

CS608

Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

Tutorial: Lab 8

- Web-based application testing for Adventure Travel
- User Stories and Completion Criteria:
 - Task 1: trial run
 - Task 2: analysis/interface elements
 - Task 3: analysis/data representation
 - Task 4: TCIs
 - Task 5: Data values
 - Task 6: Test Cases
 - Task 7: Implement using TestNG & Selenium

CS608

Application Testing in More Detail

(Essentials of Software Testing, Chapter 10, Sections 10.4-10.8)

Application Testing in More Detail

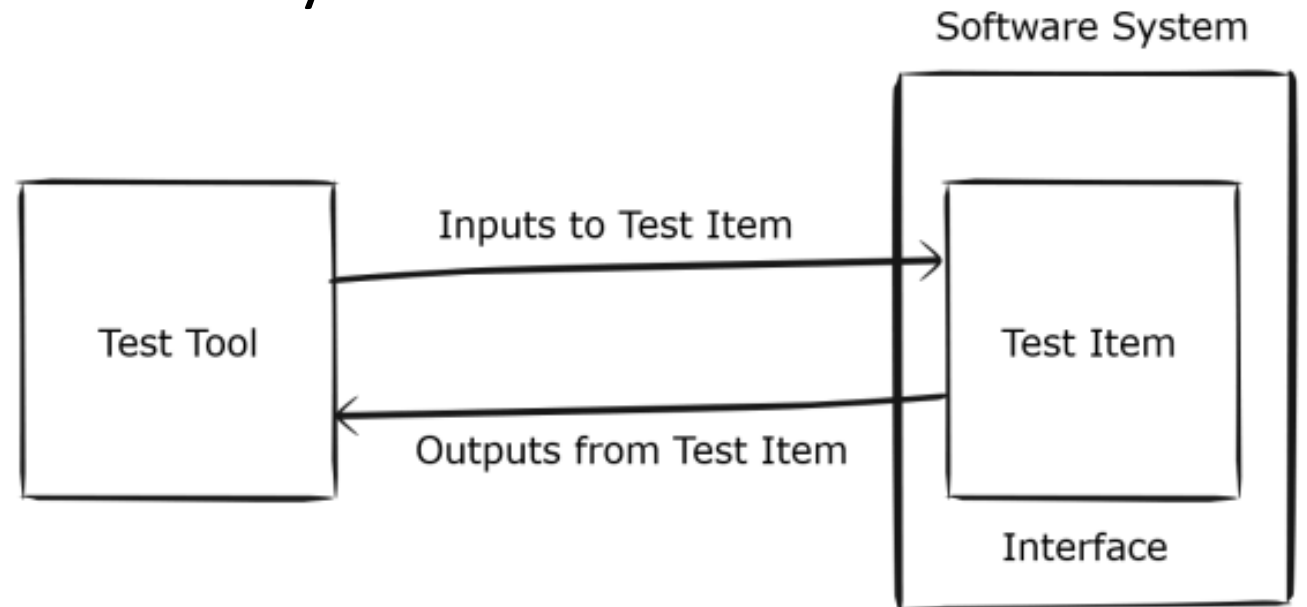
- Applications are a type of computer-based system with a user interface
- We will now consider system testing in general to better understand application testing

System Testing

- System testing means verifying that a system as a whole works correctly
 - Usually with black-box tests
- Simple white-box coverage (statement and branch) may be measured
 - May be used to develop further tests
- Many types of system
 - Each has its own unique testing challenges
 - All systems are tested over their system interface, and the principles of black-box and white-box testing apply when generating test data

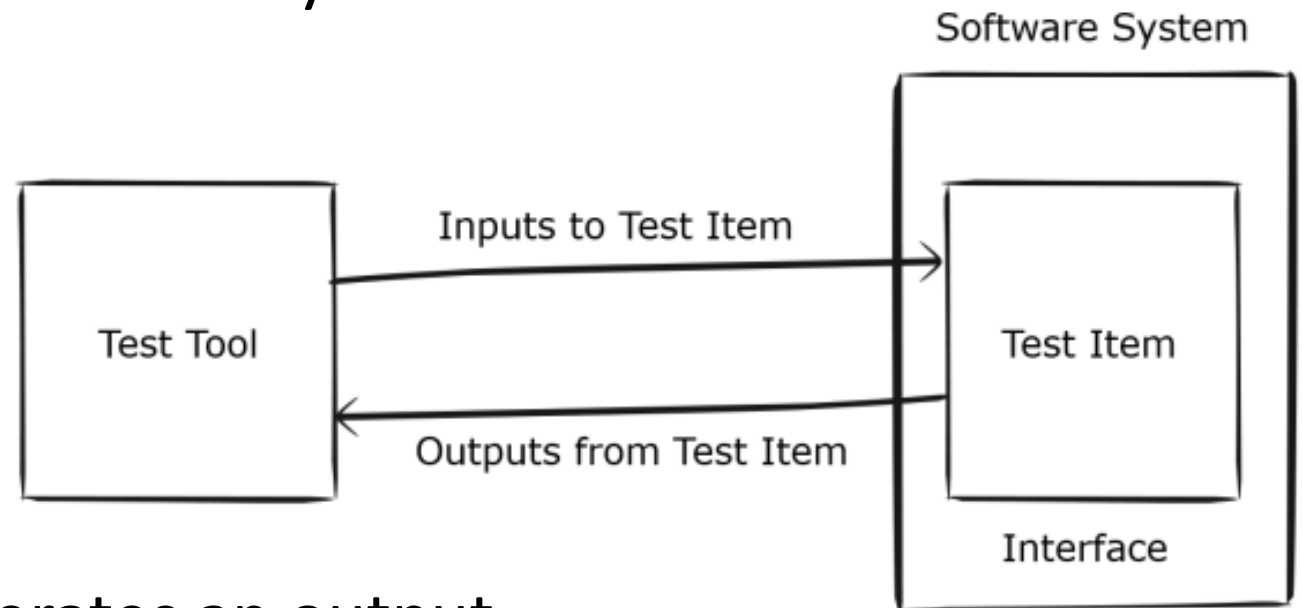
System Test Model

- Software systems are tested over their system interface



System Test Model

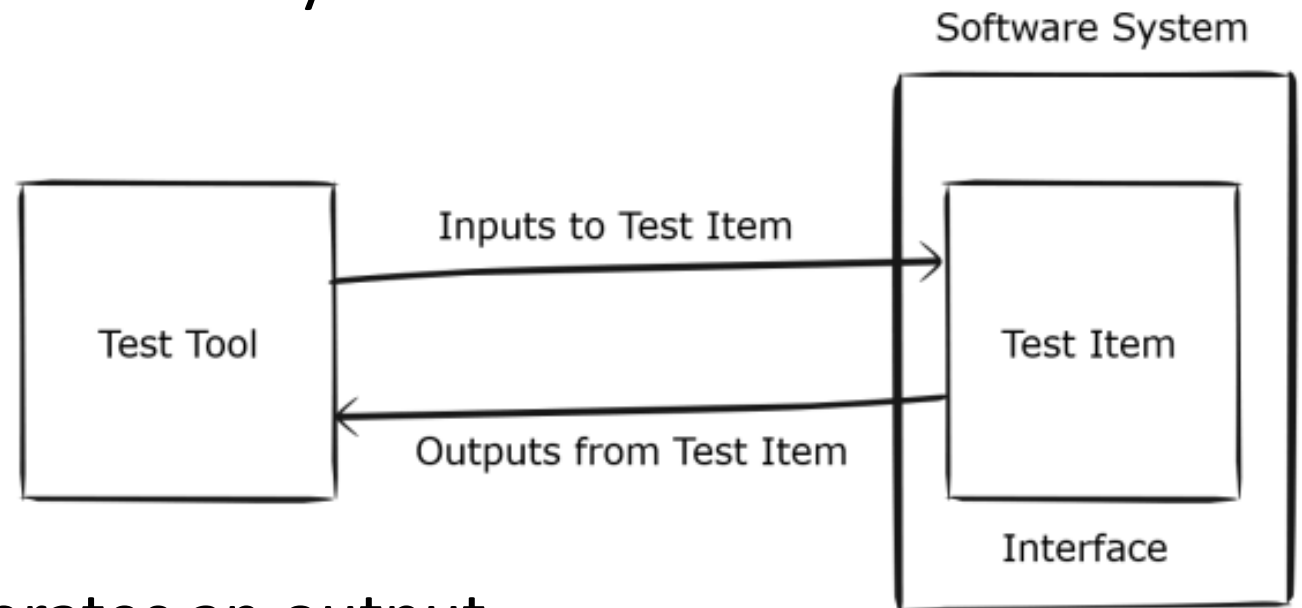
- Software systems are tested over their system interface



- **Synchronous:** every input generates an output

System Test Model

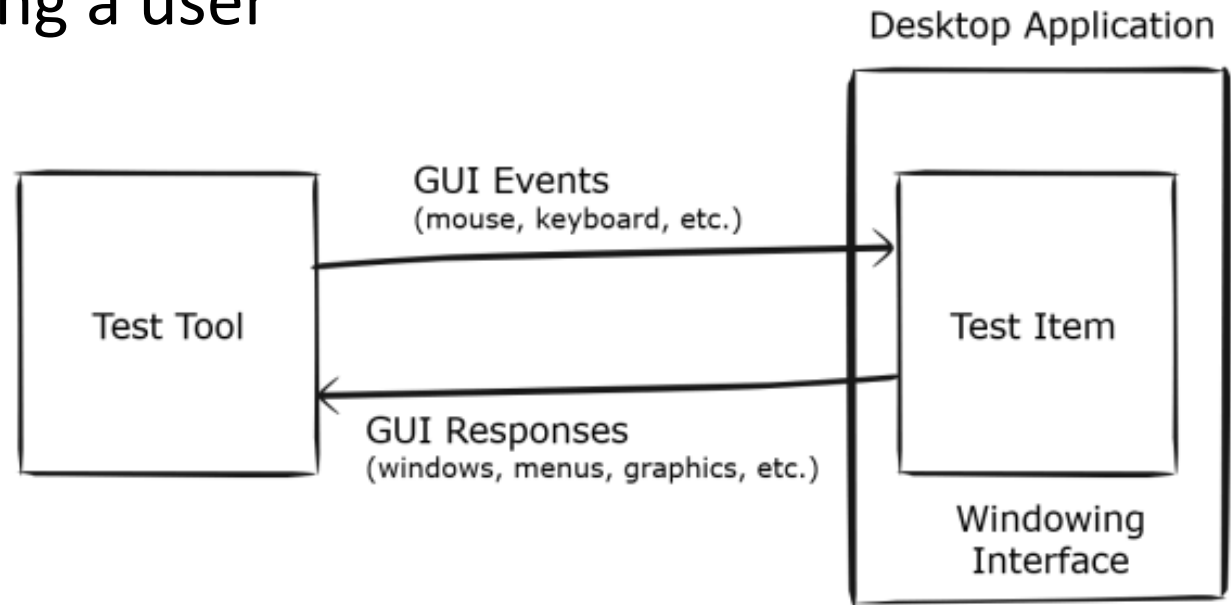
- Software systems are tested over their system interface



- Synchronous: every input generates an output
- **Asynchronous:**
 - Inputs may create sequences of outputs
 - Test Item may spontaneously generate outputs (timers, events)

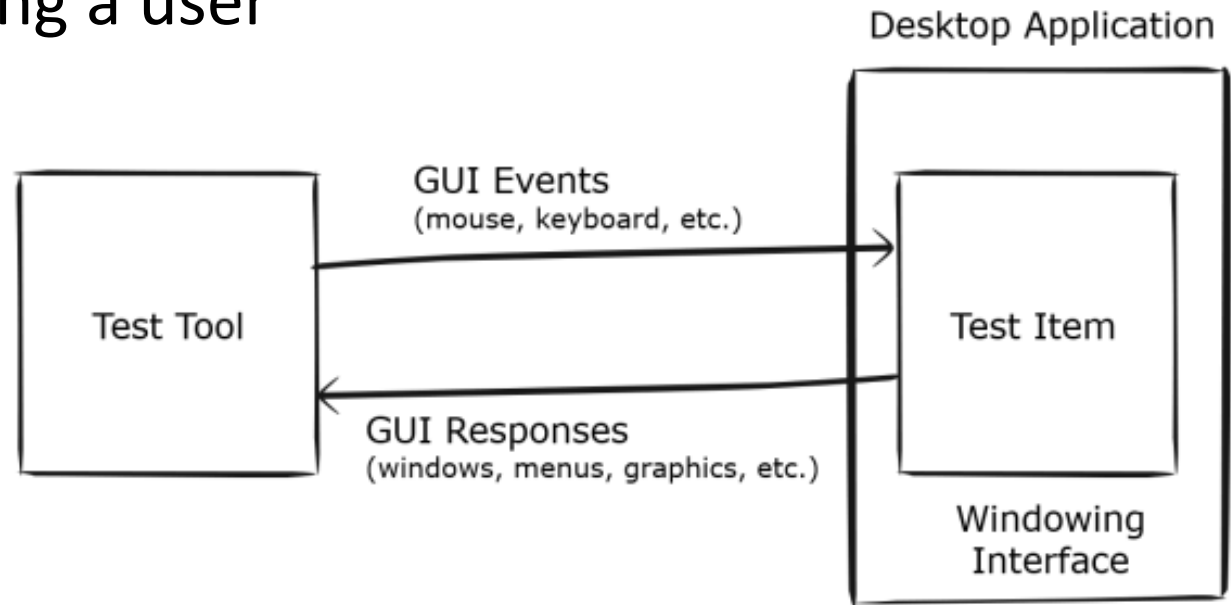
(Desktop) Application Test Model

- Test Tool interacts with Desktop Application over Windowing Interface (Java: AWT, Swing, JavaFX) by emulating a user



