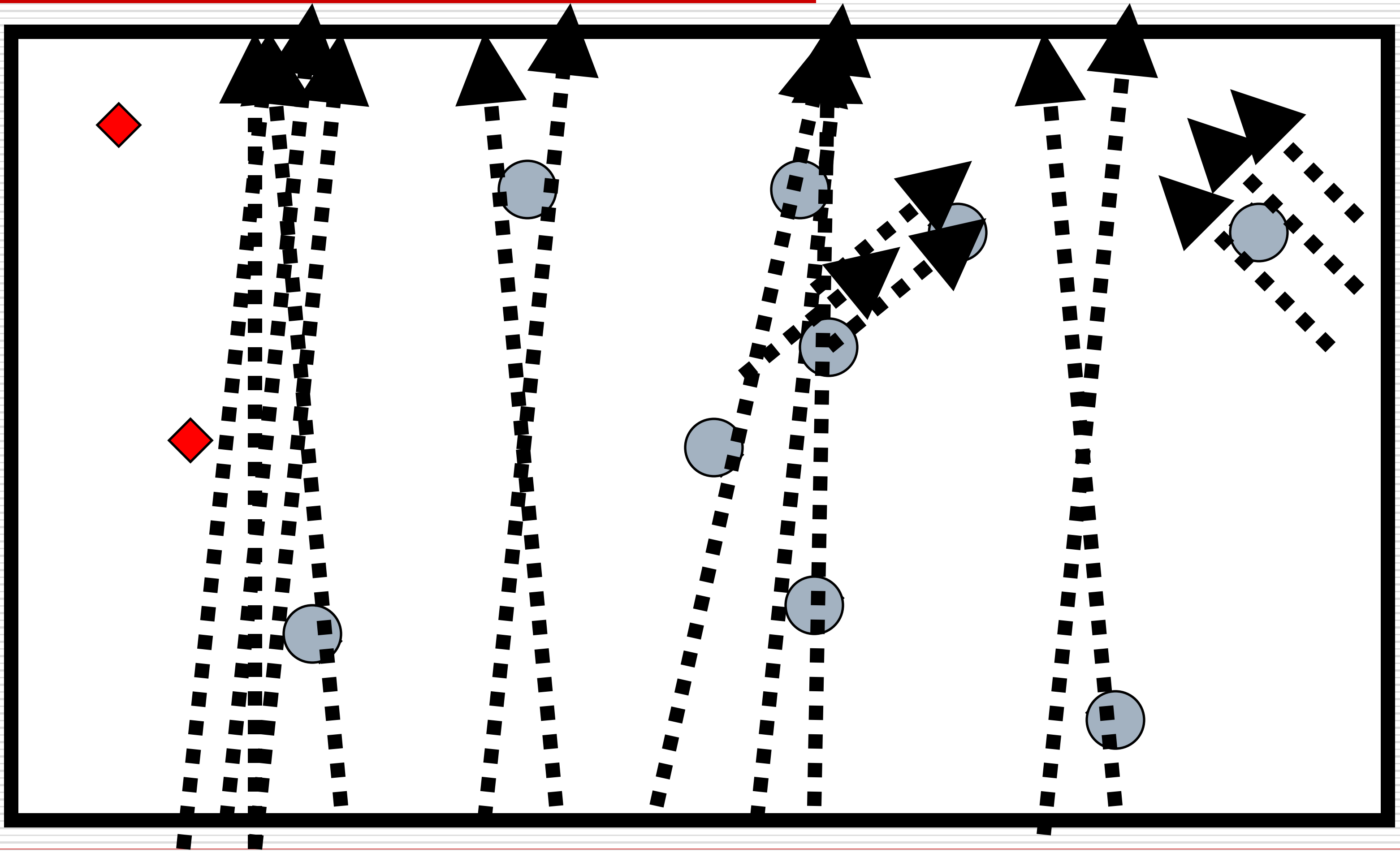
◆ mines

This analogy was first presented by Brian Marick.
These slides are from James Bach..

Totally repeatable tests won’t clear the minefield



Variable Tests are Often More Effective



Automated GUI regression testing

- Look back at the minefield analogy
- Are you convinced that variable tests will find more bugs under all circumstances?
  - ***If so, why would people do repeated tests?***

Is this really automation?

