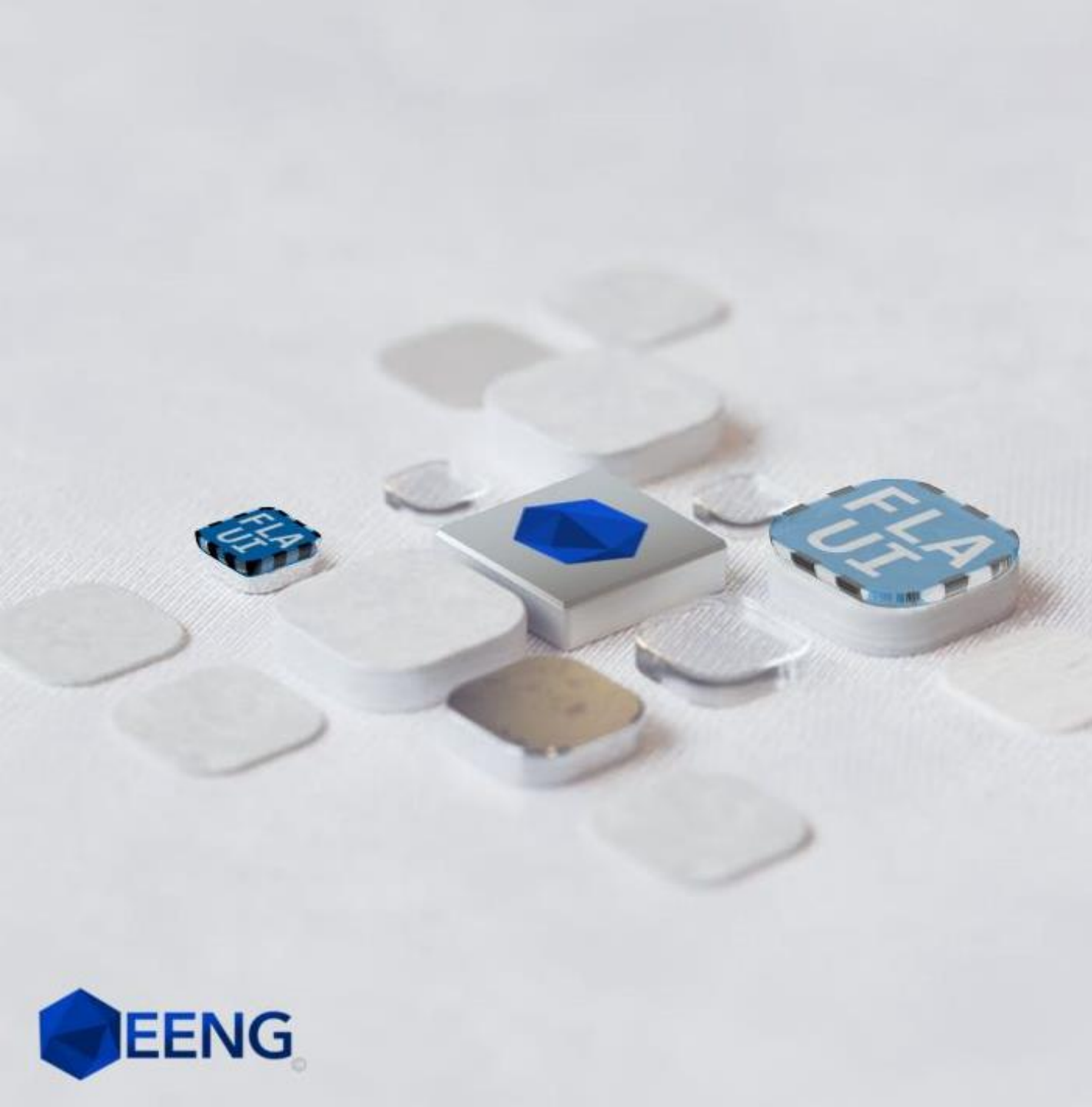


Integrationstests mit C#



Hi!

ich bin Caglar Özdemir -
36 Jahre jung und schon
knapp 12 Jahre im
Coding-Game
#visualstudio
#cpp #csharp
#WPF #QML





Wer bist du ?
Was machst du ?
Was sind deine Erwartungen im Kurs?
Was ist deine Lieblings-IDE?

Round 1

Einführung

Was sind Softwaretests?

Was sind Integrationstests?

Unterschiede zu Unittests

Arten von Integrationstests

- Service-Level
- Database
- Dependency Injections
- UI
- End to End

Verwendbare IDEs und Einrichtungen

Ab wann sind Integrationstests sinnvoll?

Automatisierung von UIs

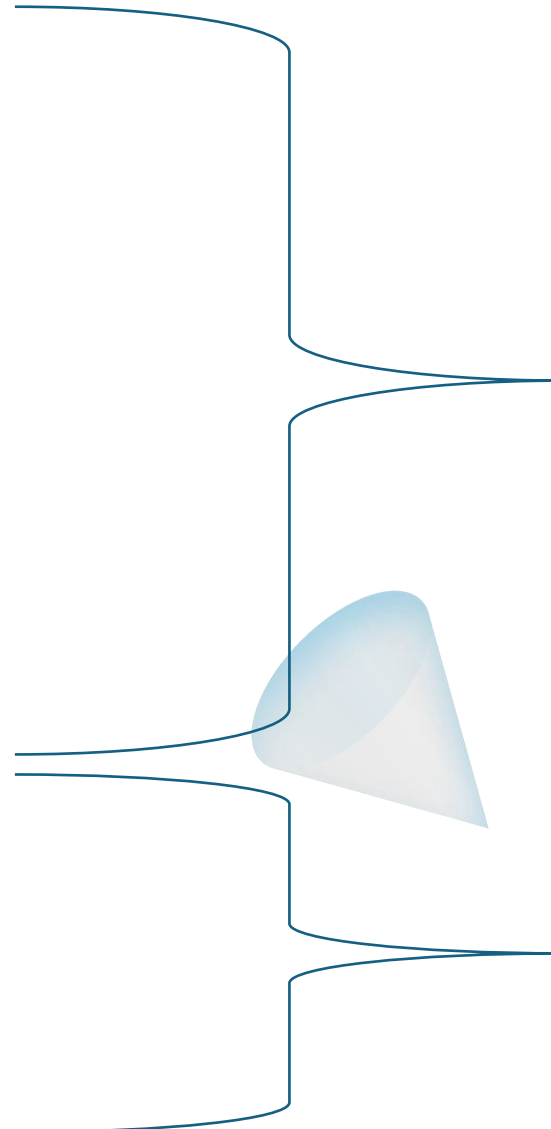
Flauti

FlautiInspect

Kontrolltypen

Accessoren

Übungen



Basics

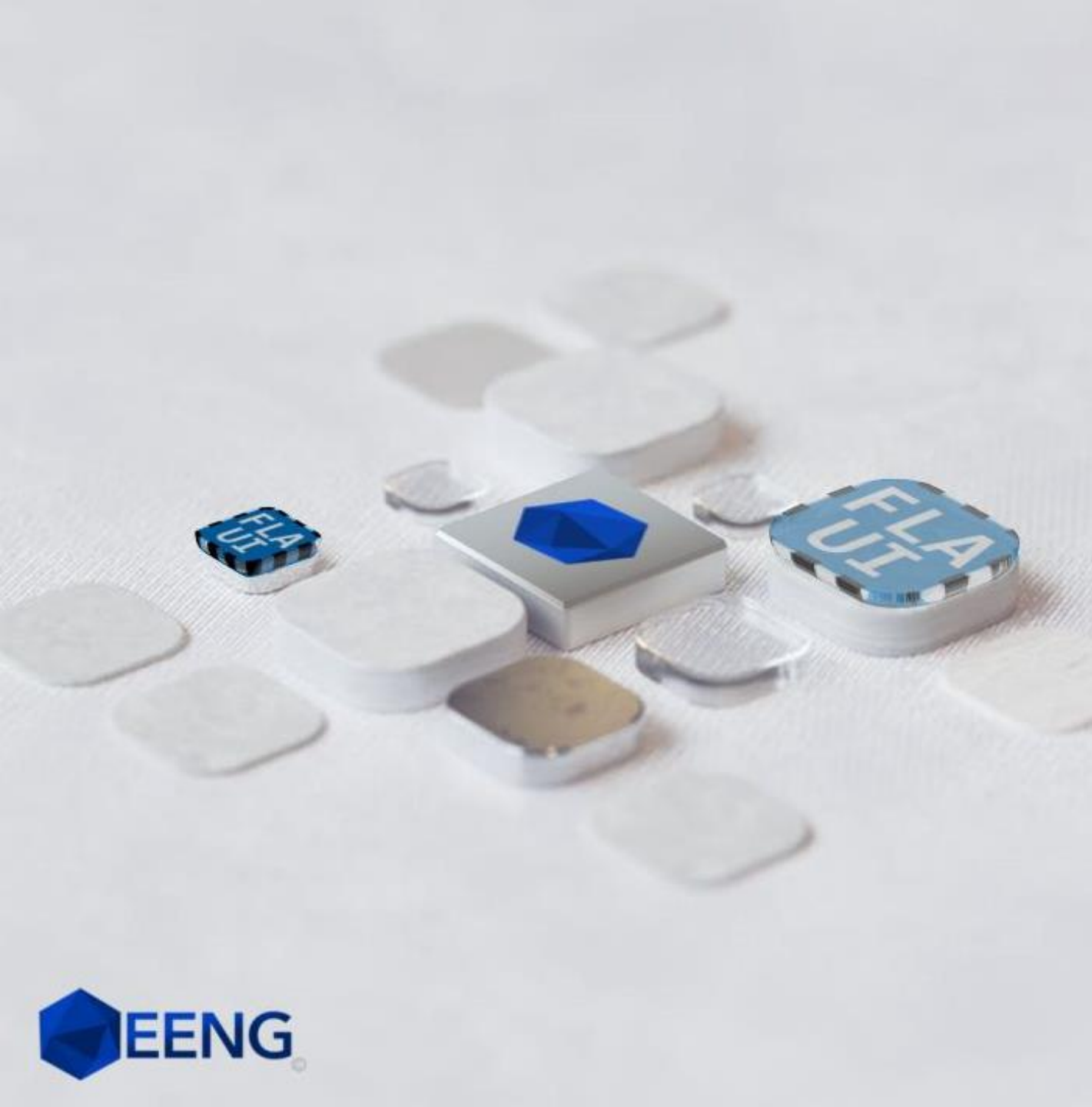
UI Automatisierung

Round 2

Unterschied Ranorex
Was ist Ranorex?
Übungen mit Ranorex
CI/CD
Azure Devops
Test Coverage
Best Practise
Kombination mit Copilot

Abschlussübung
Q&A

Erweiterte Kenntnisse



Was sind
Integrationstests?