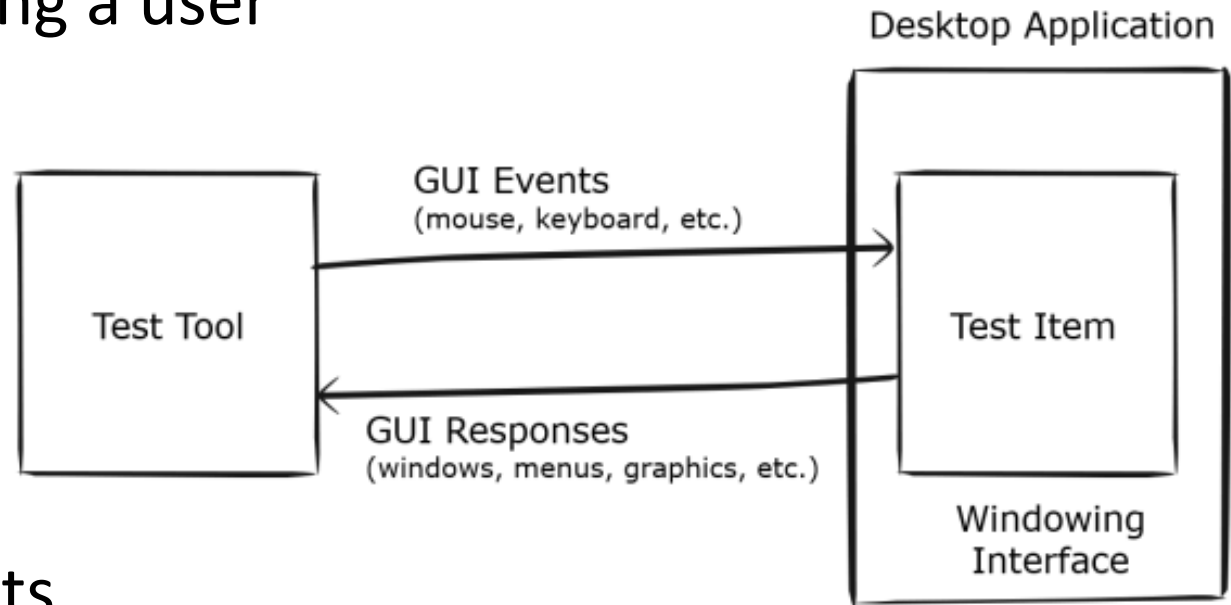
(Desktop) Application Test Model

- Test Tool interacts with Desktop Application over Windowing Interface (Java: AWT, Swing, JavaFX) by emulating a user
- When user interacts with the screen, the Windowing Interface software generates **GUI Events**



(Desktop) Application Test Model

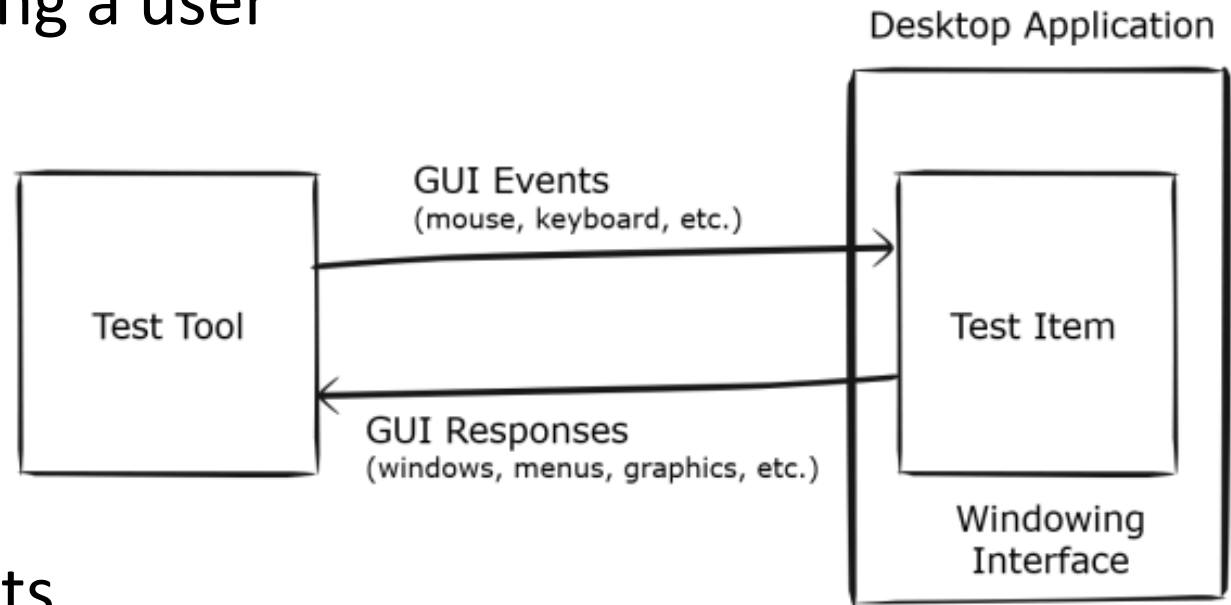
- Test Tool interacts with Desktop Application over Windowing Interface (Java: AWT, Swing, JavaFX) by emulating a user
- When user interacts with the screen, the Windowing Interface software generates **GUI Events**
- Test Tool generate the same GUI Events



(Desktop) Application Test Model

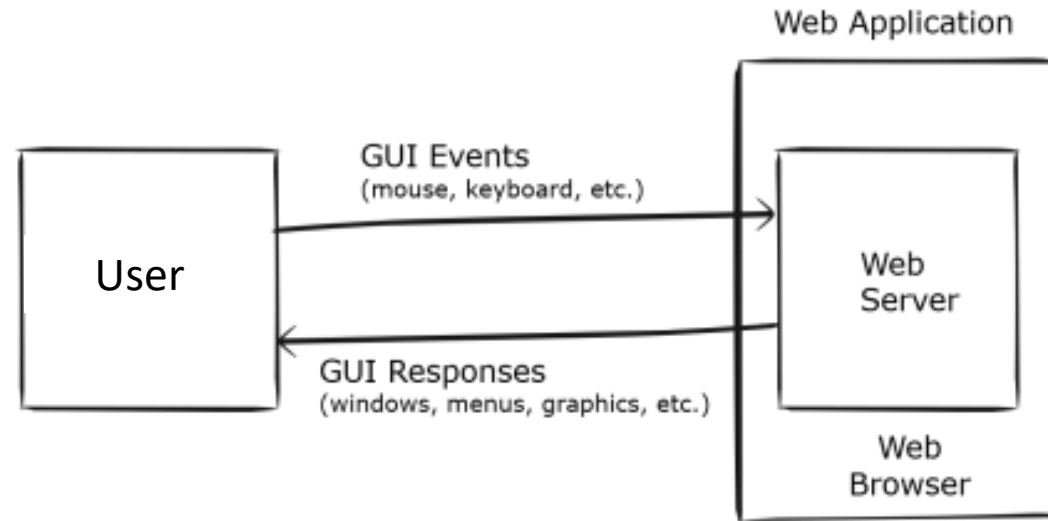
- Test Tool interacts with Desktop Application over Windowing Interface (Java: AWT, Swing, JavaFX) by emulating a user

- When user interacts with the screen, the Windowing Interface software generates **GUI Events**



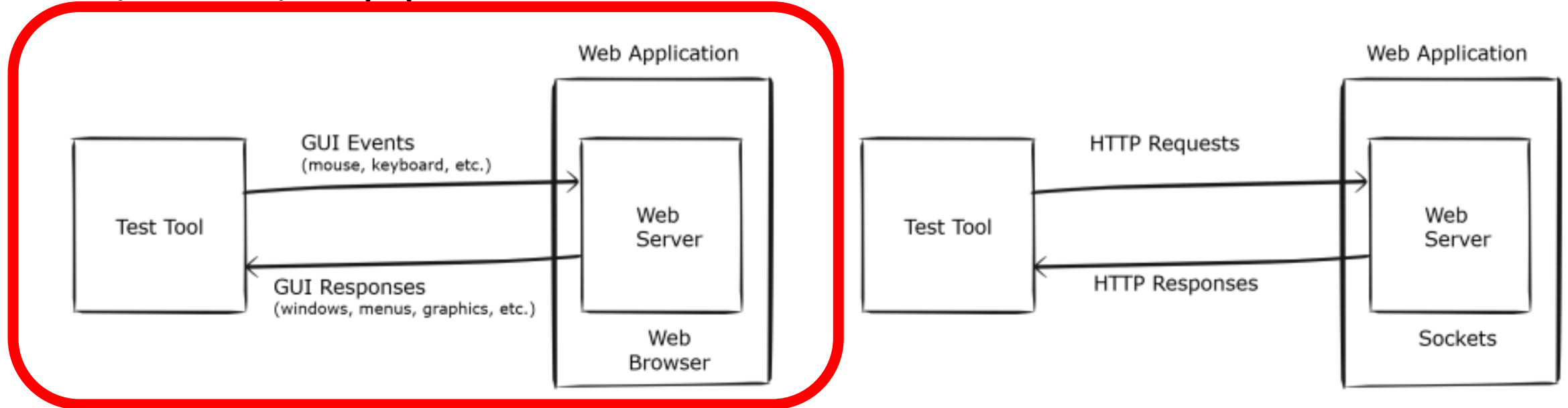
- Test Tool generate the same GUI Events
- Responses delivered back to Test Tool via Windowing Interface as GUI Responses

(Web) Application Test Models



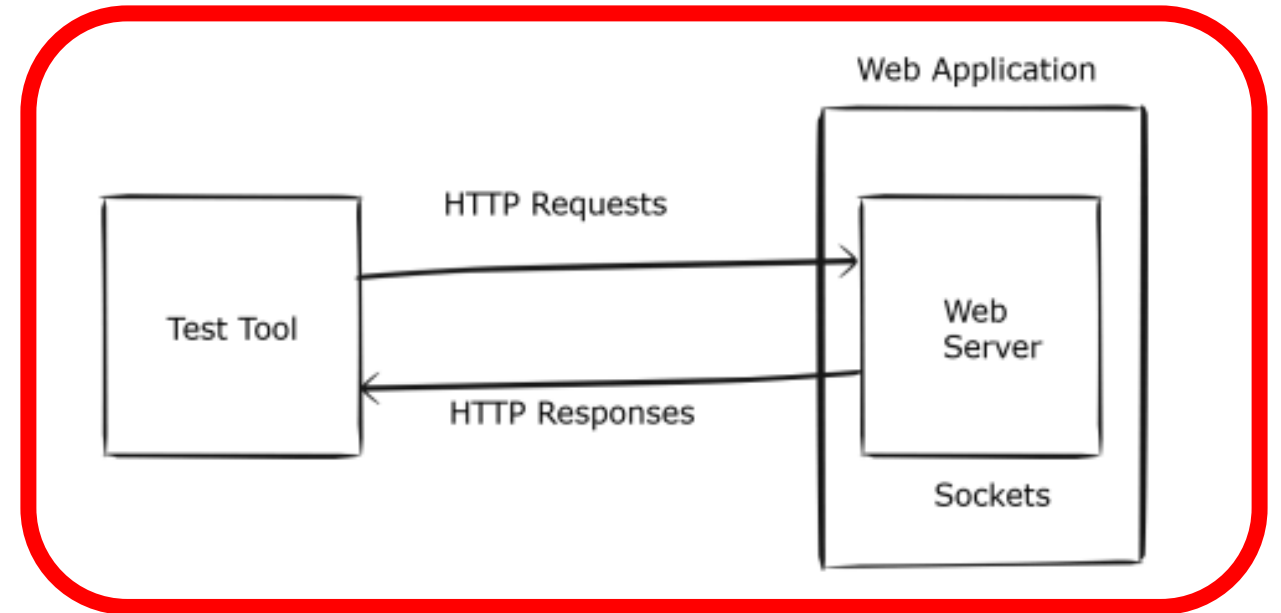
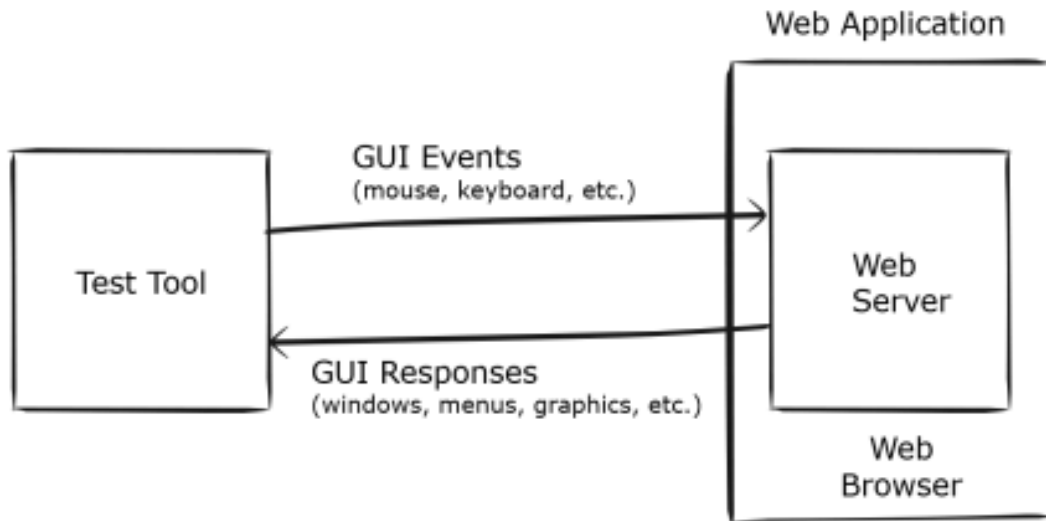
- Web applications use a Web Browser to provide the user interface
- Communication between the browser and application on Web Server is via the Hypertext Transfer Protocol (HTTP) over the network