Analyze product	--	human
Design test	--	human
Run test 1st time	--	human
Evaluate results	--	human
Report 1st bug	--	human
Save code	--	human
Save result	--	human
Document test	--	human
Re-run the test	**--**	**MACHINE**
Evaluate result	**--**	**MACHINE plus human if there's any mismatch**
Maintain result	--	human

GUI automation is expensive

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. \

GUI automation pays off late

- GUI changes force maintenance of tests
  - May need to wait for GUI stabilization
  - Most early test failures are due to GUI changes
- Regression testing has low power
  - Rerunning old tests that the program has passed is less powerful than running new tests
  - Old tests do not address new features
- Maintainability is a core issue because our main payback is usually in the next release, not this one

Test automation is programming

- Win NT 4 had 6 million lines of code, and 12 million lines of test code
- Common (and often vendor-recommended) design and programming practices for automated testing are appalling (kinh khủng):
  - **Embedded constants**
  - No modularity
  - ***No source control***
  - *No documentation*
  - `No requirements analysis`
  - **No wonder we fail.**

Common mistakes in GUI automation

- Don’t underestimate the cost of automation.
- Don’t underestimate the need for staff training.
- Don’t expect to be more productive over the short term.
- Don’t spend so much time and effort on regression testing.
- Don’t use instability of the code as an excuse.
- Don’t put off finding bugs in order to write test cases.
- Don’t write simplistic test cases.
- Don’t shoot for “100% automation.”
- Don’t use capture/replay to create tests.
- Don’t write isolated scripts in your spare time

Common mistakes in GUI automation (cont.)

- Don’t create test scripts that won’t be easy to maintain over the long term.
- Don’t make the code machine-specific.
- Don’t fail to treat this as a genuine programming project.
- Don’t “forget” to document your work.
- Don’t deal unthinkingly with ancestral code.
- Don’t give the high-skill work to outsiders.
- Don’t insist that all of your testers be programmers.
- Don’t put up with bugs and crappy support for the test tool.
- Don’t forget to clear up the fantasies that have been spoonfed to your management.

Think about:

- Automation is software development.
- Regression automation is expensive and can be inefficient.
- Automation need not be regression--you can run new tests instead of old ones.
- Maintainability is essential.
- Design to your requirements.
- Set management expectations with care.

Some Simple Automation Approaches

Getting Started With Automation of Software Testing

Six Sometimes-Successful “Simple” Automation Architectures

- Quick & dirty
- Equivalence testing
- Frameworks
- Real-time simulator with event logs
- Simple Data-driven
- Application-independent data-driven

Quick & Dirty

- Smoke tests
- Configuration tests
- Variations on a theme
- Stress, load, or life testing

Equivalence Testing

- A/B comparison
- Random tests using an oracle (Function Equivalence Testing)
- Regression testing is the weakest form

Framework-Based Architecture

- Frameworks are code libraries that separate routine calls from designed tests.
  - modularity
  - reuse of components
  - hide design evolution of UI or tool commands
  - partial salvation from the custom control problem
  - independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
  - important utilities, such as error recovery
- For more on frameworks, see Linda Hayes’ book on automated testing, Tom Arnold’s book on Visual Test, and Mark Fewster & Dorothy Graham’s excellent new book “Software Test Automation.”

Real-time Simulator

- Test embodies rules for activities
- Stochastic process
- Possible monitors
  - Code assertions
  - Event logs
  - State transition maps
  - Oracles

Data-Driven Architecture

- In test automation, there are (at least) three interesting programs:
  - The software under test (SUT)
  - The automation tool that executes the automated test code
  - The test code (test scripts) that define the individual tests
- From the point of view of the automation software, we can assume
  - The SUT's variables are data
  - The SUT's commands are data
  - The SUT's UI is data
  - The SUT's state is data
  - The test language syntax is data
- Therefore it is entirely fair game to treat these implementation details of the SUT as values assigned to variables of the automation software.
- Additionally, we can think of the externally determined (e.g. determined by you) test inputs and expected test results as data.
- Additionally, if the automation tool's syntax is subject to change, we might rationally treat the command set as variable data as well.