Was sind Softwaretests?

Definition:

- Softwaretesting ist der Prozess, bei dem eine Anwendung oder ein System überprüft wird, um Fehler zu finden und die Qualität sicherzustellen.

Ziele:

- Fehler frühzeitig erkennen.
- Sicherstellen, dass die Software die Anforderungen erfüllt.
- Reduzieren von Risiken bei der Einführung von Software.

Wichtige Konzepte:

- Testplanung.
- Testausführung.
- Dokumentation von Ergebnissen.

Überblick Testarten

- **Unittests:** Testen einzelne Funktionen oder Methoden.
- **Integrationstests:** Testen die Zusammenarbeit zwischen Modulen.
- **Systemtests:** Testen das gesamte System als Einheit.
- **End-to-End-Tests:** Überprüfen gesamte Geschäftsprozesse.
- **Weitere Testarten:**
 - Regressionstests.
 - Akzeptanztests.
 - Last- und Performancetests.

Unittests

Definition:

- Testen von kleinen, isolierten Codeeinheiten (z. B. Funktionen, Methoden).

Merkmale:

- Schnell und einfach zu implementieren.
- Isoliert – keine Abhängigkeiten von anderen Modulen.

```
[Fact]
0 references
public void Add_TwoIntegers_ExpectedSum()
{
    // Arrange
    var calculator = new Calculator();
    var expected = 3;

    // Act
    var result = calculator.Add(1, 2);

    // Assert
    Assert.Equal(expected, result);
}
```

Systemtests

Definition:

- Testen des gesamten Systems als Einheit.

Merkmale:

- Überprüfung aller Module und deren Interaktionen.
- Testen aus der Perspektive des Endbenutzers.

Beispiel:

- Ein Systemtest könnte überprüfen, ob eine Webanwendung von der Anmeldung bis zur Bestellung korrekt funktioniert.

Integrationstests

Definition:

- Testen der Interaktion zwischen verschiedenen Modulen oder Subsystemen.

Merkmale:

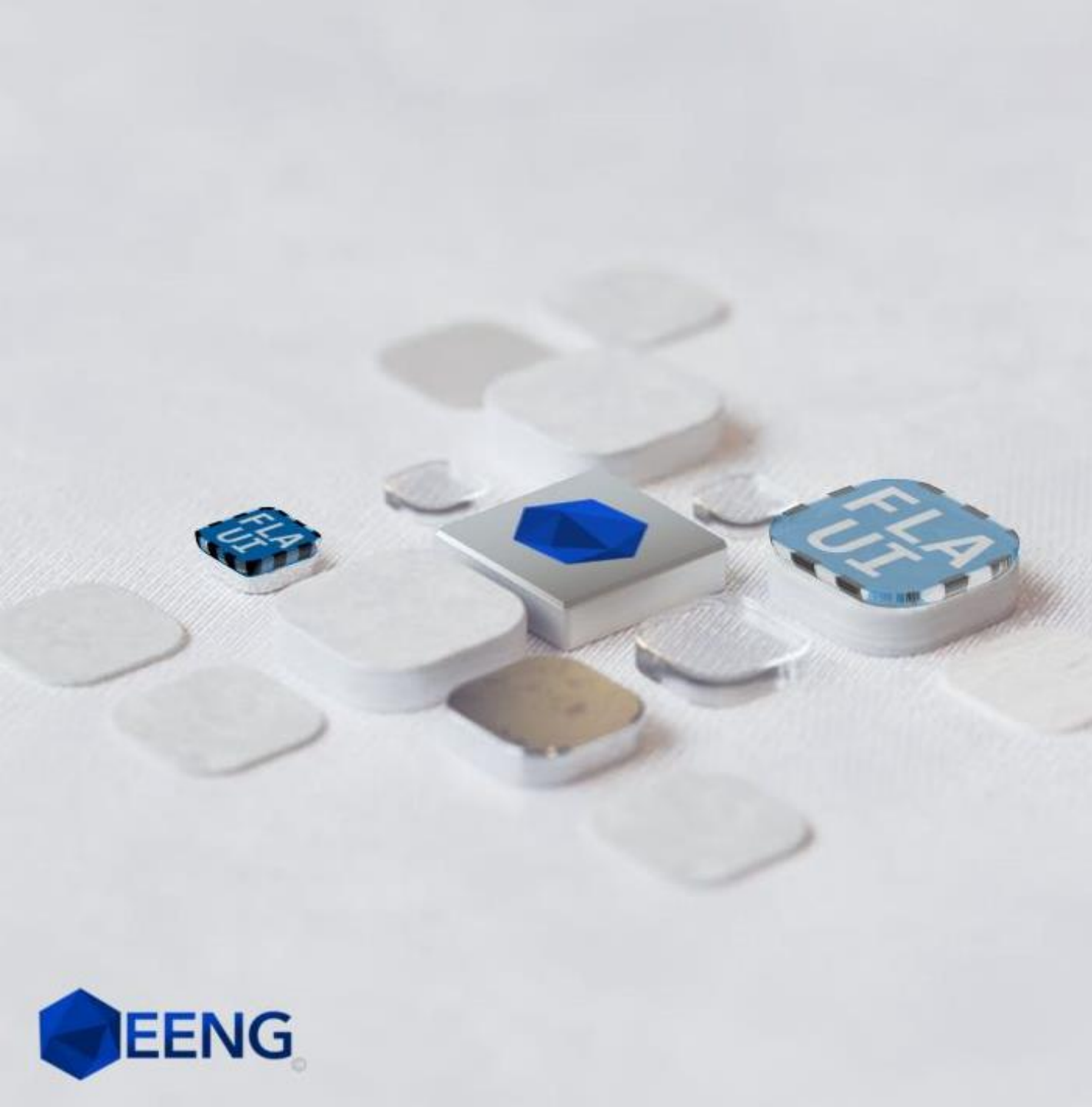
- Fokus auf Schnittstellen und Datenfluss zwischen Modulen.
- Überprüfung realer Nutzungsszenarien.

Vorteile:

- Erkennen von Fehlern an Schnittstellen.
- Sicherstellen, dass Module korrekt zusammenarbeiten.

Beispiel:

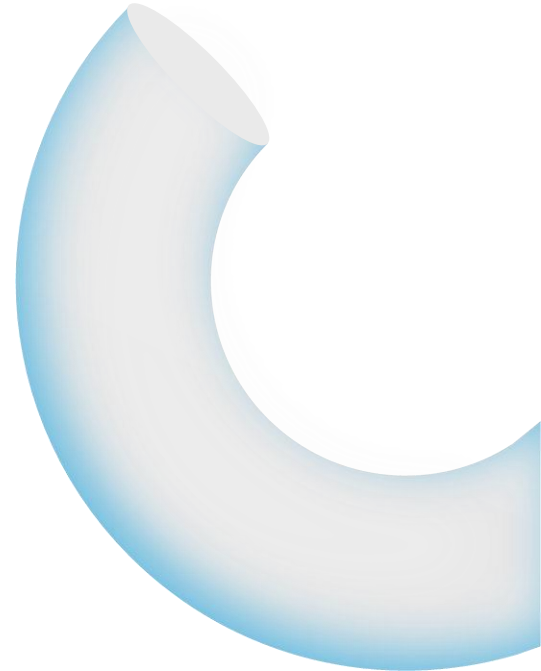
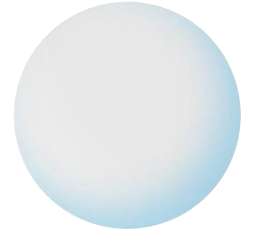
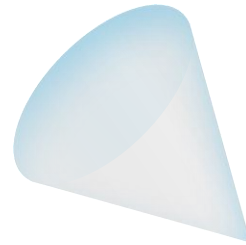
- Testen, ob eine API korrekt Daten an die Datenbank schreibt.
- Testen, ob UI-Interaktion sich nach Vorgabe verhält



Integrationstests- arten

Integrationstestarten

- Service-Level
- Database
- Dependency Injections
- UI
- End to End



Service Level

Definition:

- Service-Level-Testing überprüft, ob ein spezifischer Service (z. B. eine API) korrekt funktioniert und die erwarteten Ergebnisse liefert.

Fokus:

- Testen einzelner Services oder Microservices unabhängig von anderen Systemteilen.

Merkmale:

- Validiert die Funktionalität, Performance und Zuverlässigkeit eines Services.
- Kann reale oder simulierte Abhängigkeiten (Mocks/Stubs) nutzen.

Ziele:

- Sicherstellen, dass der Service den Anforderungen entspricht.
- Überprüfung von Eingaben, Ausgaben und Fehlerbehandlungen.

Beispiele:

- Testen eines REST-API-Endpunkts für Benutzeranmeldungen.
- Prüfen, ob ein Microservice korrekte Daten an einen anderen Service weiterleitet.

Service Level

```
[Fact]
0 references
public async Task GetRandomFact_Url_ShouldReturnSuccessAndContainFactField()
{
    // Arrange
    HttpClient _client = new HttpClient();
    string url = "https://catfact.ninja/fact";

    // Act
    var response = await _client.GetAsync(url);

    // Assert
    var content = await response.Content.ReadAsStringAsync();

    Assert.Equal(200, (int)response.StatusCode);
    Assert.Contains("fact", content);
}
```

Database

Definition:

- Datenbanktests überprüfen die Funktionalität, Integrität und Performance von Datenbankoperationen.