(Web) Application Test Models



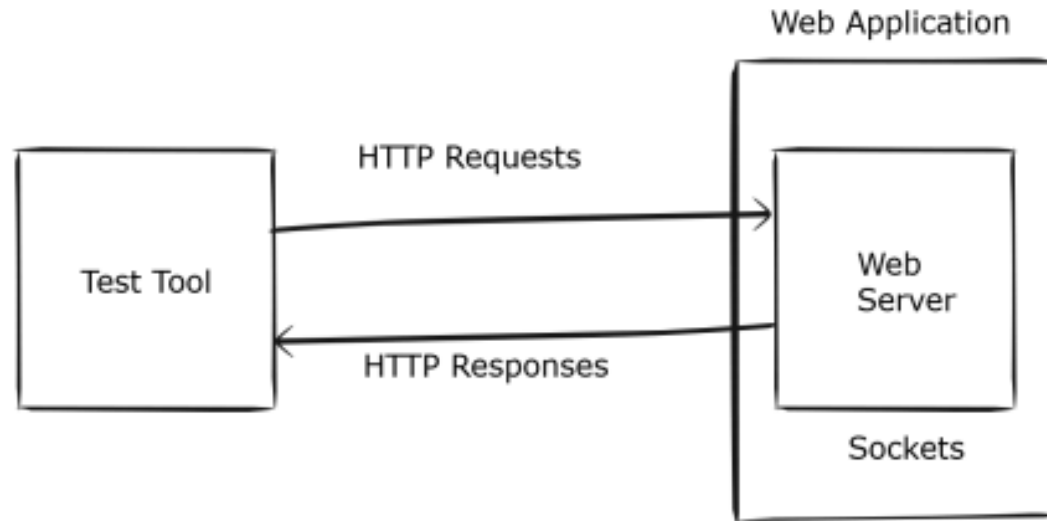
- Web applications use a Web Browser to provide the user interface
- Communication between the browser and application on Web Server is via the Hypertext Transfer Protocol (HTTP) over the network
- **Left: Test via browser – Test Tool emulates a user**
- Right: Test directly over network interface – using HTTP messages

(Web) Application Test Models



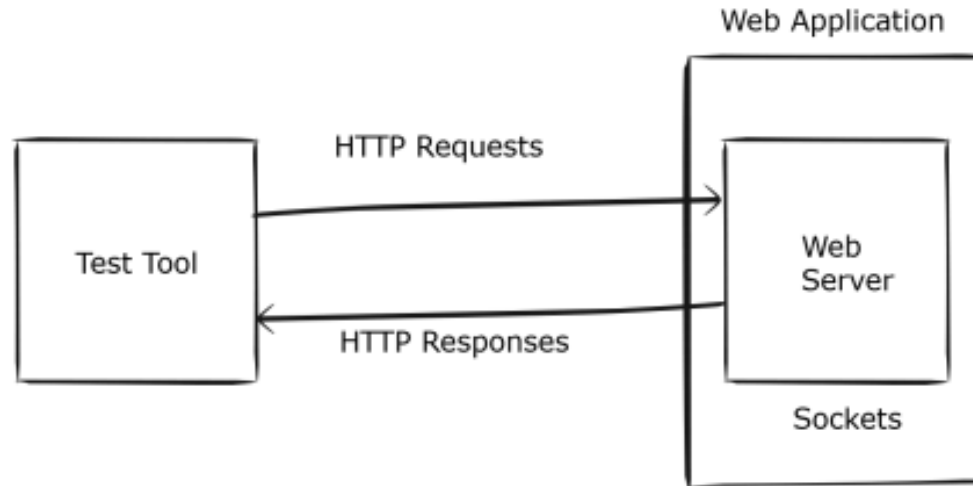
- Web applications use a Web Browser to provide the user interface
- Communication between the browser and application on Web Server is via the Hypertext Transfer Protocol (HTTP) over the network
- Left: Test via browser – Test Tool emulates a user
- **Right: Test directly over network interface – using HTTP messages**

(Web) Application Test Model/Direct Testing



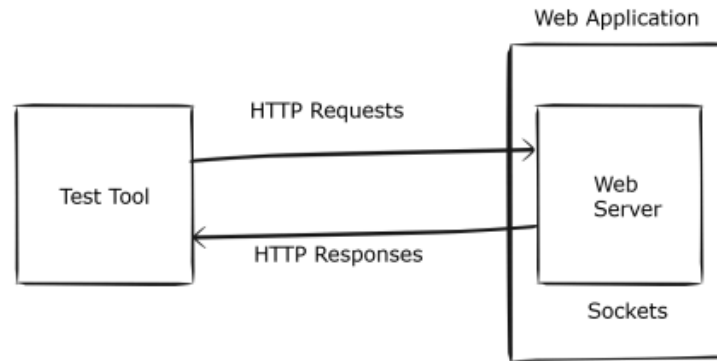
- Utilities/libraries to generate HTTP requests and parse HTTP responses for this purpose (e.g. curl, BeautifulSoup, etc.)

(Web) Application Test Model/Direct Testing



- Utilities/libraries to generate HTTP requests and parse HTTP responses for this purpose (e.g. curl, BeautifulSoup, etc.)
- Generally executes much faster – and testing is independent of the web browser

(Web) Application Test Model/Direct Testing



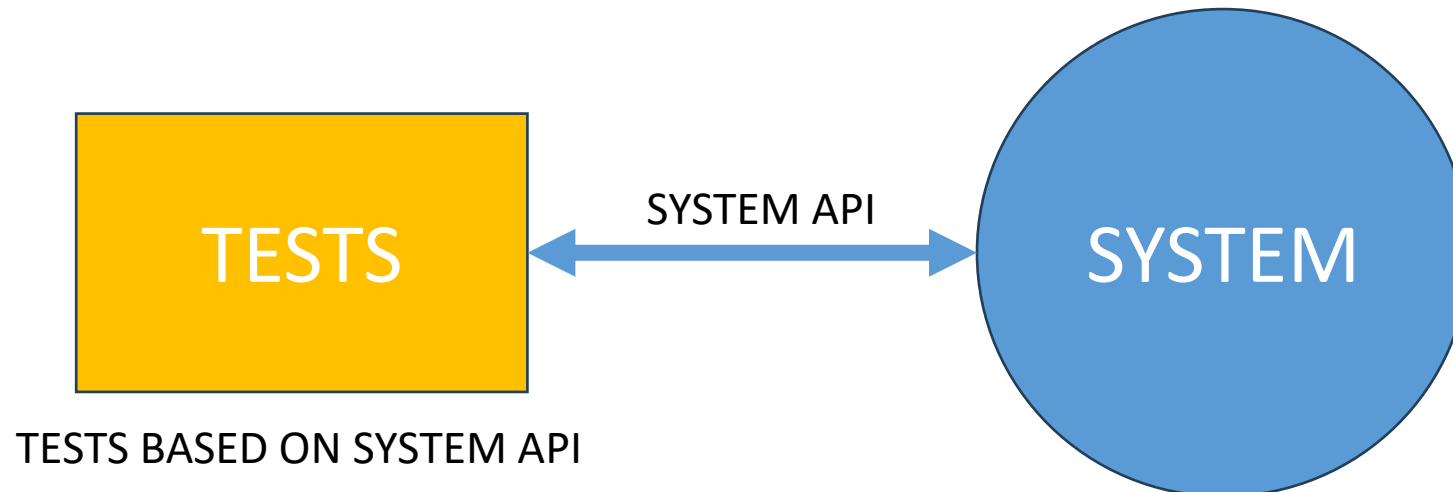
- Utilities/libraries to generate HTTP requests and parse HTTP responses for this purpose (e.g. curl, BeautifulSoup, etc.)
- Generally executes much faster – and testing is independent of the web browser
- However:
 - it is more complex to implement the tests:
 - requiring a deep understanding of HTML and HTTP
 - does not guarantee correct operation when running against a web browser
 - if the application uses JavaScript, then the test tool must support this (a substantial task)

System Testing and Integration Testing

- System testing and integration testing often both take place over the **same interface** – the system interface)
- But it is important to note that the **purpose** of each is different

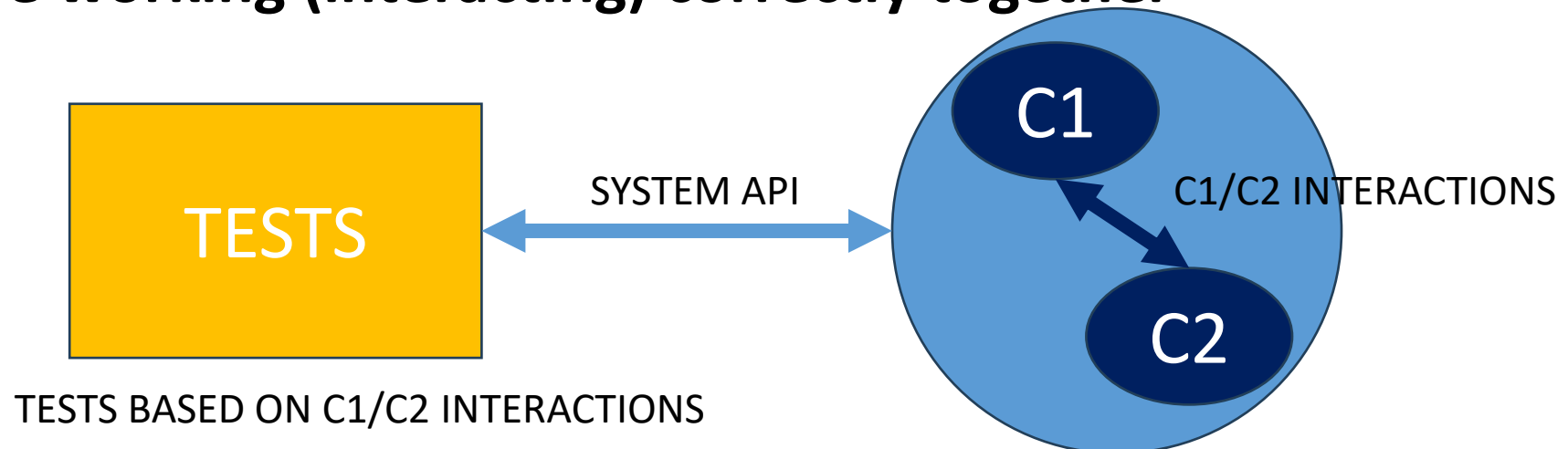
System Testing and Integration Testing

- System testing and integration testing often both take place over the same interface – the system interface)
- But it is important to note that the purpose of each is different
- System tests verify that the **system as a whole** is working correctly



System Testing and Integration Testing

- System testing and integration testing often both take place over the same interface – the system interface)
- But it is important to note that the purpose of each is different
- System tests verify that the system as a whole is working correctly
- Integration tests verify that **some components (or sub-systems) of a system are working (interacting) correctly together**

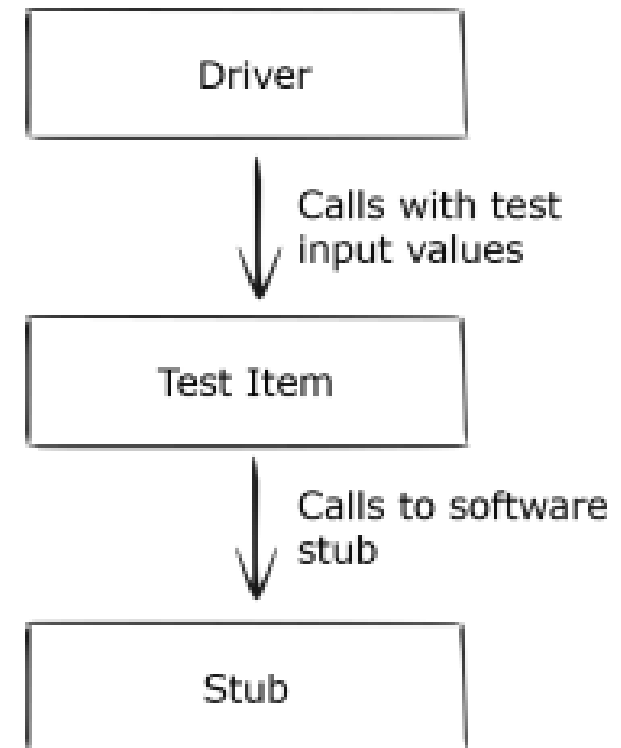


Integration Testing

- Takes a number of forms depending on the software process
- Traditional software process, based on producing layers of software, integration takes places between these layers:
 - Top Down Integration Testing
 - Bottom-Up Integration Testing
 - Feature Testing
- In a modern Agile process, adding new features changes many layers:
 - Integration testing makes sure that old and new user features integrate correctly together
 - Testing “End-to-End Functionality”
- Also, low level integration testing may also take place in a way similar to unit testing, where the purpose of the testing is to verify that two software components (usually represented by classes) work correctly together