Data-Driven Architecture

- In general, we can benefit from separating the treatment of one type of data from another with an eye to:
  - optimizing the maintainability of each
  - optimizing the understandability (to the test case creator or maintainer) of the link between the data and whatever inspired those choices of values of the data
  - minimizing churn that comes from changes in the UI, the underlying features, the test tool, or the overlying requirements
- You store and display the different data can be in whatever way is most convenient for you

The dos and don'ts of testing automation

- Keeping up with upkeep
- Choose which tests to automate
- Don't automate from day one
- Get all the right people involved
- Connect manual tests with automation using a clear methodology
- Carefully choose which tests should be run at each execution

The Dos of Test Automation

- Break Your Tests Into Shorter And Independent Scenarios
- Choose The Tests You Need Automating
- Start Small And Gradually Create Test Suites As You Proceed
- Set Priorities
- Employ Professionals And Use The Appropriate Tools
- Let people know of the results of the automated runs - and think about who the receiver is
- Make your automated setup work hard
- Make your suite quick and easy to run

The Don’ts of Test Automation

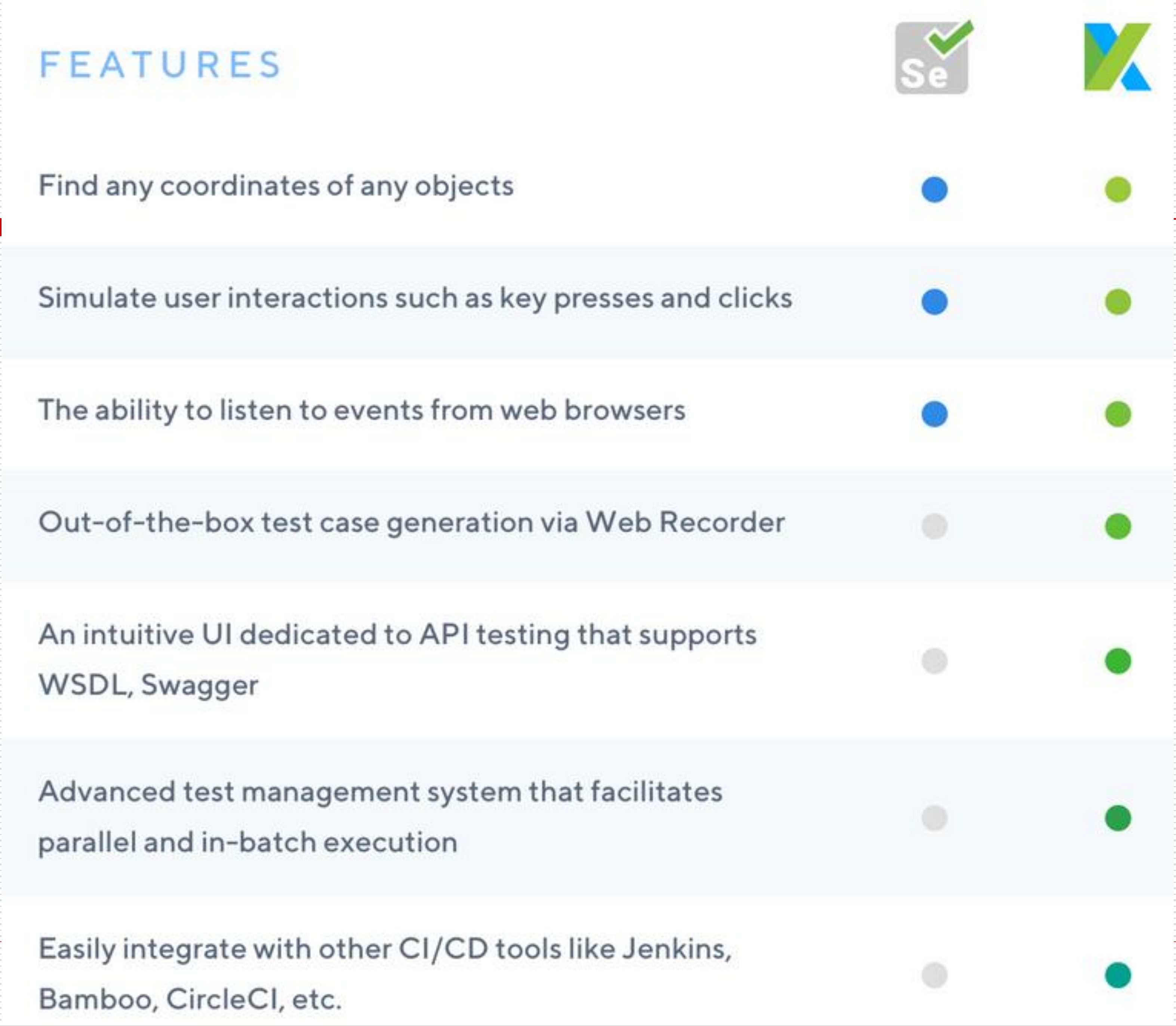
- Do Not Start Automating From The First Day Itself
- Don’t Try Running Everything At Every Time
- Don’t Automate All The Manual Tests
- Don’t Automate Everything
- Don’t Ignore Further Development
- Buy the first, the best automation tool
- Run everything, every time
- Forget to test your tests
- Underestimate the effort it takes to keep an automated suite running