Fokus:

- Sicherstellen, dass Daten korrekt gespeichert, aktualisiert, abgerufen und gelöscht werden.
- Validieren der Konsistenz und Beziehungen zwischen Tabellen.

Typische Szenarien:

- CRUD-Operationen (Create, Read, Update, Delete).
- Validierung von Constraints (z. B. Primary Key, Foreign Key).
- Performance-Tests für komplexe Abfragen.

Database

```
[Fact]
0 | 0 references
public void AddUser_User_ShouldStoreUserInDatabase()
{
    // Arrange: In-Memory-Datenbank einrichten
    var options = new DbContextOptionsBuilder<TestDbContext>()
        .UseInMemoryDatabase(databaseName: "TestDatabase")
        .Options;
    var userService = new UserService(new TestDbContext(options));

    // Act
    userService.AddUser(new User { Id = 1, Name = "John Doe" });

    // Assert
    var user = userService.GetUser(1);
    Assert.NotNull(user);
    Assert.Equal("John Doe", user.Name);
}
```

Dependency Injection

Definition:

- Ein Entwurfsmuster, bei dem Abhängigkeiten einer Klasse über einen Konstruktor oder DI-Container bereitgestellt werden.

Vorteile:

- Erleichtert das Testen (Mock-Objekte, In-Memory-Datenbanken).
- Modularer und wartbarer Code.
- Einfacher Austausch von Implementierungen (z. B. In-Memory- zu SQL-Datenbank).

Dependency Injection

```
[Fact]
✓ | 0 references
public void AddUser_ShouldStoreUserInDatabaseUsingDI()
{
    // Arrange
    var serviceProvider = new ServiceCollection()
        .AddDbContext<TestDbContext>(options =>
            options.UseInMemoryDatabase("TestDatabase"))
        .AddSingleton<UserService>()
        .BuildServiceProvider();

    var userService = serviceProvider.GetRequiredService<UserService>();

    // Act
    userService.AddUser(new User { Id = 1, Name = "John Doe" });

    // Assert
    var user = userService.GetUser(1);
    Assert.NotNull(user);
    Assert.Equal("John Doe", user.Name);
}
```

UI Tests



Definition:

- UI-Testing (User Interface Testing) überprüft die Benutzeroberfläche einer Anwendung.
- Validiert die Funktionalität, Usability und Konsistenz der UI-Elemente.

Warum wichtig?

- Sicherstellen, dass Benutzeraktionen wie Klicks und Eingaben korrekt verarbeitet werden.
- Verhindern von Fehlern durch manuelle Bedienung.

Typen:

- **Manuelles Testing:** Von einem Tester durchgeführt.
- **Automatisiertes Testing:** Verwendung von Tools wie FlaUI, Ranorex oder Selenium.

UI Tests



Vorteile:

- **Zeitersparnis:** Automatisierte Tests führen repetitive Aufgaben schneller aus.
- **Konsistenz:** Eliminiert menschliche Fehler.
- **Skalierbarkeit:** Tests können auf verschiedenen Geräten und Plattformen ausgeführt werden.
- **Frühes Feedback:** Probleme werden frühzeitig erkannt.

UI Testing Tools

FlaUI:

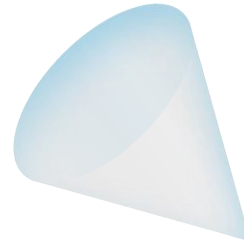
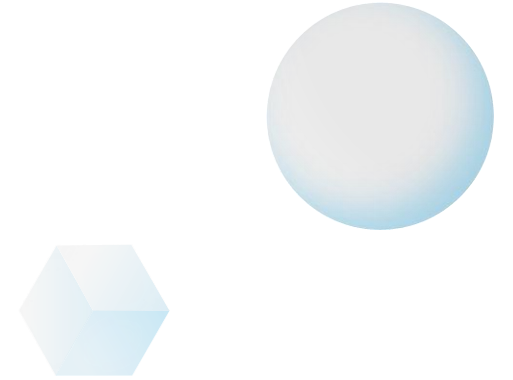
- Open Source.
- Unterstützung für Windows-Anwendungen (WPF, WinForms).

Selenium:

- Ideal für Webanwendungen.
- Plattformübergreifend.

Ranorex:

- Kommerziell, umfassende Funktionalität.
- Unterstützt Desktop-, Web- und Mobilanwendungen.



E2E

Definition:

- End-to-End-Testing überprüft den gesamten Workflow einer Anwendung von Anfang bis Ende.
- Testet alle Komponenten: UI, Backend, Datenbank, externe Services.

Ziele:

- Sicherstellen, dass die gesamte Anwendung wie erwartet funktioniert.
- Validieren realer Nutzungsszenarien (z. B. "Produkt kaufen", "Benutzer registrieren").

Beispiel:

- Benutzeranmeldung → Warenkorb → Bezahlung → Bestellbestätigung.

E2E

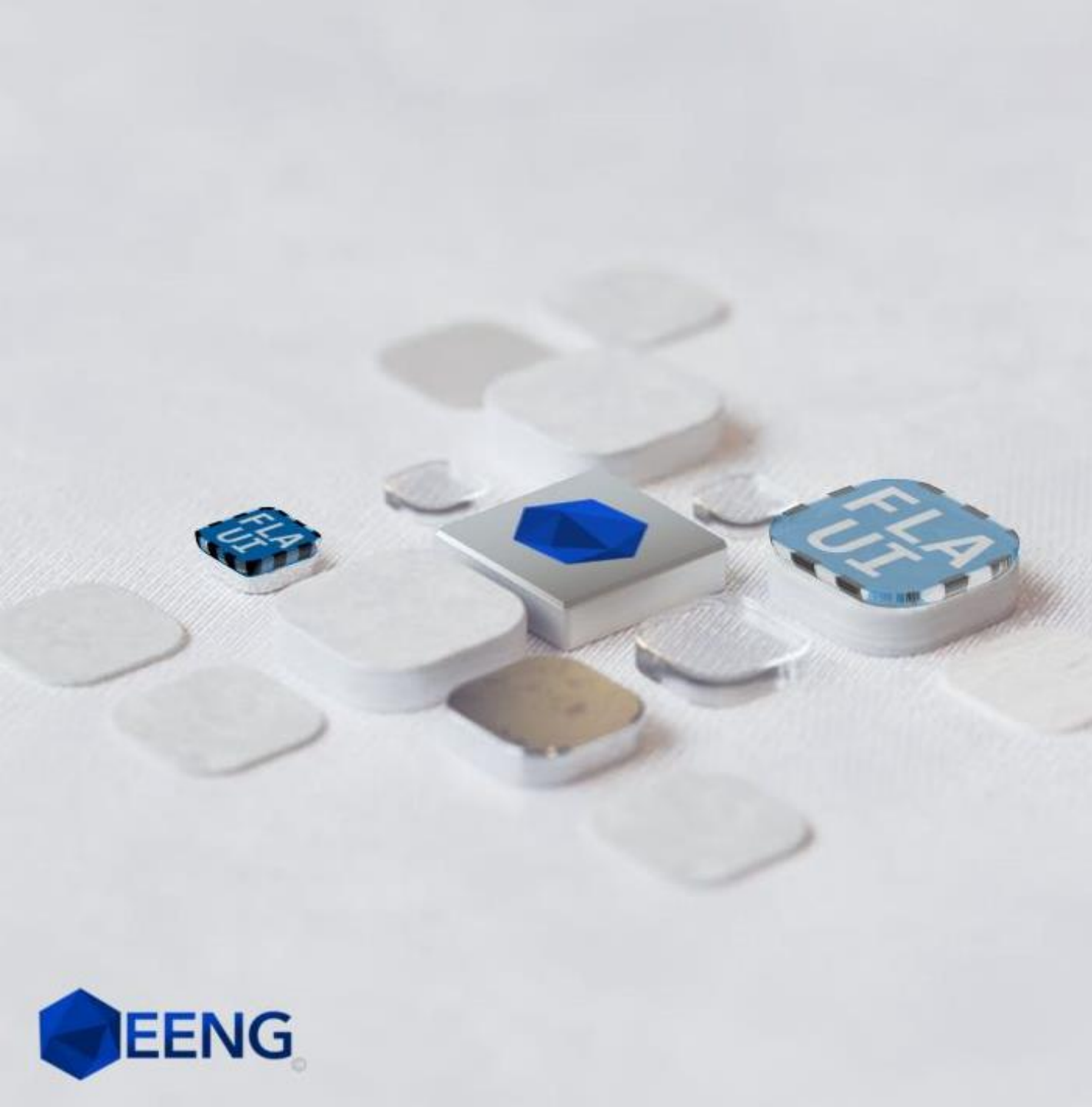


Vorteile:

- **Umfassende Abdeckung:** Testet alle Komponenten und deren Interaktionen.
- **Realitätsnah:** Validiert echte Nutzerabläufe.
- **Frühes Feedback:** Erkennt Fehler, die durch Integration verschiedener Systeme entstehen.

Herausforderungen:

- **Komplexität:** Aufbau und Wartung der Tests können aufwändig sein.
- **Zeitaufwändig:** E2E-Tests sind langsamer als Unit- oder Integrationstests.
- **Abhängigkeiten:** Erfordert eine vollständige, funktionierende Testumgebung.



Ab wann ist ein
Integrationstest
sinnvoll?

Ab wann ist ein Integrationstest sinnvoll?

Sobald Contracts definiert sind:

- Integrationstests können parallel mit TDD begonnen werden, wenn die **Verträge (Contracts)** zwischen Modulen oder Services festgelegt sind.
- Contracts definieren die Struktur und Erwartung von Daten, z. B. API-Endpoints, Schnittstellen, DTOs.

Warum funktioniert das?