Drivers, Stubs, and Mocks

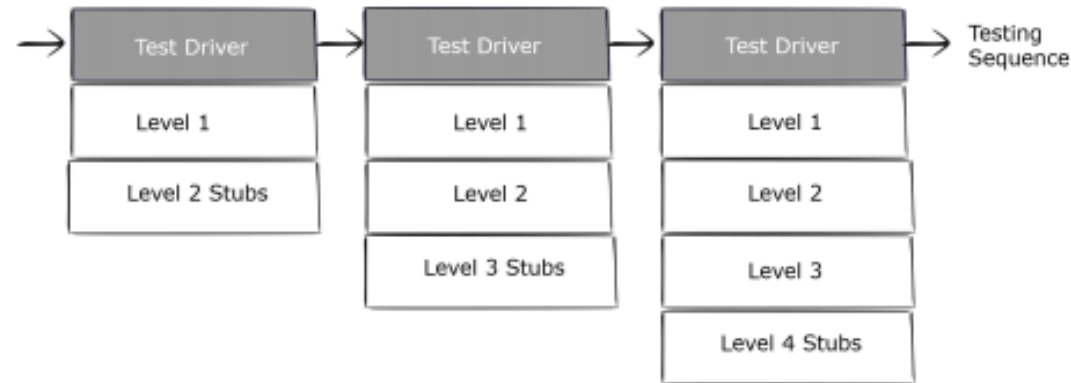
- When doing integration testing, the software is often incomplete
- This may require the tester to write temporary code to take the place of not-yet-written software
- Goal is to minimise the volume of this temporary software – but it may be necessary for testing
- Temporary software referred to as drivers, stubs, or mocks



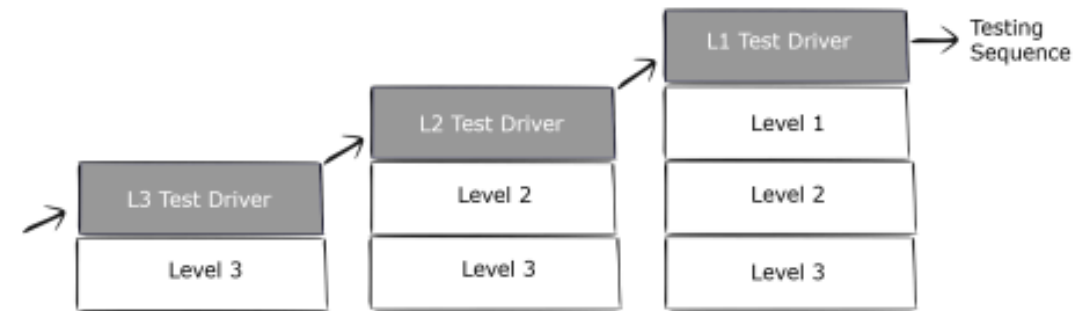
Stubs and Mocks

- The temporary stubs have limited functionality – often providing just enough support for the tests to execute
- Stubs may also be used to speed up tests by avoiding slow networking calls or to prevent actual actions taking place (such as sending emails or modifying an active database)
- Stubs may include instrumentation (e.g. counters added to the stub code) to measure how much of the temporary code has actually been called, or use assertions to verify correct operation (e.g. BIT)
- These instrumented stubs are often referred to as mocks
- Testing with mocks is referred to as mock testing or mocking, which is supported by many tools (e.g. EasyMock, JMock, Mockito, PowerMock)

Top-Down and Bottom-Up Integration Testing

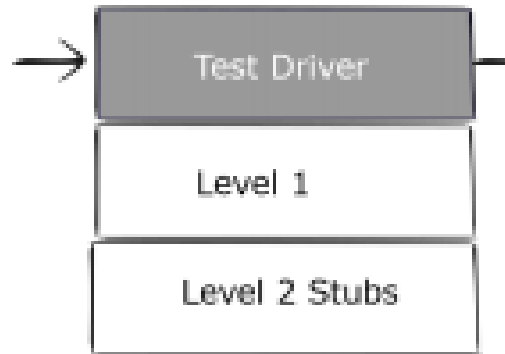


(a) Top-down



(b) Bottom-Up

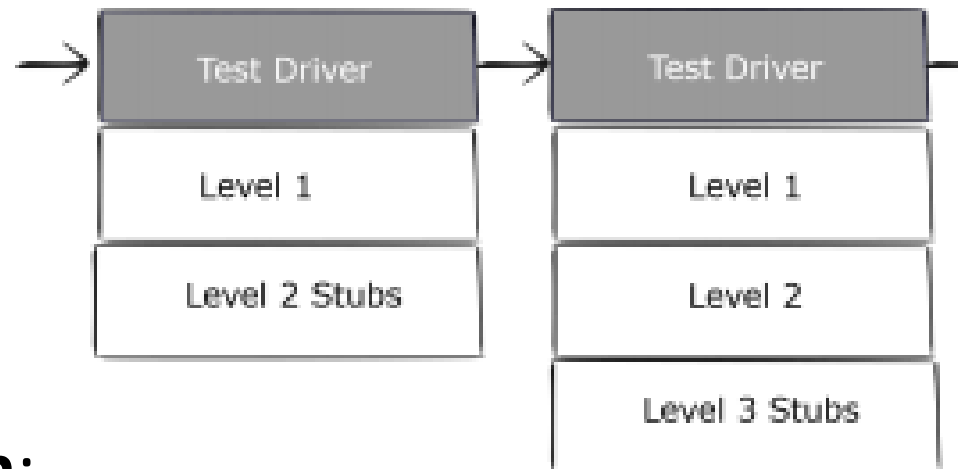
Top-Down Integration Testing



Top-down:

- Level 1 code has been written
- Write tests (test driver) for level 1
- Write stubs to stand in for lower levels of software (level 2) – just enough to pass the tests
- Level 1 tested first with Level 2 stubs

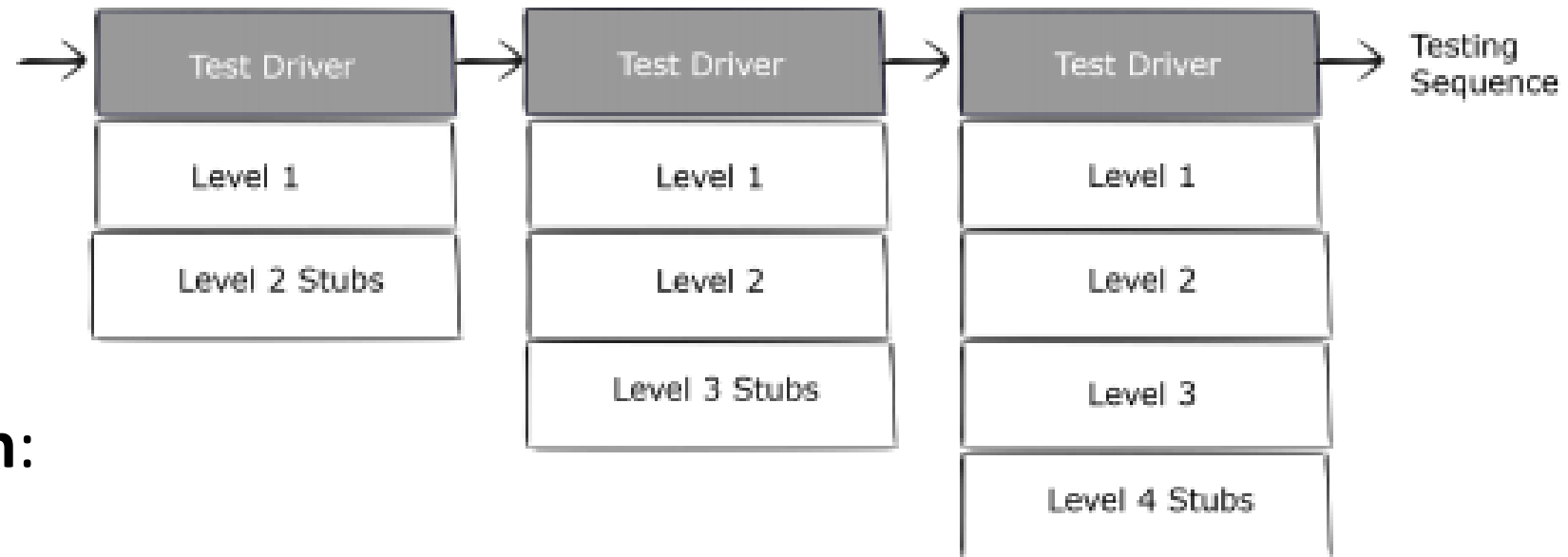
Top-Down Integration Testing



Top-down:

- Level 2 code is now written and integrated with the level 1 software
- Write stubs to stand in for lower levels of software (level 3)
- Level 1 tested next with Level 3 stubs (same tests)

Top-Down Integration Testing



Top-down:

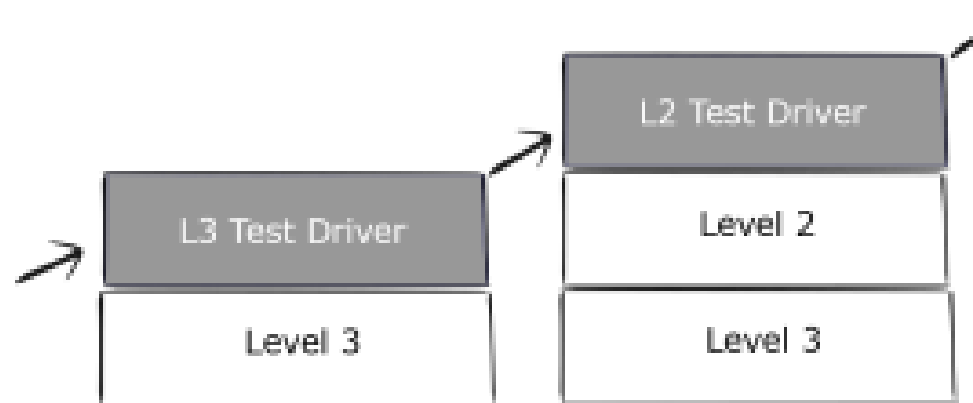
- Etc.
- One driver, multiple stubs

Bottom-Up Integration Testing



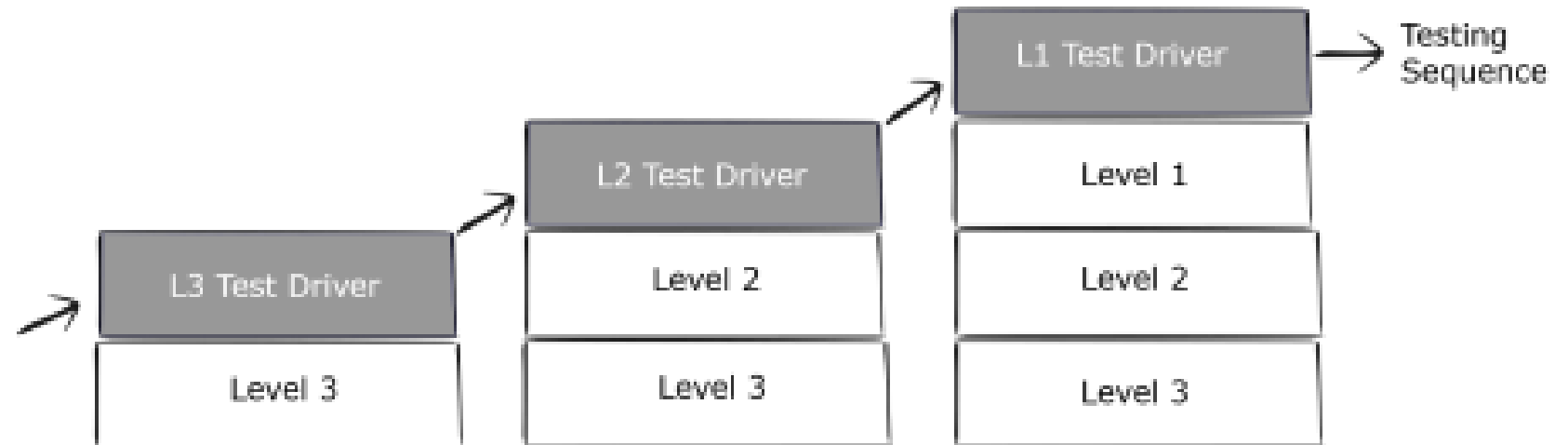
- **Bottom-up:**
- Level 3 code is written first
- The develop tests for level 3 (test driver)
- Level 3 tested first with no stubs