Difference Between Retesting and Regression Testing

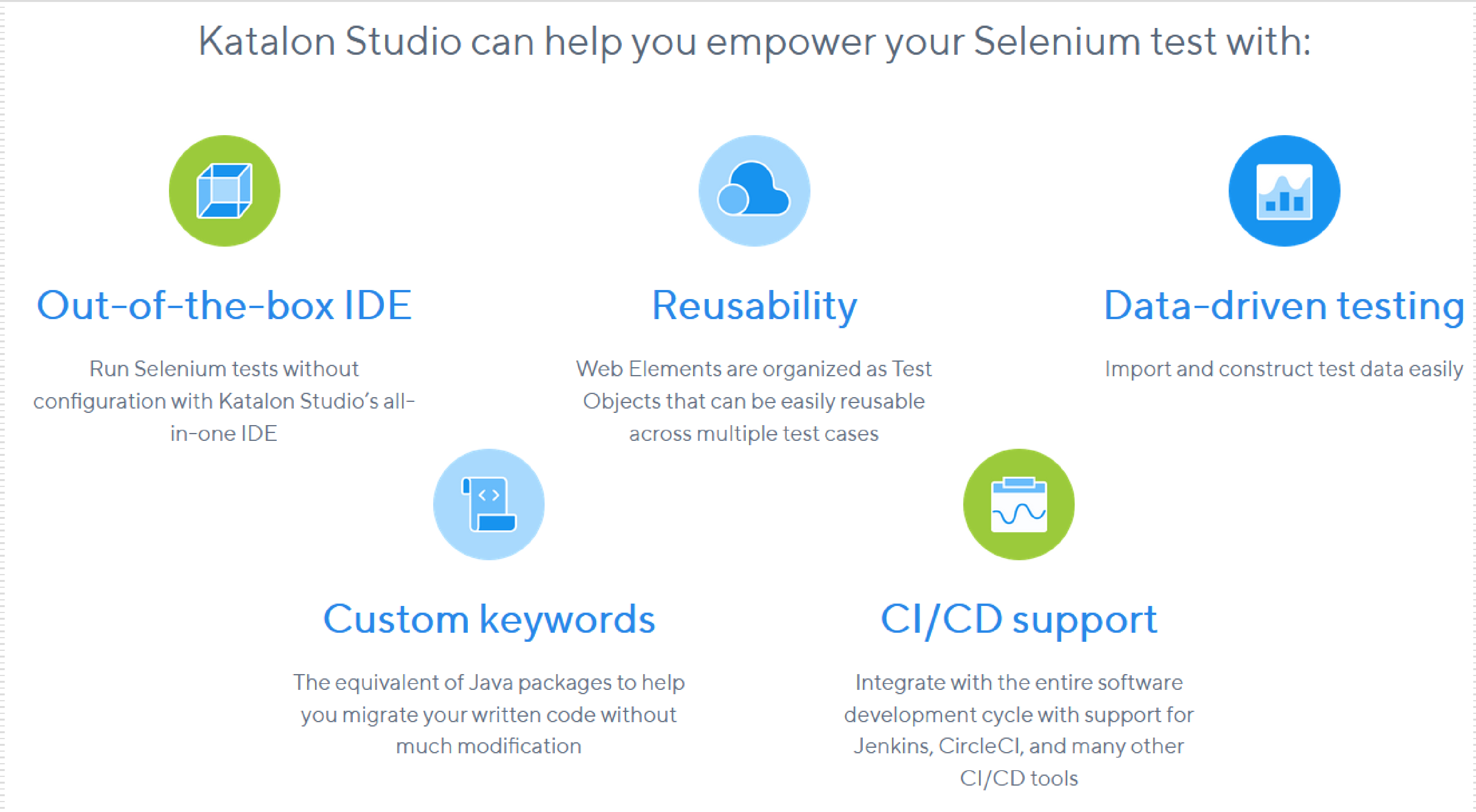
- Retesting is a process to check specific test cases that are found with bug/s in the final execution. Generally, testers find these bugs while testing the software application and assign it to the developers to fix it. Then the developers fix the bug/s and assign it back to the testers for verification. This continuous process is called Retesting.
- Regression Testing is a type of software testing executed to check whether a code change has not unfavourably disturbed current features & functions of an Application