- Tests validieren die Interaktion zwischen Modulen, auch wenn die interne Implementierung noch fehlt.
- Mit Mocks können Abhängigkeiten simuliert werden.

Mock

```
[Fact]
0 | 0 references
public void AddUser_ShouldStoreUserInMockedService()
{
    // Arrange
    var testUser = new User { Id = 1, Name = "John Doe" };

    var mockUserService = Substitute.For<UserService>(Substitute.For<TestDbContext>());
    mockUserService.Received(1).AddUser(Arg.Any<User>());
    mockUserService.GetUser(Arg.Any<int>()).Returns(testUser);

    // Act
    mockUserService.AddUser(testUser);

    // Assert
    var resultUser = mockUserService.GetUser(1);
    Assert.NotNull(resultUser);
    Assert.Equal("John Doe", resultUser.Name);
}
```


Ab wann ist es sinnvoll mit der UI-Automatisierung anzufangen?



Stabile Benutzeroberfläche:

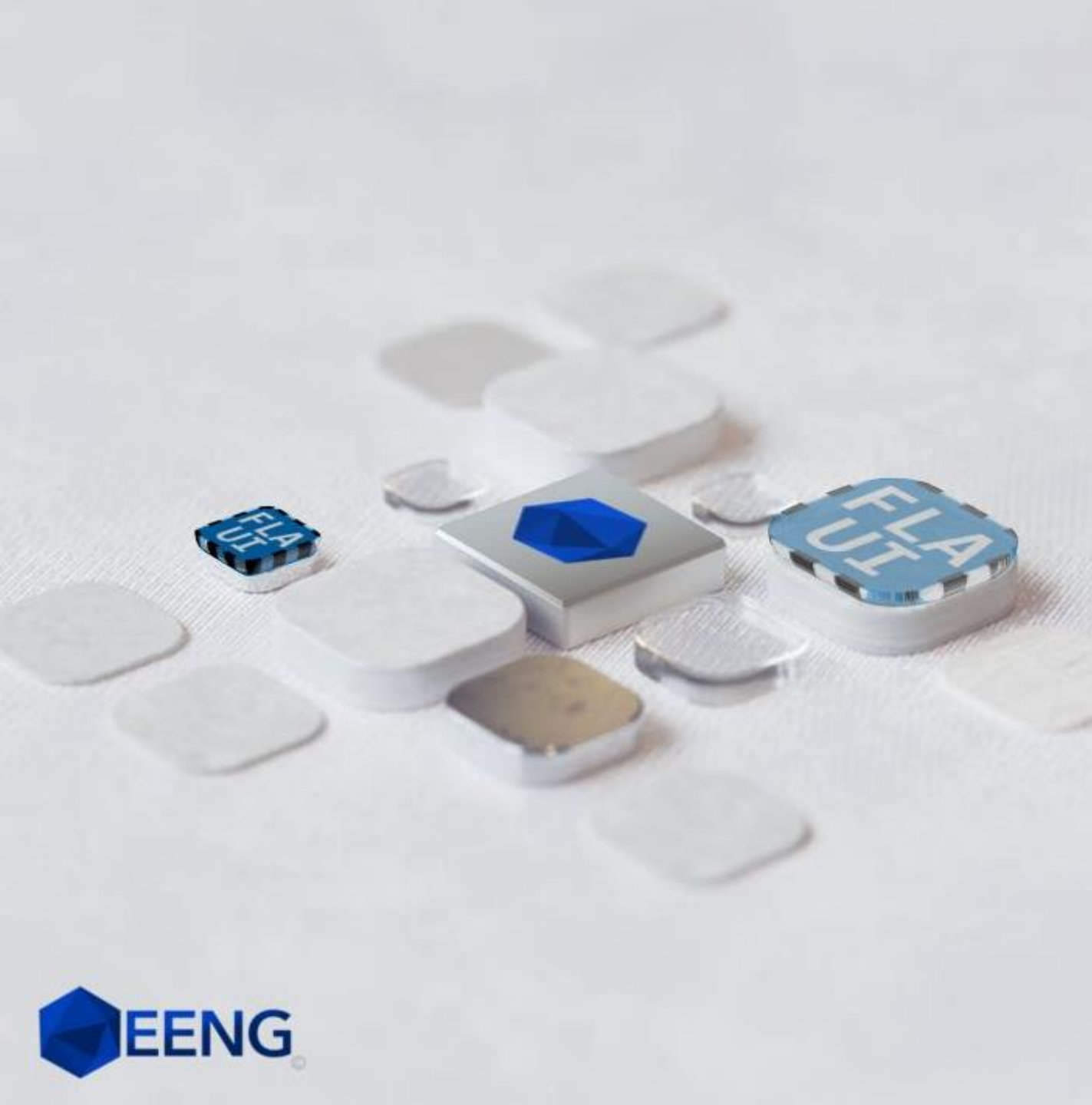
- Die Grundstruktur der UI ist etabliert und ändert sich nicht mehr häufig.
- Häufige Änderungen in der UI führen zu hohen Wartungskosten für automatisierte Tests.

Wiederholbare Szenarien:

- Funktionen, die regelmäßig manuell getestet werden müssen.
- Beispiele: Login, Formularübermittlungen, Suchfunktionen.

Nach der ersten Release-Version:

- Sobald die Anwendung produktiv geht, um Regressionen in Folge-Releases zu verhindern.



Windows Automation

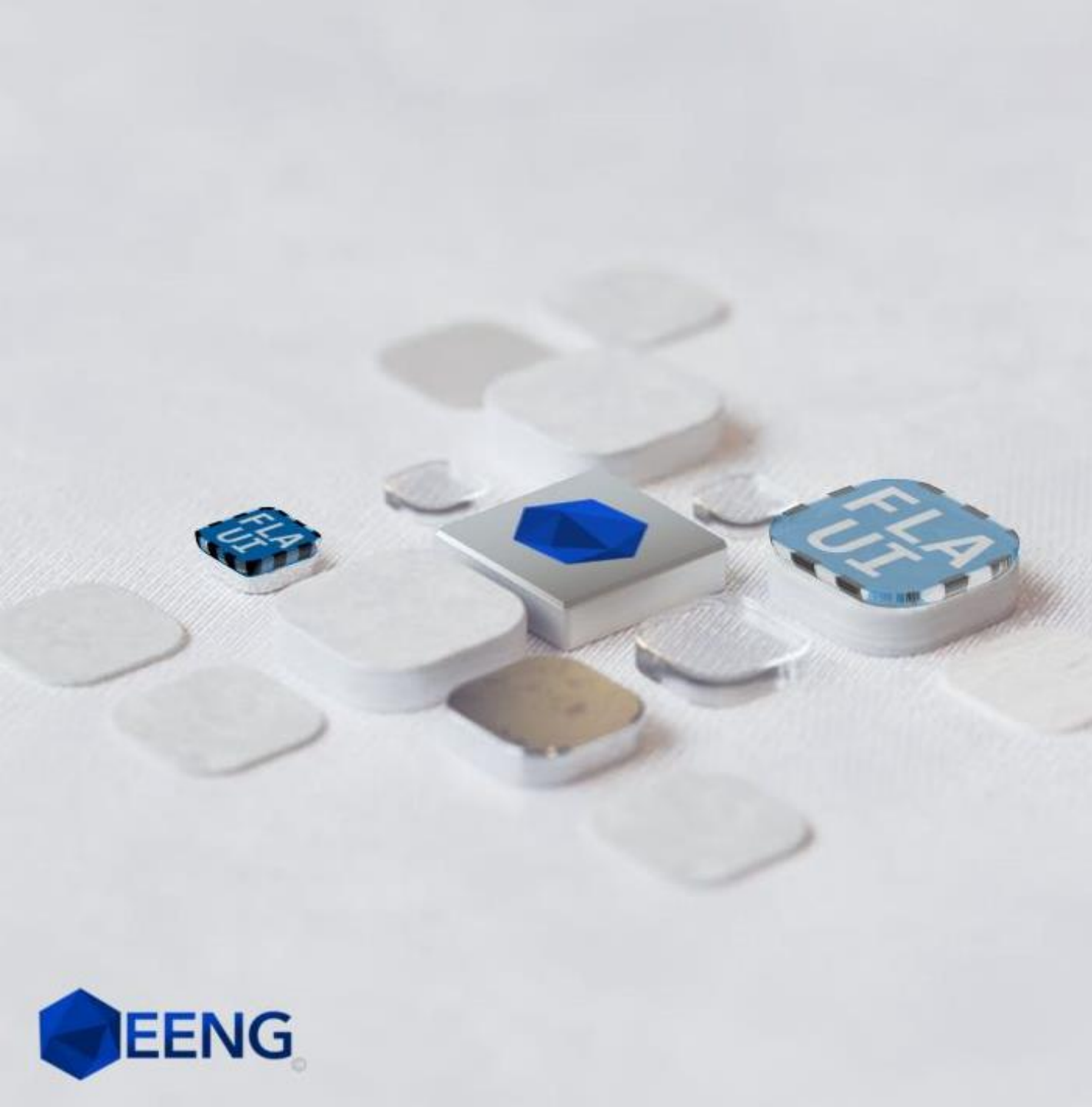
Windows Automation



- **UI-Automatisierung:** Ermöglicht das Steuern und Testen von Benutzeroberflächen, ohne direkten Zugriff auf den Quellcode zu haben.
- **Barrierefreiheit:** Bietet eine Schnittstelle, mit der Accessibility-Tools, wie Screenreader oder Sprachsteuerungen, mit der Anwendung interagieren können.
- **Testing:** Ermöglicht automatisierte Tests von UI-Komponenten und Interaktionen.