Bottom-Up Integration Testing



- **Bottom-up:**
- Then develop test driver for level 2
- Level 2 tested with actual Level 3 software

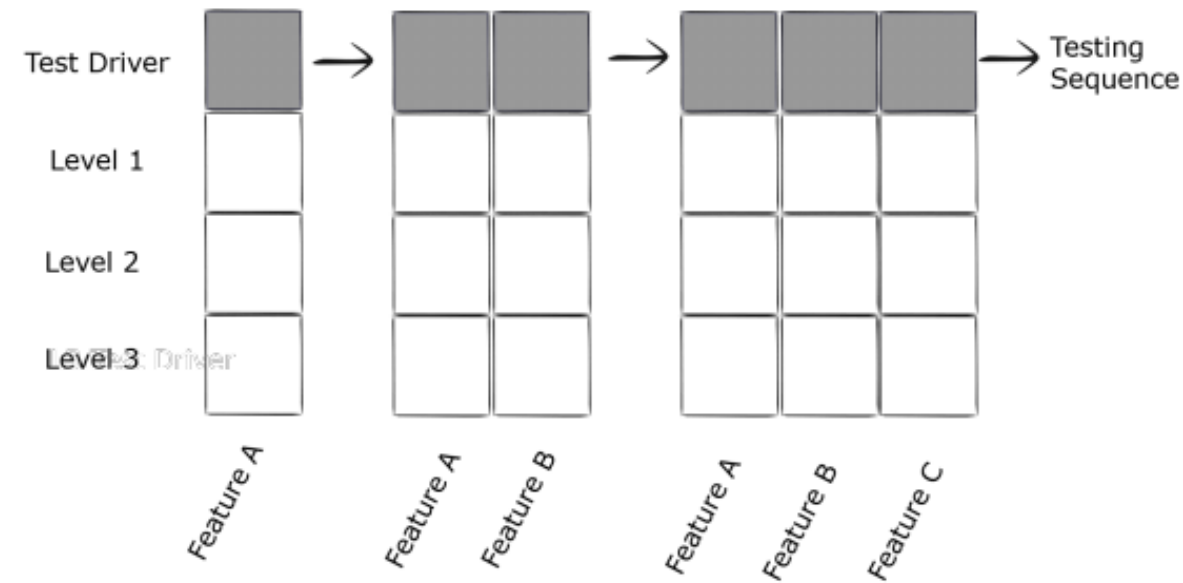
Bottom-Up Integration Testing



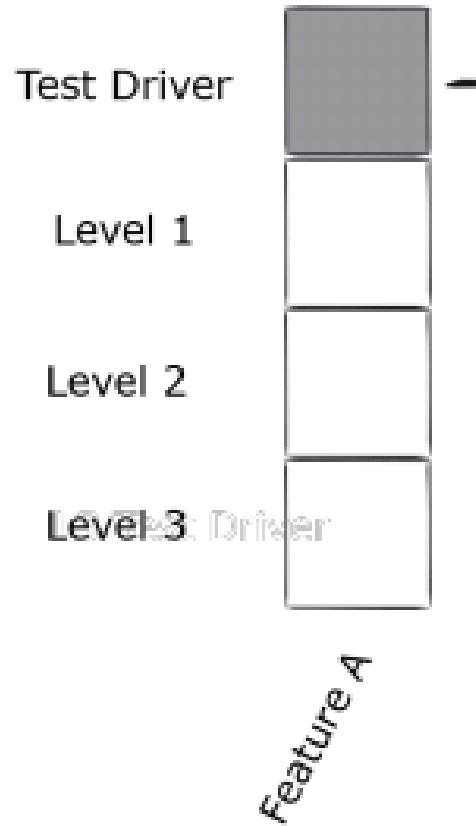
- **Bottom-up:**
- Etc.
- No stubs, multiple drivers (tests)

Feature Integration Testing

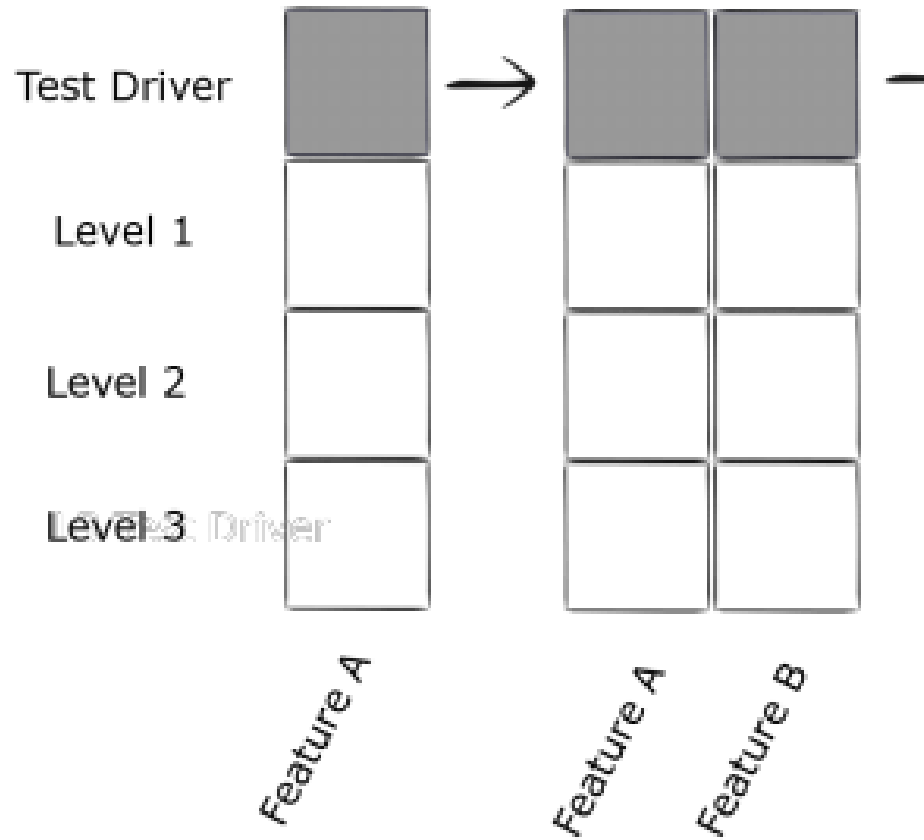
- Modern development processes develop increments of end-to-end user functionality
- E.g. “*SCRUM*” each deliverable feature in the *product backlog*
- Features tested in order they are added
- In practice, system tests (or application tests) are often used to act as integration tests, with mocks used to verify the integration
- Developing tests that thoroughly exercise each interface between the levels is time consuming (research topic)



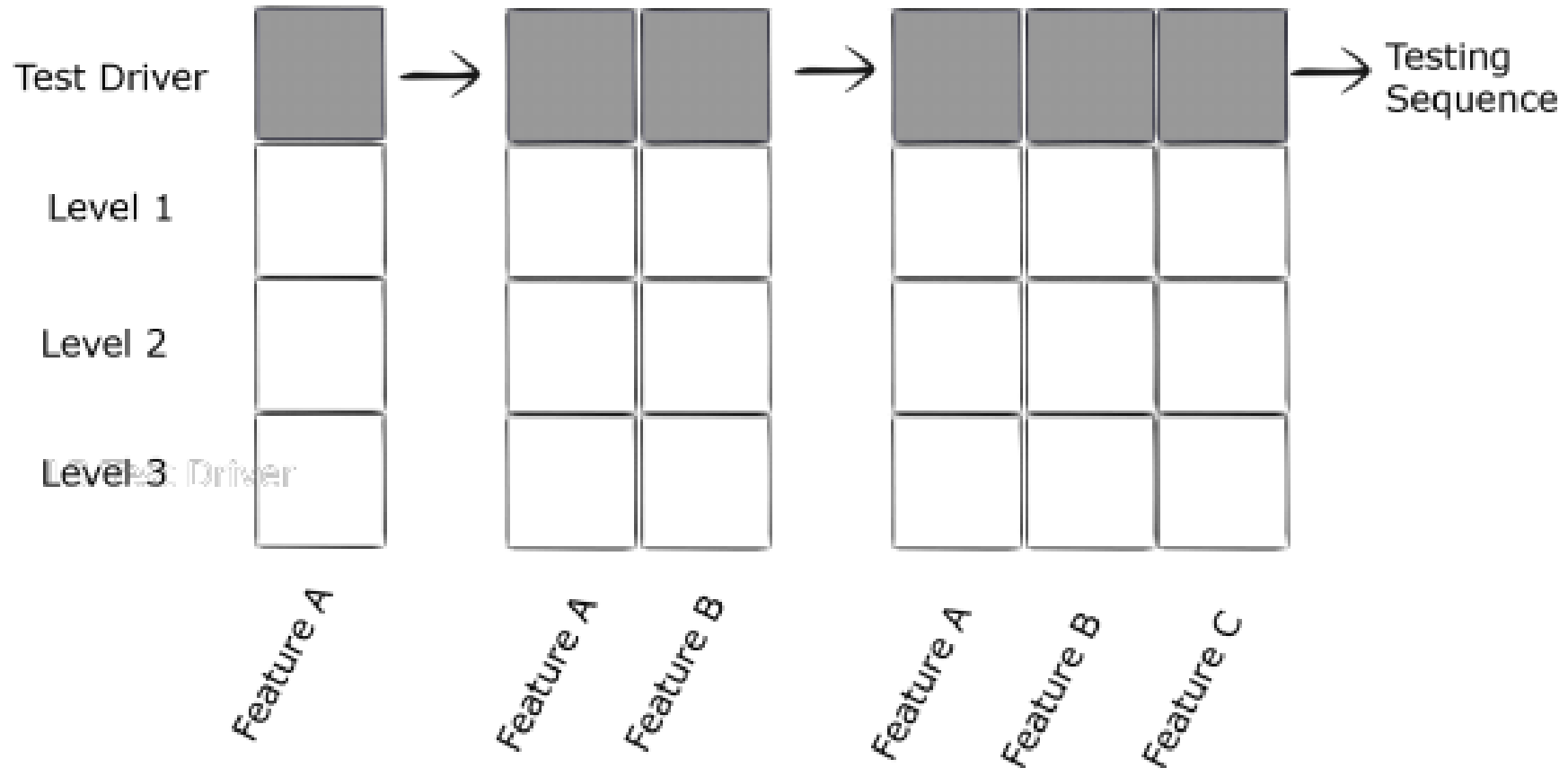
Feature Integration Testing (Feature A)



Feature Integration Testing (Features A,B)



Feature Integration Testing (Features A,B,C)



Methodology

- There are many **mechanisms & tools** for integration testing (some shown on previous slides)
- But the **methodology** for selecting tests (test coverage items, test cases, and test data) based on the internal component interactions, is poorly developed

Methodology

- There are many **mechanisms & tools** for integration testing (some shown on previous slides)
- But the **methodology** for selecting tests (test coverage items, test cases, and test data) based on the internal component interactions, is poorly developed
- The mechanisms and tools mainly allow for "testing during software integration" rather than "testing of the integration between software components"
- And most companies I am aware of use forms of system testing to serve the purpose

Methodology

- There are many **mechanisms & tools** for integration testing (some shown on previous slides)
- But the **methodology** for selecting tests (test coverage items, test cases, and test data) based on the internal component interactions, is poorly developed
- Very few concrete examples:
 - Swain and Mohapatra, *Test Case Generation from Behavioral UML Models*, International Journal of Computer Applications (Vol. 6/No. 8), Sept. 2010
 - Uses UML Sequence and Activity Diagrams to identify test cases based on internal interactions between components

Fault Models for Application Testing

- For *User Stories*, the principle fault model is related to the user tasks:
 - **User Requirements Faults** occur where the actual responses from the application do not match the expected responses as specified in the associated user story
- User stories seldom cover full functionality of the interface

Fault Models for Application Testing

- For *User Stories*, the principle fault model is related to the user tasks:
 - **User Requirements Faults** occur where the actual responses from the application do not match the expected responses as specified in the associated user story
- User stories seldom cover full functionality of the interface
- When using an application:
 - A user will navigate between different screens
 - Each screen contains multiple interface elements
 - These elements interact with underlying software features
- Maps to MVC (model-view-controller) design

Fault Models from MVC Model

- **[MODEL] Software Feature Faults** occur where a set of interactions with the system do not result in the expected result/output
 - A user story often involves several features (e.g. login, book a seat, pay, logout)
 - A feature can be regarded as a method or class, and the black-box techniques of equivalence partition, boundary value analysis, and decision tables applied to testing each

Fault Models from MVC Model

- **[VIEW] Screen Element Faults** occur where interactive interface component of the system does not exhibit the expected behaviour:
 - A button should perform some action when clicked
 - A text input box should accept typing
 - A hyperlink should navigate to another page
 - Expected behaviour often undocumented. For example, designer assumes user type name into textbox with prompt "Username" and not specify exactly how the textbox is should behave

Fault Models from MVC Model

- **[CONTROLLER] Navigation Faults** occur where the navigation between the different interface components (windows, screens, pages, forms, etc.) does not work correctly. The application may display the wrong screen, or may ignore the user action and do nothing.