KEY DIFFERENCE

- Regression testing is performed for passed test cases while Retesting is done only for failed test cases.
- Regression testing checks for unexpected side-effects while Re-testing makes sure that the original fault has been corrected.
- Regression Testing doesn’t include defect verification whereas Re-testing includes defect verification.
- Regression testing is known as generic testing whereas Re-testing is planned testing.
- Regression Testing is possible with the use of automation whereas Re-testing is not possible with automation.



FEATURES		
Find any coordinates of any objects	🔵	🟢
Simulate user interactions such as key presses and clicks	🔵	🟢
The ability to listen to events from web browsers	🔵	🟢
Out-of-the-box test case generation via Web Recorder	⚪	🟢
An intuitive UI dedicated to API testing that supports WSDL, Swagger	⚪	🟢
Advanced test management system that facilitates parallel and in-batch execution	⚪	🟢
Easily integrate with other CI/CD tools like Jenkins, Bamboo, CircleCI, etc.	⚪	🟢



Katalon Studio can help you empower your Selenium test with:

Out-of-the-box IDE

Run Selenium tests without configuration with Katalon Studio's all-in-one IDE

Reusability

Web Elements are organized as Test Objects that can be easily reusable across multiple test cases

Data-driven testing

Import and construct test data easily

Custom keywords

The equivalent of Java packages to help you migrate your written code without much modification

CI/CD support

Integrate with the entire software development cycle with support for Jenkins, CircleCI, and many other CI/CD tools

- https://eduinpro.com/blog/difference-between-selenium-vs-testcomplete/

Compare TestComplete and Selenium

Comparison Basis	TestComplete	Selenium
Software Type	It is a Windows-based tool. TestComplete is essentially Windows-based application and thus cannot run on Linux/Unix systems.	It is a set of APIs which can be used on at least three major operating systems:- Windows, Linux, and Mac.
Cost	It is a highly paid tool. You can bundle options or customize your order. You can view its pricehere.	Open Source/Free tool. You can download and use it for free. You can download it fromhere.
Developed By	SmartBear Software (A subsidiary of AutomatedQA)	Initially developed by ThoughtWorks later made as open source.
Application Type	It supports desktop, web, and mobile application	Selenium can be used only for Web-based applications. However, WebDriver based tool called Appium can be used for mobile

Comparison Basis	TestComplete	Selenium
Application Layer	TestComplete can be used to automate front end and API of the application.	Selenium can be used to test only the front end or interface layer of a Web Application.
Supported Languages	It supports seven different languages including JavaScript, Python, VBScript, Jscript, DelphiScript, C#, and C+	Java, C#, Ruby, Python, Perl Javascript, R, etc.
Supported Browsers	Chrome, Firefox, IE, Edge	IE, Firefox, Chrome, Safari, Opera, Headless browsers
Supported Operating Systems	Only Microsoft Windows	It supports Microsoft Windows, Apple OS X, Linux