Automation mit .Net

```
var mainWindow = AutomationElement.RootElement.FindFirst(TreeScope.Children,  
    new PropertyCondition(AutomationElement.NameProperty, "MainWindow"));  
class UITestNative  
  
var usernameBox = mainWindow.FindFirst(TreeScope.Descendants,  
    new PropertyCondition(AutomationElement.AutomationIdProperty, "UsernameBox"));  
  
var passwordBox = mainWindow.FindFirst(TreeScope.Descendants,  
    new PropertyCondition(AutomationElement.AutomationIdProperty, "PasswordBox"));  
  
var loginButton = mainWindow.FindFirst(TreeScope.Descendants,  
    new PropertyCondition(AutomationElement.AutomationIdProperty, "LoginButton"));  
  
if (usernameBox.TryGetCurrentPattern(ValuePattern.Pattern, out var usernamePattern))  
{  
    ((ValuePattern)usernamePattern).SetValue("admin");  
}  
  
if (passwordBox.TryGetCurrentPattern(ValuePattern.Pattern, out var passwordPattern))  
{  
    ((ValuePattern)passwordPattern).SetValue("password");  
}  
  
if (loginButton.TryGetCurrentPattern(InvokePattern.Pattern, out var loginPattern))  
{  
    ((InvokePattern)loginPattern).Invoke();  
}
```



FLAUI

Was ist FLAUI?



Definition:

- FlaUI ist ein Open-Source-Framework für die Automatisierung von Benutzeroberflächen (UI) in Windows-Anwendungen.

Hauptmerkmale:

- Unterstützung für WPF, WinForms und andere Desktop-Anwendungen.
- Zugriff auf UI-Elemente über Accessibility APIs.
- Erlaubt automatisierte Tests von UI-Interaktionen, z. B. Klicks, Texteingaben, Navigation.

Was ist FLAUI?

- Entwickelt von **Roman Pfluger**.
- Inspiriert von White und TestStack.White, älteren UI-Automatisierungsframeworks.
- Aktive Weiterentwicklung durch die Open-Source-Community.
- Ziel: Moderne und zuverlässige UI-Automatisierung für Windows.

Was ist FLAUI?

- FlaUI arbeitet mit den **Microsoft UI Automation (UIA)** APIs.
- UIA ermöglicht Zugriff auf die zugrunde liegenden UI-Elemente von Windows-Anwendungen.
- Unterstützt verschiedene Automation Engines:
 - **UIA2:** Für ältere .NET-Framework-Anwendungen.
 - **UIA3:** Für moderne WPF/WinForms-Anwendungen.

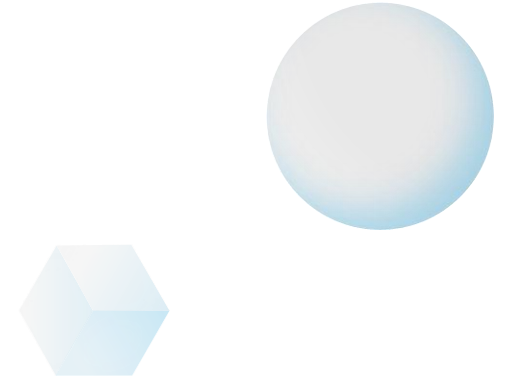
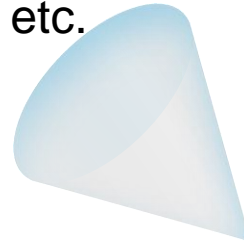
Was ist FLAUI?

Schichten:

- **FlaUI-Core:** Kernbibliothek mit grundlegender Funktionalität.
- **Automation Libraries:** UIA2/3 für Zugriff auf UI-Elemente.
- **Test Frameworks:** Integration mit xUnit, NUnit, etc.

Workflow:

- Anwendung starten.
- UI-Elemente identifizieren.
- Interaktionen durchführen.



Was ist FLAUI?

Automatisierte Tests:

- Validierung von UI-Elementen (Buttons, Listen, Texteingaben).

Regressionsprüfungen:

- Sicherstellen, dass bestehende Funktionen nach Updates weiterhin funktionieren.

End-to-End-Tests:

- Überprüfen vollständiger Workflows (z. B. Login → Dashboard).

Accessibility Testing:

- Sicherstellen, dass Anwendungen barrierefrei sind.

Was ist FLAUI?

AutomationProperties:

- Identifikation über Automation IDs, Namen, Klassen.

XPath-Unterstützung:

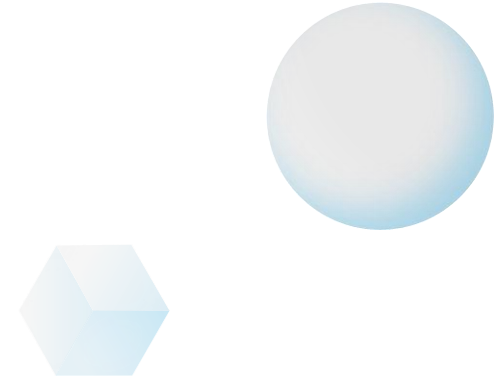
- UI-Elemente können mit XPath-Abfragen gefunden werden.

Inspect Tool:

- Externes Tool zur Analyse von UI-Elementen.

Hierarchischer Zugriff:

- Navigation durch UI-Hierarchien (z. B. Fenster → Panel → Button).



Vorteile FLAUI

- **Unabhängigkeit:** Unterstützt alle Arten von Windows-Anwendungen.
- **Einfache Integration:** Funktioniert mit Testframeworks wie xUnit, NUnit.
- **Community Support:** Open Source mit aktiver Community.
- **Zukunftssicher:** Unterstützung für moderne und ältere Technologien.

Beispiel

```
[Fact]
| 0 references
public void Login_ShouldNavigateToDashboard()
{
    // Arrange: WPF-Anwendung starten
    var process = Process.Start("MathUi.exe");

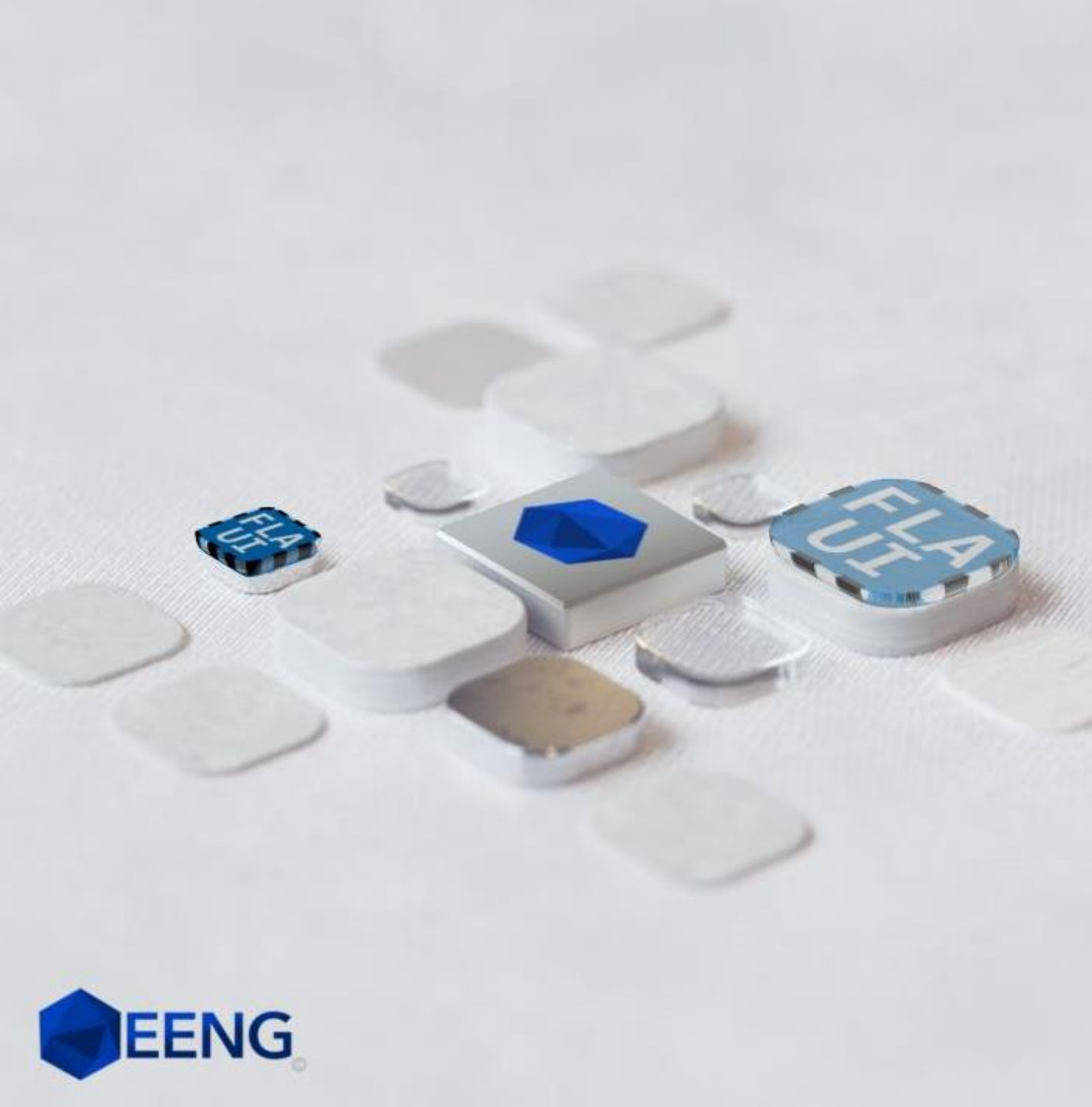
    var automation = new UIA3Automation();
    var app = FlaUI.Core.Application.Attach(process);
    var mainWindow = app.GetMainWindow(automation).WaitUntilEnabled();

    // Act
    var usernameBox = mainWindow.FindFirstDescendant(cf => cf.ByAutomationId("UsernameBox")).AsTextBox();
    var passwordBox = mainWindow.FindFirstDescendant(cf => cf.ByAutomationId("PasswordBox")).AsTextBox();
    var loginButton = mainWindow.FindFirstDescendant(cf => cf.ByAutomationId("LoginButton")).AsButton();

    usernameBox.Text = "admin";
    passwordBox.Text = "password";
    loginButton.Click();

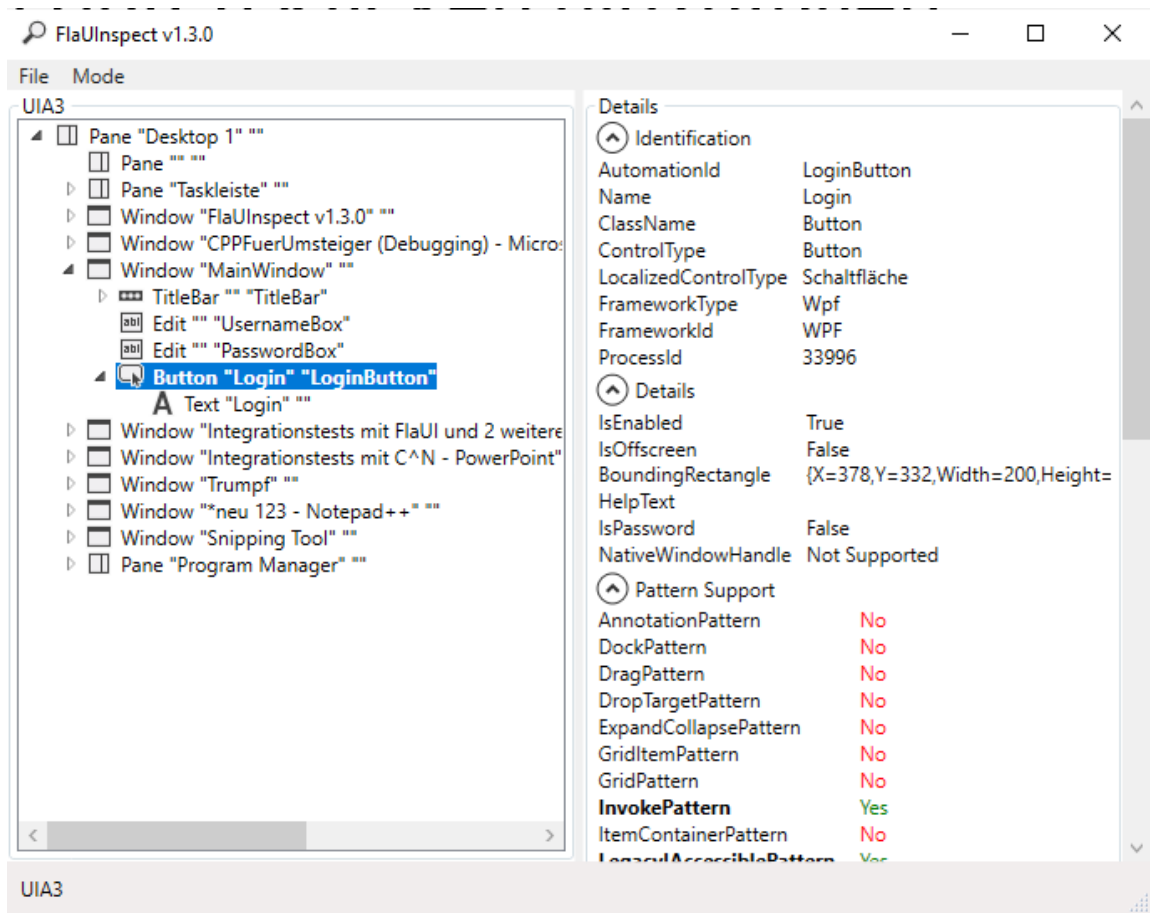
    mainWindow.FindFirstDescendant(cf => cf.ByAutomationId("DashBoard")).WaitUntilEnabled();

    // Assert
    var dashboardPanel = mainWindow.FindFirstDescendant(cf => cf.ByAutomationId("DashBoard"));
    Assert.NotNull(dashboardPanel);
    mainWindow.Close();
}
```