Other Fault Models

- User Requirements Based:
 - User Stories
 - UML Use Cases
- Design/Architecture Based:
 - MVC
 - MVP
 - FLUX
 - MVVM
 - etc.

Fault Model Overlaps

- Some of these fault models may overlap
- For example, if an HTML anchor link is not implemented correctly:
 - It will not behave correctly when clicked (**Element Behaviour Fault**)
 - Leading to the next screen not being displayed (**Navigation Fault**)
 - Which may cause a feature to not display its outputs (**Feature Fault**)
 - And the application will not satisfy the acceptance criteria for the associated user story (**User Requirements Fault**)
- User stories attempt to catch most of these faults by testing that the user can complete the required tasks

Analysis

- The key analysis task in application testing is the analysis of the user interface
- In unit testing, a method call requires no analysis: the parameters, their types, and order are all well defined
- For a user interface, the inputs and outputs are located on a screen, and are represented using text (or other user interface metaphors for the underlying data types)
- The location of these interface elements may change based on the screen size or orientation, and the user must interact with the application using other interface elements (such as keyboard shortcuts, buttons, links, etc)

Data Transformations

- Designing automated tests for an application involves finding a way to
 - locate necessary interface elements
 - Enter and extract data in a way compatible with the data representation used

Data Transformations

- Designing automated tests for an application involves finding a way to
 - locate necessary interface elements
 - Enter and extract data in a way compatible with the data representation used
- For example, numeric data will often be represented as text on the screen:
 - **Input:** Each numeric key pressed is interpreted as a digit, added to string displayed on the screen. String then converted to integer for use in the program. Test tool must convert required integer inputs to strings
 - **Output:** integer is converted to a text string for display. The test tool needs to retrieve this string, and convert it to an integer, before checking its value

Data Transformations

- Designing automated tests for an application involves finding a way to
 - locate necessary interface elements
 - Enter and extract data in a way compatible with the data representation used
- For example, numeric data will often be represented as text on the screen:
 - **Input:** Each numeric key pressed is interpreted as a digit, added to string displayed on the screen. String then converted to integer for use in the program. Test tool must convert required integer inputs to strings
 - **Output:** integer is converted to a text string for display. The test tool needs to retrieve this string, and convert it to an integer, before checking its value
- Numeric inputs and outputs may also use different screen elements as *metaphors* for the value – e.g. dials, sliders, pull-down menus
 - Test tool must manipulate these screen elements to provide inputs
 - And convert from the displayed representation to check the outputs

Programming Interfaces and User Interfaces

- Sample programming interface:

```
boolean check(int volume, boolean highSafety)  
    throws FuelException
```

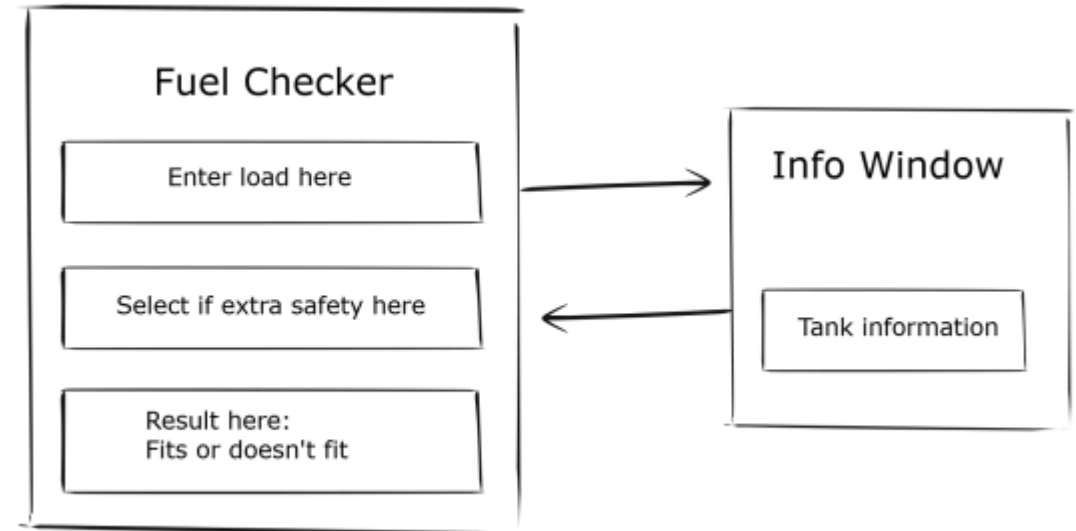
Programming Interfaces and User Interfaces

- Sample programming interface:

```
boolean check(int volume, boolean highSafety)
    throws FuelException
```

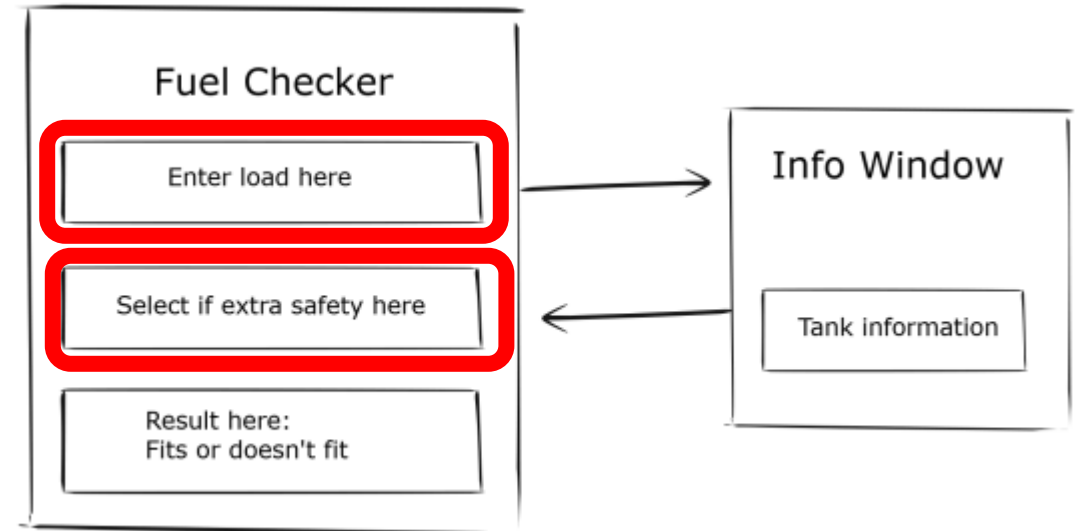
- Interpretation:
 - The input for the volume of fuel is passed in the parameter **volume**
 - The input for whether the fuel is highly volatile and requires extra space for safety is passed in the parameter **highSafety**.
 - The output is returned in the method **return value**.
 - If an error occurs, then the method will raise a **FuelException**.
 - The method is identified by its name: **example.FuelChecker.check()**
- Easy to call the method with the required inputs & check the results

Software User Interface



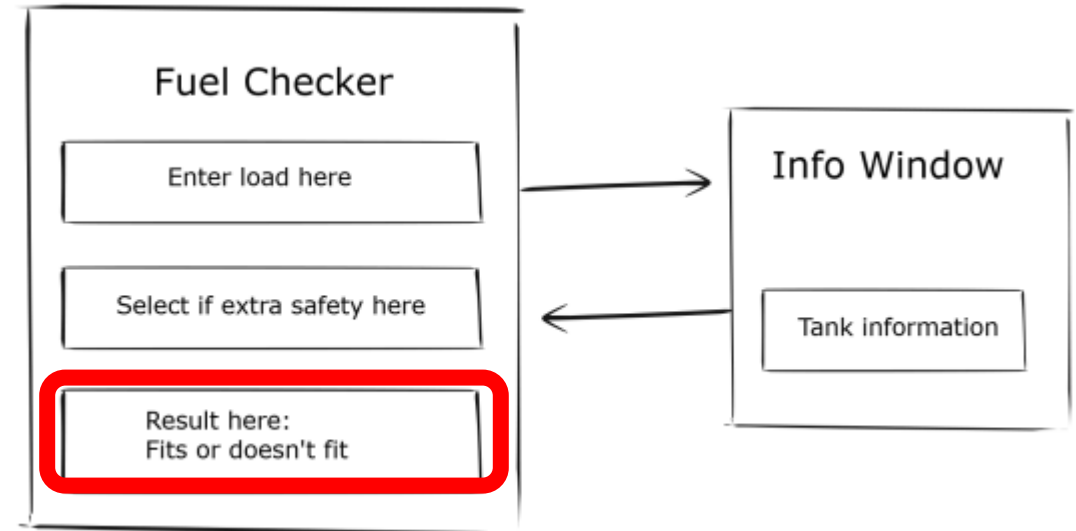
Software User Interface

- Interpretation:
 - The inputs for the **volume** of fuel, and whether **extra safety** space is required, are located on the screen
 - By convention, input prompts are either above or to the left of the entry fields
 - Prompts may be short and cryptic, as shown, or long and descriptive



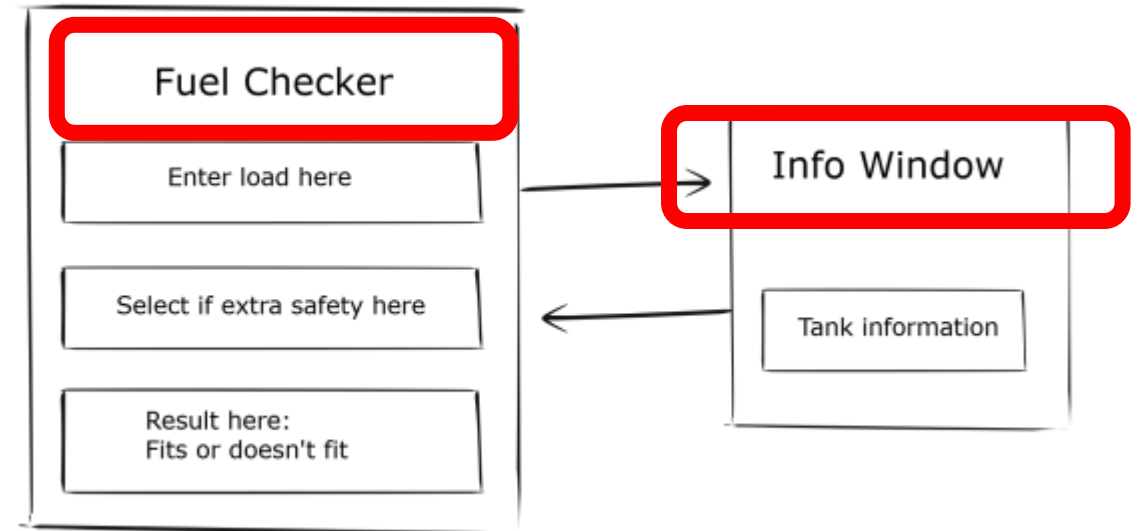
Software User Interface

- Interpretation:
 - The inputs for the volume of fuel, and whether extra safety space is required, are located on the screen
 - By convention, input prompts are either above or to the left of the entry fields
 - Prompts may be short and cryptic, as shown, or long and descriptive
 - The **result** is shown in a separate field, in this case it is below the entry fields
 - It may have a prompt, or a title, or may just be identified by context



Software User Interface

- Interpretation:
 - The inputs for the volume of fuel, and whether extra safety space is required, are located on the screen
 - By convention, input prompts are either above or to the left of the entry fields
 - Prompts may be short and cryptic, as shown, or long and descriptive
 - The result is shown in a separate field, in this case it is below the entry fields
 - It may have a prompt, or a title, or may just be identified by context
 - **Screens** identified by their titles



Automated Test Program Difficulties

- Interacting based purely on the absolute (x,y) **location** of each element on the screen can prove problematic
 - Fails if layout changes – e.g. if browser window size or zoom changes
 - Particular problem for responsive interface design

Automated Test Program Difficulties

- Interacting based purely on the absolute (x,y) location of each element on the screen can prove problematic
 - Fails if layout changes – e.g. if browser window size or zoom changes
 - Particular problem for responsive interface design
- Using the **prompts** also problematic
 - The prompt is usually located by various conventions
 - Prompt for a dial may be within the element
 - Prompt for a text box may be to the left of or above the textbox, or even in the textbox in a grey font

Automated Test Program Difficulties

- Interacting based purely on the absolute (x,y) location of each element on the screen can prove problematic
 - Fails if layout changes – e.g. if browser window size or zoom changes
 - Particular problem for responsive interface design
- Using the prompts also problematic
 - The prompt is usually located by various conventions
 - Prompt for a dial may be within the element
 - Prompt for a text box may be to the left of or above the textbox, or even in the textbox in a grey font
- It may also prove difficult to locate non-text elements by prompt – for example, a warning message prompt may be a **warning flag icon**

Interacting With Screen Elements

- These user interface elements may be easy for a user to locate and interpret, based on prompts, convention, or just experience
- But locating them, and interpreting the data representation, provide significant challenges for test automation
- Unless there is detailed documentation available, specifying the interface in detail, this requires further investigation before black-box techniques can be used to select test data
- An application may use a wide range of on-screen interface elements (such as pulldown menus, popup menus, sliders, drag-and-drop, keypad gestures etc.), and even off-screen interfaces such as audio, visual, and haptic interfaces; these are significantly harder to test

Interacting with HTML Elements

- For a well designed (DFT) web application, use:
 - HTML title for each page
 - HTML id for each element

Interacting with HTML Elements

- For a well designed (DFT) web application, use:
 - HTML title for each page
 - HTML id for each element
- If not available, more challenging:
 - elements may be identified by their contents (e.g. the text on a button)
 - or in the worst case by their relative or absolute location on the page (or, more accurately, in the currently loaded DOM model)

Interacting with HTML Elements

- For a well designed (DFT) web application, use:
 - HTML title for each page
 - HTML id for each element
- If not available, more challenging:
 - elements may be identified by their contents (e.g. the text on a button)
 - or in the worst case by their relative or absolute location on the page (or, more accurately, in the currently loaded DOM model)
- The Selenium library is representative of other web automation tools in that it provides a number of methods to find HTML elements and interact with them:
 - A **By** object is used to represent search criteria
 - HTML elements may be identified by a number of characteristics: id, classname, css selector, anchor link text, anchor partial link text match, name, tag or xpath
 - The `WebDriver.findElement()/findElements()` methods use the `By` to return the first or all matching elements.

Data Representation

- The other key issue facing the tester
- Example: a **text string that represents an integer**

Data Representation

- The other key issue facing the tester
- Example: a **text string that represents an integer**

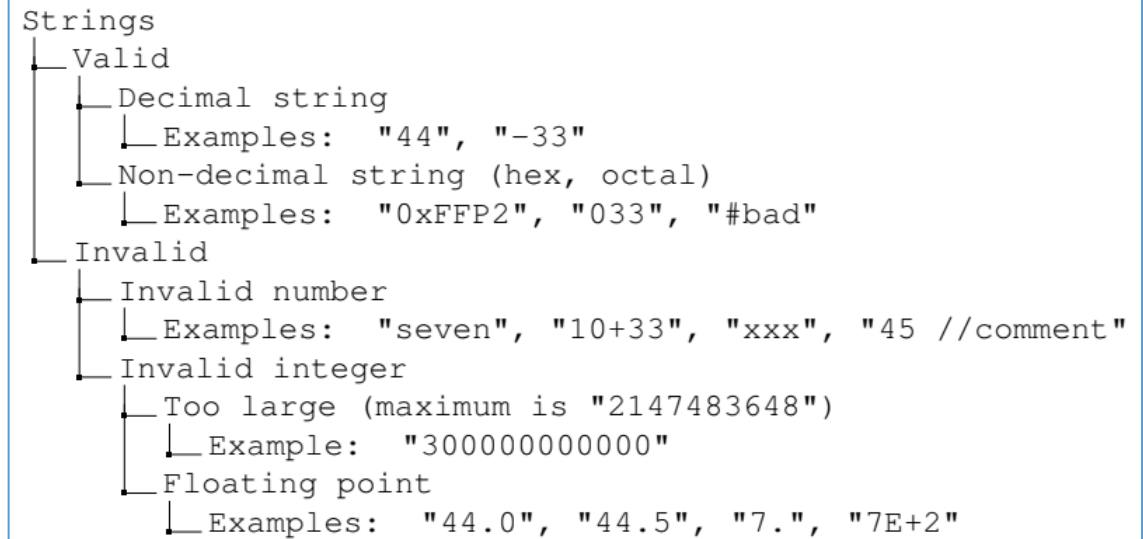
```
Strings
├── Valid
│   ├── Decimal string
│   │   └── Examples: "44", "-33"
│   └── Non-decimal string (hex, octal)
│       └── Examples: "0xFFP2", "033", "#bad"
```


Data Representation

- The other key issue facing the tester
- Example: a **text string that represents an integer**

```
Strings
├── Valid
│   ├── Decimal string
│   │   └── Examples: "44", "-33"
│   └── Non-decimal string (hex, octal)
│       └── Examples: "0xFFP2", "033", "#bad"
└── Invalid
    ├── Invalid number
    │   └── Examples: "seven", "10+33", "xxx", "45 //comment"
    ├── Invalid integer
    │   ├── Too large (maximum is "2147483648")
    │   │   └── Example: "3000000000000"
    │   └── Floating point
    │       └── Examples: "44.0", "44.5", "7.", "7E+2"
```

Data Representation



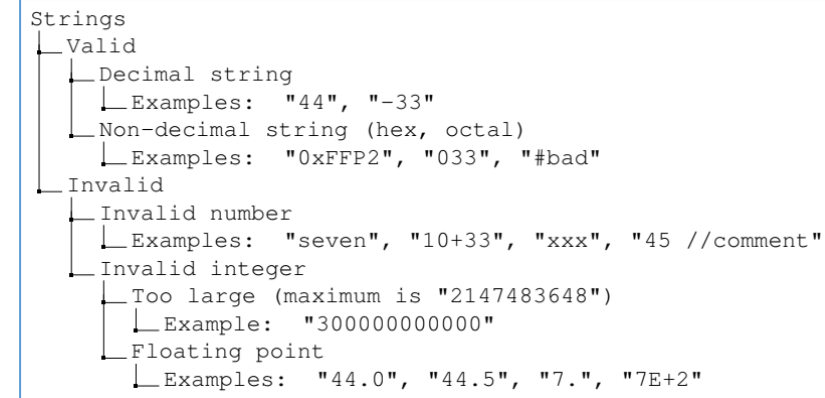
- Large number of ways in which a string can be formed that can or cannot be converted to an integer in a program
- Some of the following analysis is Java-specific, and some is common to many languages
- The term "*valid strings*" is used to indicate strings which can be converted to an integer value by using a single standard Java library call: `Integer.parseInt()` or `Integer.decode()`

Some Issues

```
Strings
├── Valid
│   ├── Decimal string
│   │   └── Examples: "44", "-33"
│   └── Non-decimal string (hex, octal)
│       └── Examples: "0xFFP2", "033", "#bad"
└── Invalid
    ├── Invalid number
    │   └── Examples: "seven", "10+33", "xxx", "45 //comment"
    └── Invalid integer
        ├── Too large (maximum is "2147483648")
        │   └── Example: "3000000000000"
        └── Floating point
            └── Examples: "44.0", "44.5", "7.", "7E+2"
```

- Strings with whitespace (e.g. " 44 ") cannot be converted by Integer.parseInt() or Integer.decode() without pre-processing – e.g. String.strip()
- Need to test if an application uses non-standard integers representations, that are not supported by Java
- What is important is **not** what method has been used by the programmer to do the conversion, but what the application **should** support

Some Issues



- Strings with whitespace (e.g. " 44 ") cannot be converted by Integer.parseInt() or Integer.decode() without pre-processing – e.g. String.strip()
- Need to test if an application uses non-standard integers representations, that are not supported by Java
- What is important is **not** what method has been used by the programmer to do the conversion, but what the application **should** support
- **Common-sense** is frequently used, but liable to lead to errors:
 - for example, a programmer may regard the difference between "033" (octal) and "33" (decimal) as obvious, but a customer might well regard this as data corruption

Test Coverage Items

- In basic user story testing each **acceptance criterion** is a test coverage item
- Other standard techniques (e.g. UML Use Cases and IEEE Scenarios):
 - In UML Use Case testing, each **scenario** is a test coverage item
 - In IEEE scenario testing, the **main** and **alternative scenarios** are all test coverage items

Test Coverage Items/User Story Testing

- User story testing is easily extended by considering not only the acceptance criteria but also the equivalence partitions, boundary values, or combinations of the input values as test coverage items. Typically this leads to a large number of additional error test cases (e.g. many ways of misrepresenting an integer)

Test Coverage Items/User Story Testing

- User story testing is easily extended by considering not only the acceptance criteria but also the equivalence partitions, boundary values, or combinations of the input values as test coverage items. Typically this leads to a large number of additional error test cases (e.g. many ways of misrepresenting an integer)
- A wide range of error test cases is often added to user story testing on an ad-hoc basis
 - A systematic approach is likely to provide more rigorous testing

Test Coverage Items/User Story Testing

- User story testing is easily extended by considering not only the acceptance criteria but also the equivalence partitions, boundary values, or combinations of the input values as test coverage items. Typically this leads to a large number of additional error test cases (e.g. many ways of misrepresenting an integer)
- A wide range of error test cases is often added to user story testing on an ad-hoc basis
 - A systematic approach is likely to provide more rigorous testing
- If unit testing has not been performed beforehand, application testing using equivalence partition/boundary value analysis/decision tables may be used as a substitute
 - Not as rigorous as testing individual methods or classes via their programming interface
 - The equivalence partition/boundary value analysis/decision tables technique will identify the test coverage items

Test Coverage Items/User Story Testing

- User story testing is easily extended by considering not only the acceptance criteria but also the equivalence partitions, boundary values, or combinations of the input values as test coverage items. Typically this leads to a large number of additional error test cases (e.g. many ways of misrepresenting an integer)
- A wide range of error test cases is often added to user story testing on an ad-hoc basis
 - A systematic approach is likely to provide more rigorous testing
- If unit testing has not been performed beforehand, application testing using equivalence partition/boundary value analysis/decision tables may be used as a substitute
 - Not as rigorous as testing individual methods or classes via their programming interface
 - The equivalence partition/boundary value analysis/decision tables technique will identify the test coverage items
- In addition to the user stories agreed with the customer, tester may identify extra test coverage items using EP, BVA, DT, OO test techniques, and experience

Test Cases

- Each test coverage item is a test case
- Additional test cases may be added to provide a broader range of input values, especially for errors (as seen in the discussion on data representation)

Implementation

- Basic Structure

- Setup
- Teardown
- Tests
- Post-method processing

```
// TestNG import statements
// Selenium import statements

public class WebTestTemplate {

    WebDriver driver;    // used to drive the application
    Wait<WebDriver> wait; // used to wait for screen
                        // elements to appear
    String url=System.getProperty("url"); // the URL for
                                        // the web application to test

    @BeforeClass
    public void setupDriver() throws Exception {
        // This runs before any other method in the class:
        open web browser
    }

    @AfterClass
    public void shutdown() {
        // This runs after all other methods:
        close the web browser
    }

    // Tests go here

    @Test(timeOut=20000)
    public void test_Tnnn() {
    }

    // Post test method processing goes here

    @AfterMethod
    public void postMethodProcessing() {
        // Runs after each test method:
        return to a common start screen
    }

}
```

Setup

```
WebDriver driver;      // used to drive the application
Wait<WebDriver> wait;  // used to wait for screen
                       // elements to appear
```

- Browser controlled using Selenium WebDriver

```
@BeforeClass
public void setupDriver() throws Exception {
    System.out.println("Test started at: "+LocalDateTime.now());
    if (url==null)
        throw new Exception("Test URL not defined: use -Durl=<url>");
    System.out.println("For URL: "+url);
    System.out.println();
    // Create web driver (this code uses chrome)
    driver = new ChromeDriver();
    // Create wait
    wait = new WebDriverWait( driver, Duration.ofSeconds(5) );
    // Open web page
    driver.get( url );
}
```

- Before any tests are run, the Selenium WebDriver must be opened using an @BeforeClass method
- In this example, the Chrome browser is used. Selenium automatically downloads the program chromedriver[.exe] which is required to run the tests

```
WebDriver driver;  
Wait<WebDriver> wait;
```

```
throws Exception {  
    System.out.println("Test started at: "+LocalDateTime.now());  
    if (url==null)  
        throw new Exception("Test URL not defined: use -Durl=<url>");  
    System.out.println("For URL: "+url);  
    System.out.println();  
    // Create web driver (this code uses chrome)  
    driver = new ChromeDriver();  
    // Create wait  
    wait = new WebDriverWait( driver, Duration.ofSeconds(5) );  
    // Open web page  
    driver.get( url );  
}
```

- It is necessary to wait for items to appear in a browser, as the web browser runs asynchronously from the test program
- So a Selenium WebDriverWait object is created

```
WebDriver driver;  
Wait<WebDriver> wait;
```

```
@BeforeClass  
public void setupDriver() throws Exception {  
    System.out.println("Test started at: "+LocalDateTime.now());  
    if (url==null)  
        throw new Exception("Test URL not defined: use -Durl=<url>");  
    System.out.println("For URL: "+url);  
    System.out.println();  
    // Create web driver (this code uses chrome)  
    driver = new ChromeDriver();  
    // Create wait  
    wait = new WebDriverWait( driver, Duration.ofSeconds(5) );  
    // Open web page  
    driver.get( url );  
}
```

Setup

```
WebDriver driver;        // used to drive the application
Wait<WebDriver> wait;     // used to wait for screen
                           // elements to appear
String url=System.getProperty("url"); // the URL for
                                   // the web application to test
```

- The browser must be triggered to open the web page

```
@BeforeClass
public void setupDriver() throws Exception {
    System.out.println("Test started at: "+LocalDateTime.now());
    if (url==null)
        throw new Exception("Test URL not defined: use -Durl=<url>");
    System.out.println("For URL: "+url);
    System.out.println();
    // Create web driver (this code uses chrome)
    driver = new ChromeDriver();
    // Create wait
    wait = new WebDriverWait( driver, Duration.ofSeconds(5) );
    // Open web page
    driver.get( url );
}
```