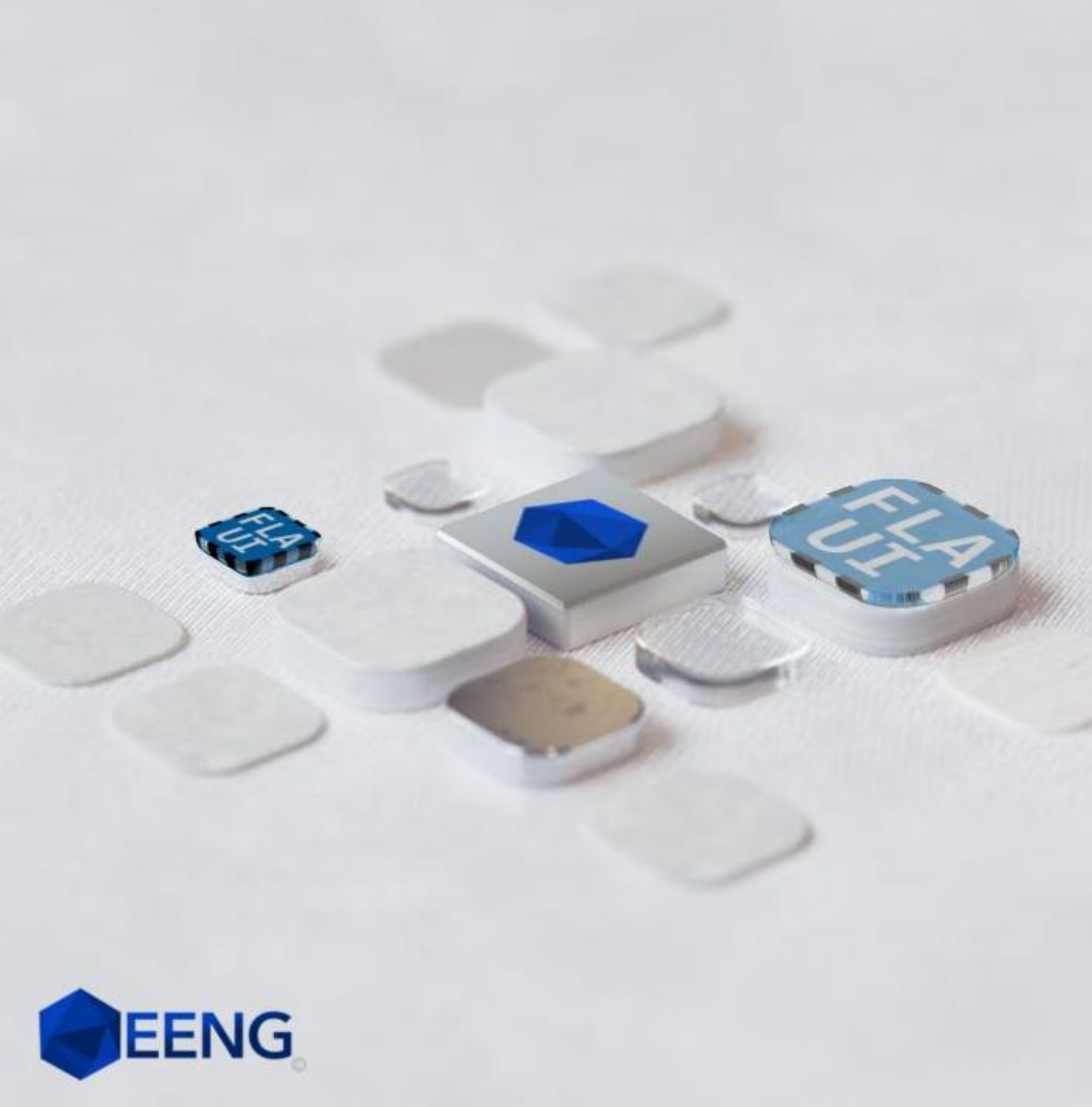


FLAUI Inspect

FLAUI Inspect



- Tree View auf alle UIA
- Hover Mode
- Focus Tracking
- Show XPath

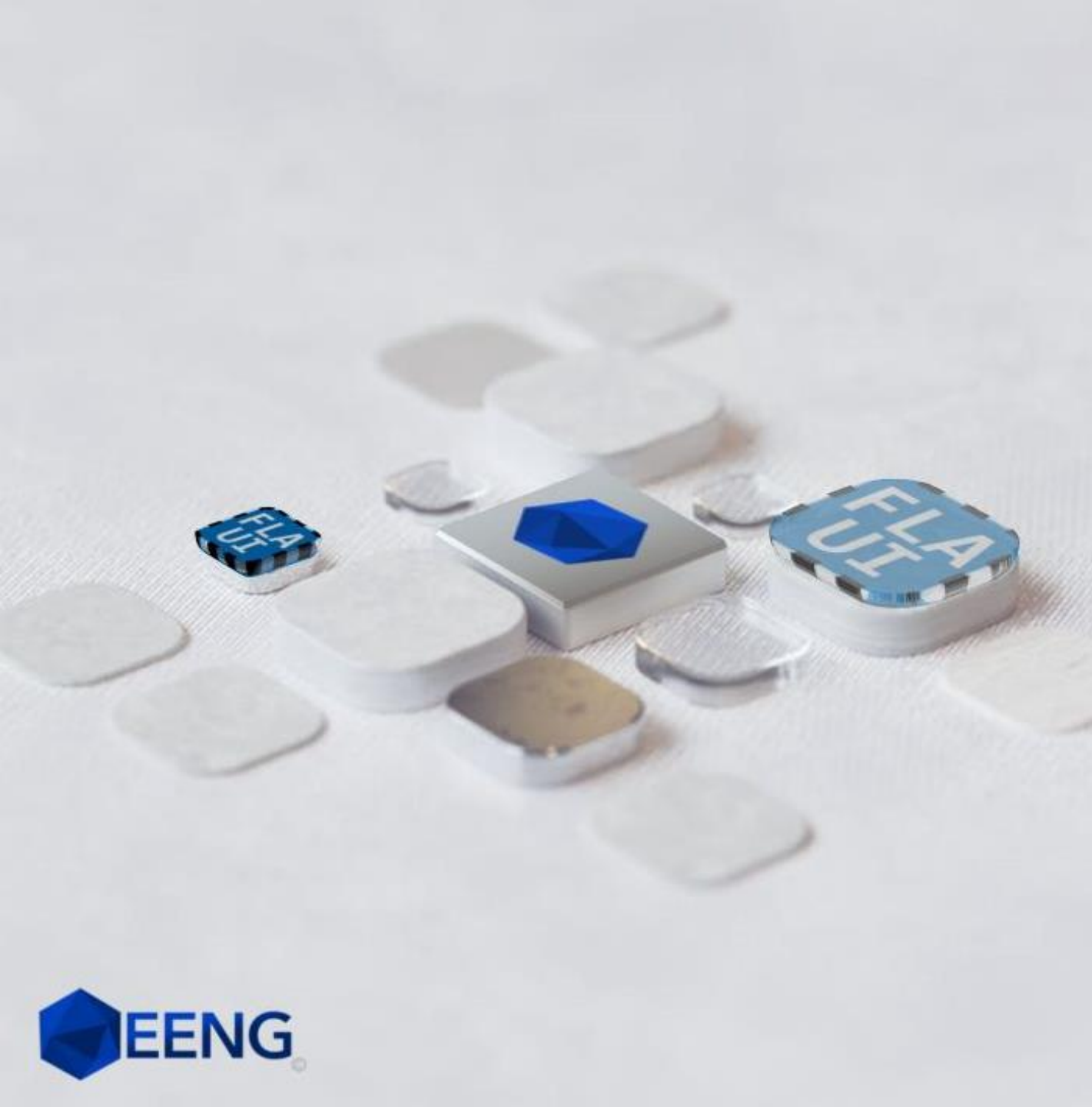


Automation mit IDs

Automation mit IDs

- sind Namen der UI Elemente
- bevorzugte Methode, da sie eindeutig ist.
- Unabhängig von der Lokalisierung (Sprache) funktioniert.

```
var loginButton = mainWindow.FindFirstDescendant(cf =>  
cf.ByAutomationId("LoginButton")).AsButton();
```

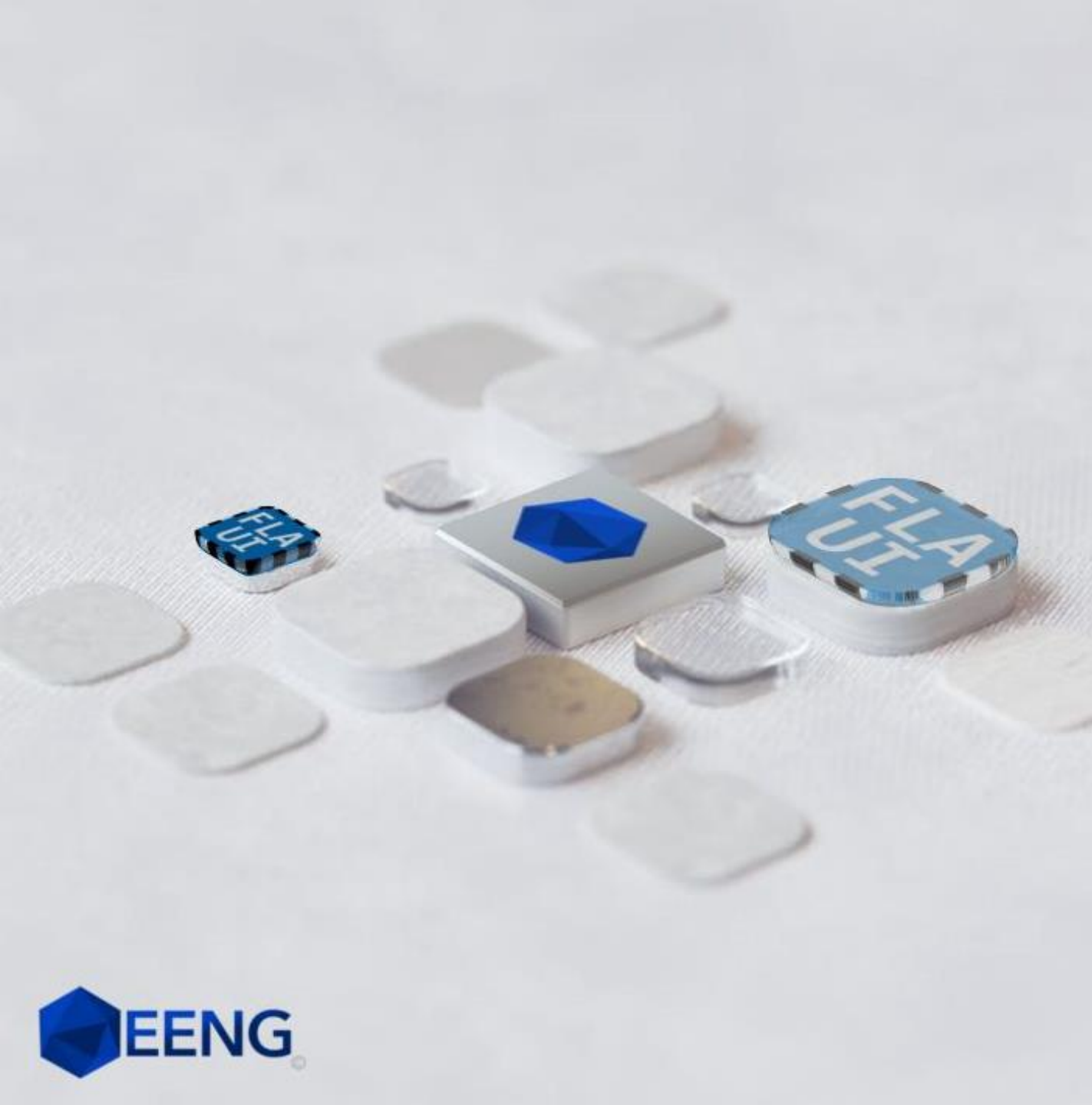


Automation mit Namen

Namen