Setup

```
WebDriver driver; // used to drive the application
Wait<WebDriver> wait; // used to wait for screen
// elements to appear
String url=System.getProperty("url"); // the URL for
// the web application to test
```

```
@BeforeClass
public void setupDriver() throws Exception {
    System.out.println("Test started at: "+LocalDateTime.now());
    if (url==null)
        throw new Exception("Test URL not defined: use -Durl=<url>");
    System.out.println("For URL: "+url);
    System.out.println();
}
```

- The URL is passed as a parameter to the test, using `-Durl=whatever` on the command line
- This allows for different versions of a web application to be more easily tested, as different URLs may be used for each version
- The URL parameter is retrieved using `System.getProperty()` as shown at top

ne)

ofSeconds(5));

Teardown

- When the tests are all finished, the web browser should be closed, using an `@AfterClass` method

```
53     @AfterClass  
54     public void shutdown() {  
55         driver.quit();  
56     }
```

Interacting with HTML Elements

- Each HTML element has its own attributes and behaviour
- These need to be understood to develop effective tests
- Example uses Selenium methods for interacting with HTML elements:
 - `anchor.click()`
 - `button.click()`
 - `checkbox.click()`
 - `checkbox.isSelected()`
 - `div.getAttribute("innerHTML")`
 - `element.id`
 - `input, type=text, element.getAttribute("value")`
 - `input, type=text, element.sendKeys()`
 - `page.title`

Tips for the Tester

- In practice, frequent reference to the HTML DOM and HTML specifications and the Selenium API documentation is needed to find the necessary API calls, and to ensure that HTML elements are being accessed correctly
- Sometimes a tester may need to perform trials with different Selenium calls or different attributes to make sure the test is working correctly
- When developing tests, it can be useful to print different attributes of an element, or the results of different Selenium methods, to the console for inspection
- Note: The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a web page

Test Output Messages

- Two extra elements of the test output:
 - Logging the time and test item
 - WebDriver startup messages (in this example, ChromeDriver)

```
Test started at: 2020-09-24T19:15:14.618050100
For URL: ch10\fuelchecker\fuelchecker.html

Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-
refs/branch-heads/4147@{#310}) on port 1388
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for
suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[1600971316.989][WARNING]: This version of ChromeDriver has not been tested with
Chrome version 85.
Sep 24, 2020 7:15:18 P.M. org.openqa.selenium.remote.ProtocolHandshake
createSession
```

Use of Extra Messages

```
Test started at: 2020-09-24T19:15:14.618050100
For URL: ch10\fuelchecker\fuelchecker.html

Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-
refs/branch-heads/4147@{#310}) on port 1388
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for
suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[1600971316.989][WARNING]: This version of ChromeDriver has not been tested with
Chrome version 85.
Sep 24, 2020 7:15:18 P.M. org.openqa.selenium.remote.ProtocolHandshake
createSession
```

- Valuable for application/system testing to record:
 - date and time of the test
 - details of the application being tested (the test item)
- Printed by the @BeforeClass method of the test program
- ChromeDriver messages show:
 - version running
 - a warning that only local connections are allowed by Chrome when running under the control of ChromeDriver
 - confirmation ChromeDriver has successfully connected to Chrome browser

Record and Playback Testing

- Alternative form of automated user interface testing
- Record a manual interaction with an application (**interactions** and **responses**)
- The recorded user **interactions** can then be automatically played back to the application
- And the actual **responses** compared with the recorded responses
- Regression testing: verify existing software behaviour not broken after a change
- Tools:
 - Record the user input automatically
 - Require the user to identify the important output fields
 - Then the data in these important fields can then be recorded automatically
- Most web-based test tools, such as Selenium, provide such a facility
- Some customisation supported (for example, date fields may change)
- Testing stories usually requires a programmatic approach

Evaluation

- In a well-established development process, the classes that implement the fuel checker functionality should have been unit tested in advance
- So it is likely that only user-interface related faults will cause failures of the system test
- Some limitations of user story testing are explored next

Limitations

- Faults we will use:
 - **Navigation fault:** a fault is inserted that prevents the application moving from one screen to another
 - **Screen Element fault:** a fault is inserted into a screen element
 - **Software Feature fault:** a fault is inserted into the software feature to check the fuel load
- The application is implemented in HTML and Javascript, and can be viewed in the file Fuelchecker.html
- Note: the main functionality is implemented as a “Single Page App” and uses Javascript to manipulate the HTML within the page

Navigation Fault

- A fault is inserted into the navigation from the Results screen to the Exit screen in the fuelchecker application
- The exitlink href is set to call the Javascript Exit() function when the hyperlink Exit is clicked on the screen

```
<li style ="display: inline"><a id="exitlink"  
    href="javascript:Exit()" style="color:black">Exit</a></li>
```

Fault Details

- When the application switches to the results page, correct code:

```
document.getElementById("result").value = result;

document.getElementById("Enter").style.display='none';
document.getElementById("Continue").style.display='block';
document.getElementById("subhead").innerHTML = "Results";
document.getElementById("result").style.display='inline';
document.getElementById("litres").disabled = true;
document.getElementById("highsafety").disabled = true;
document.title = "Results";
```

- Insert a fault: extra line of code that removes the href attribute:

```
document.getElementById("highsafety").disabled = true;
document.getElementById("exitlink").removeAttribute("href"); //
    navigation fault
document.title = "Results";
```

- Consequence: nothing happens when the hyperlink Exit is clicked

Navigation Fault Test Result

```
PASSED: testEnterCheckView("T1", "1000", false, "Fuel fits in tank.")
PASSED: testEnterCheckView("T2", "400", true, "Fuel fits in tank.")
PASSED: testEnterCheckView("T3", "2000", false, "Fuel does not fit in tank.")
PASSED: testEnterCheckView("T4", "1000", true, "Fuel does not fit in tank.")
PASSED: testEnterCheckView("T7", "xxx", true, "Invalid data values.")
PASSED: test_T5
PASSED: test_T6
=====
Command line suite
Total tests run: 7, Passes: 7, Failures: 0, Skips: 0
=====
```


- All User Story tests passed
- This link never used after the results page is displayed
- “Navigation Testing” would find this fault (check every page transition)

Screen Element Fault

- A fault is inserted into the handling of the screen element highsafety in the fuelchecker application
- It is disabled when it should be enabled when returning back to the main screen
- Consequence: When the Continue button is clicked, then the applications returns to the main screen with the highSafety checkbox disabled. The user is now unable to select/deselect the checkbox as required for the next data entry

Fault Details

- Correct code:

```
document.getElementById("Info").style.display='block';  
document.getElementById("Enter").style.display='block';  
document.getElementById("Continue").style.display='none';  
document.getElementById("result").style.display='none';  
document.getElementById("litres").disabled = false;  
 document.getElementById("highsafety").disabled = false;  
document.title = "Fuel Checker";  
document.getElementById("subhead").innerHTML = "Enter Data";
```

- The fault:

```
document.getElementById("highsafety").disabled = true; // Screen  
element fault
```

Screen Element Fault Test Result

- Four User Story tests fail:
T1,T2,T3,T4 reply on
highsafety
- T7 uses the highsafety button
but does not rely on it
working properly
- High impact faults like this
are likely to cause multiple
user story tests to fail, but
more subtle faults may not be
found by simple user story
tests

```
PASSED: testEnterCheckView("T7", "xxx", true, "Invalid data values.")
PASSED: test_T5
PASSED: test_T6
FAILED: testEnterCheckView("T1", "1000", false, "Fuel fits in tank.")
java.lang.AssertionError: expected [Fuel fits in tank.] but found [Invalid data
values.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)

FAILED: testEnterCheckView("T2", "400", true, "Fuel fits in tank.")
java.lang.AssertionError: expected [Fuel fits in tank.] but found [Invalid data
values.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)

FAILED: testEnterCheckView("T3", "2000", false, "Fuel does not fit in tank.")
java.lang.AssertionError: expected [Fuel does not fit in tank.] but found [
Invalid data values.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)

FAILED: testEnterCheckView("T4", "1000", true, "Fuel does not fit in tank.")
java.lang.AssertionError: expected [Fuel does not fit in tank.] but found [
Invalid data values.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)
=====
Command line suite
Total tests run: 7, Passes: 3, Failures: 4, Skips: 0
=====
```

Software Feature Fault

- When a user interacts with an application, they generally use one or more features to achieve their tasks (as documented in the user stories)
- The fuelchecker application has three software features:
 1. Check a fuel load.
 2. Display the tank sizes.
 3. Exit.
- Insert fault inserted into “Check a fuel load” feature:
 - Correct value not passed from the user interface to software feature code
 - Consequence: the wrong value will be returned

Fault Details

- Correct Code:

```
104     var lss = (document.getElementById("litres")).value.trim();
105     if (lss == parseInt(lss,10))
106         ls = parseInt(lss,10);
107     else
108         ls = "Invalid";
```

- The Fault (line 106):

```
ls = parseInt(lss,10)*10; // Fault: incorrect multiplication
```


Software Feature Fault Test Result

- Two of the tests fail
- The software feature still works for some input values, but not for others
- In many cases, this type of fault can create very subtle changes in behaviour that would not be caught by a simple user story test

```
PASSED: testEnterCheckView("T3", "2000", false, "Fuel does not fit in tank.")
PASSED: testEnterCheckView("T4", "1000", true, "Fuel does not fit in tank.")
PASSED: testEnterCheckView("T7", "xxx", true, "Invalid data values.")
PASSED: test_T5
PASSED: test_T6
FAILED: testEnterCheckView("T1", "1000", false, "Fuel fits in tank.")
java.lang.AssertionError: expected [Fuel fits in tank.] but found [Fuel does not
fit in tank.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)

FAILED: testEnterCheckView("T2", "400", true, "Fuel fits in tank.")
java.lang.AssertionError: expected [Fuel fits in tank.] but found [Fuel does not
fit in tank.]
    at example.FuelCheckerWebStoryTest.testEnterCheckView(
        FuelCheckerWebStoryTest.java:97)

=====
Command line test
Tests run: 7, Failures: 2, Skips: 0
=====
```

Strengths and Weaknesses

- Testing that the acceptance criteria for the user stories are met provides an assurance that the basic functionality works for the user
- User story tests are unlikely to find all of the navigation, screen element, and software feature faults in an application
- They will only find those faults that are exposed by the user stories and acceptance criteria documented for the application

Key Points

- Systems are tested over their system interface, which is usually significantly more complex than passing parameters to a method
- Application testing is a form of system testing
- Testing applications (web-based, desktop-based, or mobile apps) over their GUI requires some form of unique identifiers so that the required interface elements can be referenced for input and output
- Web-based testing over a browser requires browser support, to allow automated tests to simulate user input, and read the output
- User story testing will probably not provide extensive testing of error cases, or find all the faults in an application

Unit Testing vs Application Testing

- Application testing is a far more complex activity than unit testing
- The user interface inserts an additional translation layer between the user and the code
- The complexities are associated with manipulating this correctly to simulate the actions of a user
- The key differences in testing:
 - Locating the inputs
 - Locating the outputs
 - Automating the tests

Locating the Inputs

- Unit testing:
 - Inputs are located by their position in the method/function call
 - (Some other languages support named parameters)
 - Inputs may also be class attributes, or accessed from an external resource such as a file, database, or physical device
- Application testing:
 - Inputs are located as elements on the screen
 - They may be presented using various analogues (such as a slider for a number)
 - Trial runs of the application used to identify these
 - Well designed application has unique identifiers, at least with regards to each screen,
 - Otherwise absolute/relative position on the screen may be required
 - As for unit testing, the input may also come from external sources

Locating the Outputs

- Unit testing:
 - Outputs may be returned by the method/function call
 - Or side-effects (i.e. class attribute changes, or sent to an external resource)
- Application testing:
 - Outputs are represented in the same way as the inputs as elements on the screen
 - Outputs may be presented over multiple screens
 - Same issues apply as for locating inputs
 - As for unit testing, the outputs may also be sent to external resources (for example, a database)

Automating the Tests

- Unit testing, automated implementation is usually straightforward:
 - locate a set of input values
 - call the method/functions
 - get the return value (actual results)
 - check against the expected results.
- Application testing:
 - each input element must be located separately, possibly on different screens, and manipulated correctly to represent the input value
 - the application functionality must then be triggered, sometimes via a simple button press but often requiring more manipulation of screen elements
 - The output must then be located and extracted from the screen
 - And then interpreted to generate the actual result
 - Before the task of comparing them against the expected results can be performed

Notes for Experienced Testers

- Develop test code directly from the user stories and acceptance criteria
 - Run the application at the same time to find HTML id's and element types
 - Mentally determine the typical data values to use
- Impossible to review the testers work without redoing all the analysis
 - To alleviate this, add descriptive comments in the test code
- Additional tests often written to cover potential data representation problems, especially for input data errors
- Additional tests will also be developed based on the tester's experience in order to try and overcome some of the weaknesses in basic user story testing

More Notes

- The time spent on developing and executing tests is usually closely related to the value of the application
- This is based on the risk of failure
- e-commerce website, small company, low value items, a few customers:
 - code probably not unit tested
 - Basic user stories of listing the items, adding them to a shopping cart, checking out, and viewing the order status probably tested manually with no documentation
- At the other extreme, a website for a bank:
 - probably have all the code unit tested
 - Probably use extensive automated application testing (including user story testing, and some of the other approaches discussed) to make sure that the bank's customers are unlikely to experience problems