- Der **Name** eines Elements entspricht dem sichtbaren Text in der UI.
- Nützlich, wenn keine AutomationId verfügbar ist.

```
var loginButton = mainWindow.FindFirstDescendant(cf =>  
cf.ByName("Login")).AsButton();
```

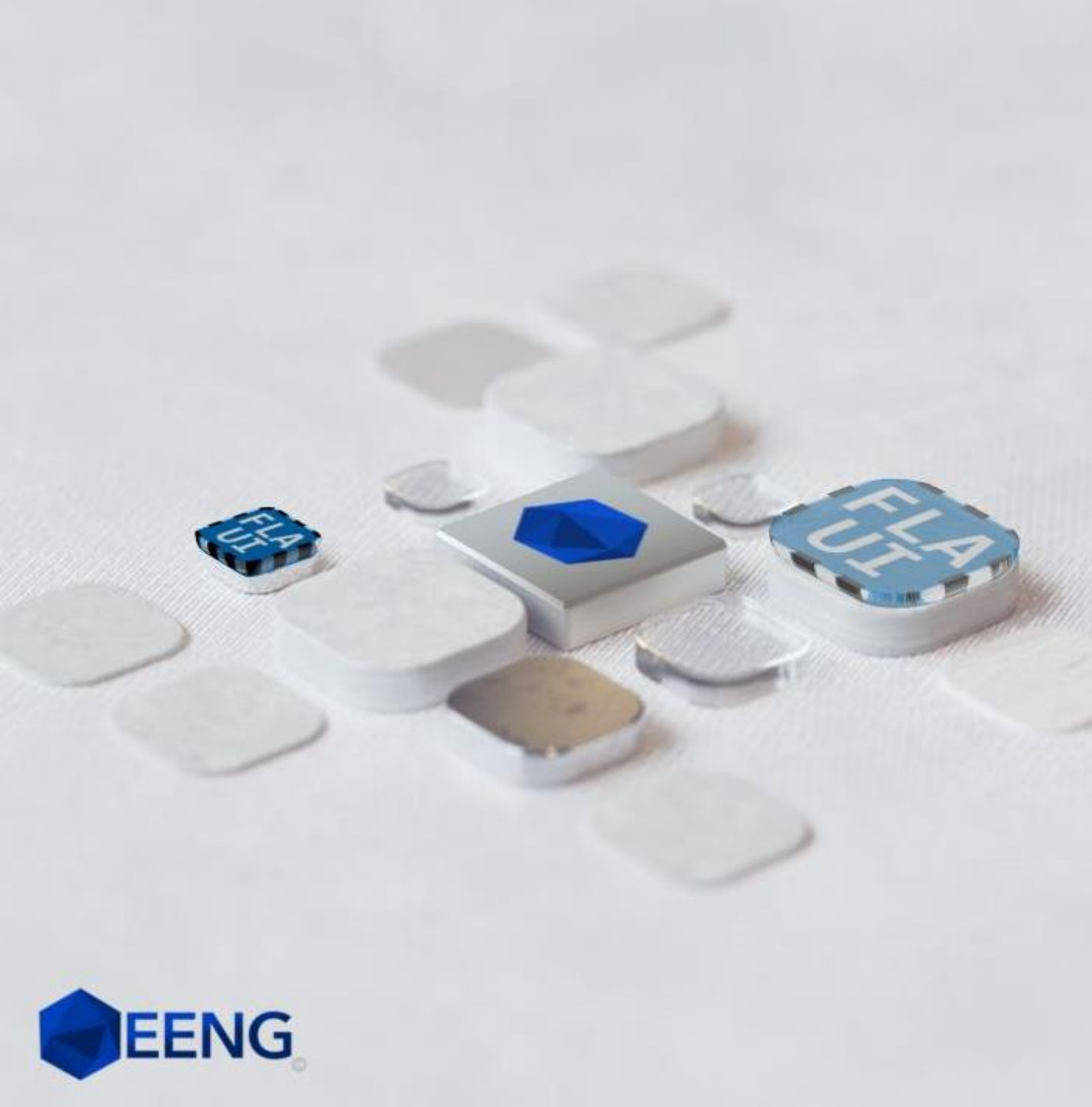


Automation mit ControlType

Control Type

- Identifikation durch den Typ des Steuerelements, z. B. Button, TextBox, ComboBox.
- Hilfreich bei generischen Tests oder wenn AutomationId und Name fehlen.

```
var firstButton = mainWindow.FindFirstDescendant(cf =>  
cf.ByControlType(FlaUI.Core.Definitions.ControlType.Button)).AsButton();
```



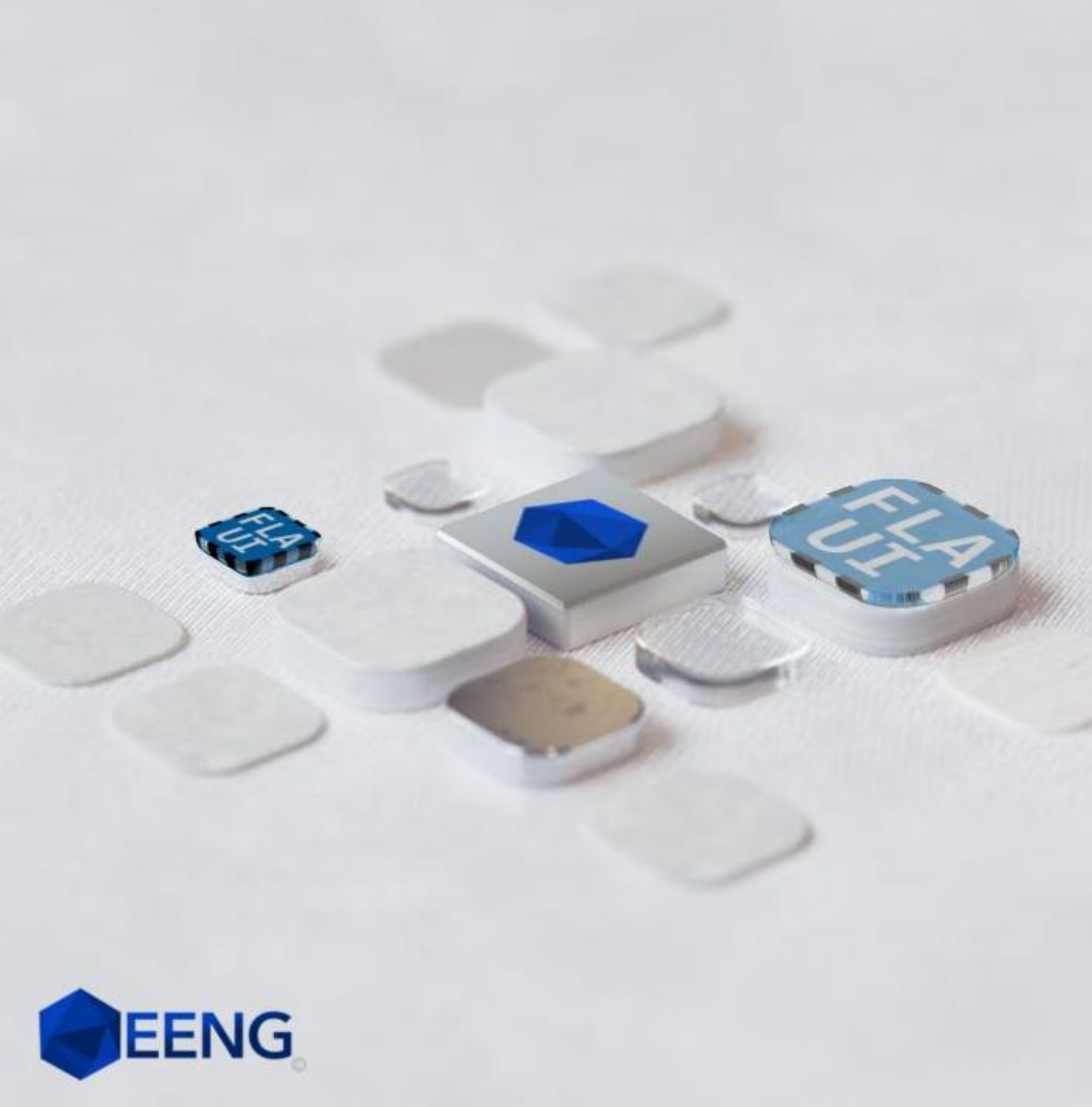
Automation mit XPath

XPath

- XPath ermöglicht hierarchische Suchen ähnlich wie in XML-Dokumenten.
- Vorteilhaft für dynamische UIs mit wiederkehrenden Strukturen.

```
var loginButton = mainWindow.FindFirstByXPath("//Button[@AutomationId='LoginButton']").AsButton();
```

```
var file = window.FindFirstByXPath($"MenuBar/MenuItem[@Name='{GetFileMenuText()}']");
```

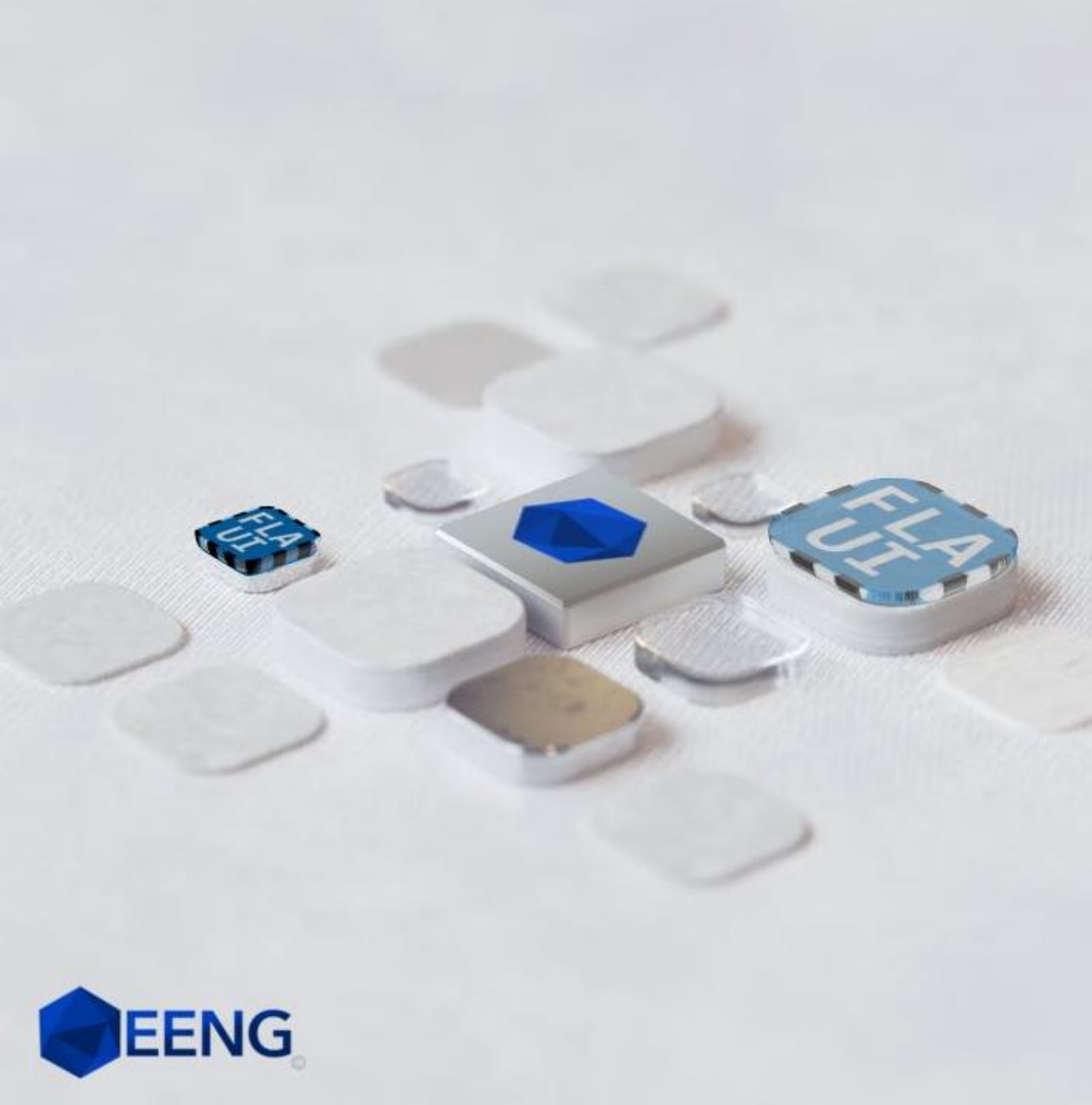
Kombination

Kombination

Die beste Praxis ist die Kombination von Zugriffsmethoden, z. B.:

- AutomationId für Eindeutigkeit.
- ControlType zur Eingrenzung.
- XPath für komplexe Hierarchien.

```
var loginButton = mainWindow .FindFirstDescendant(cf =>  
cf.ByAutomationId("LoginButton").And(cf.ByControlType(ControlType.Button)))  
.AsButton();
```



Capture

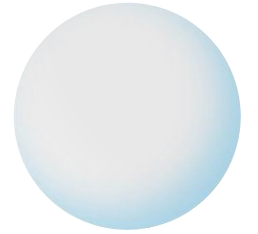
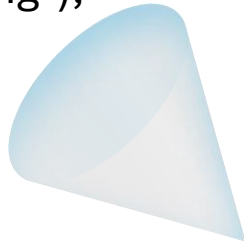
Capture

- **Capture** sind visuelle oder strukturelle Aufzeichnungen des aktuellen UI-Zustands.
- Zwei Arten von Snapshots in FlaUI:
 - **Screenshots:** Bilder der gesamten UI oder eines bestimmten Elements.
 - **UI-Dumps:** Hierarchische Strukturen der UI-Elemente in einer Datei.

Capture Images

```
var screenshot = mainWindow.Capture();  
screenshot.ToFile("MainWindow.png");
```

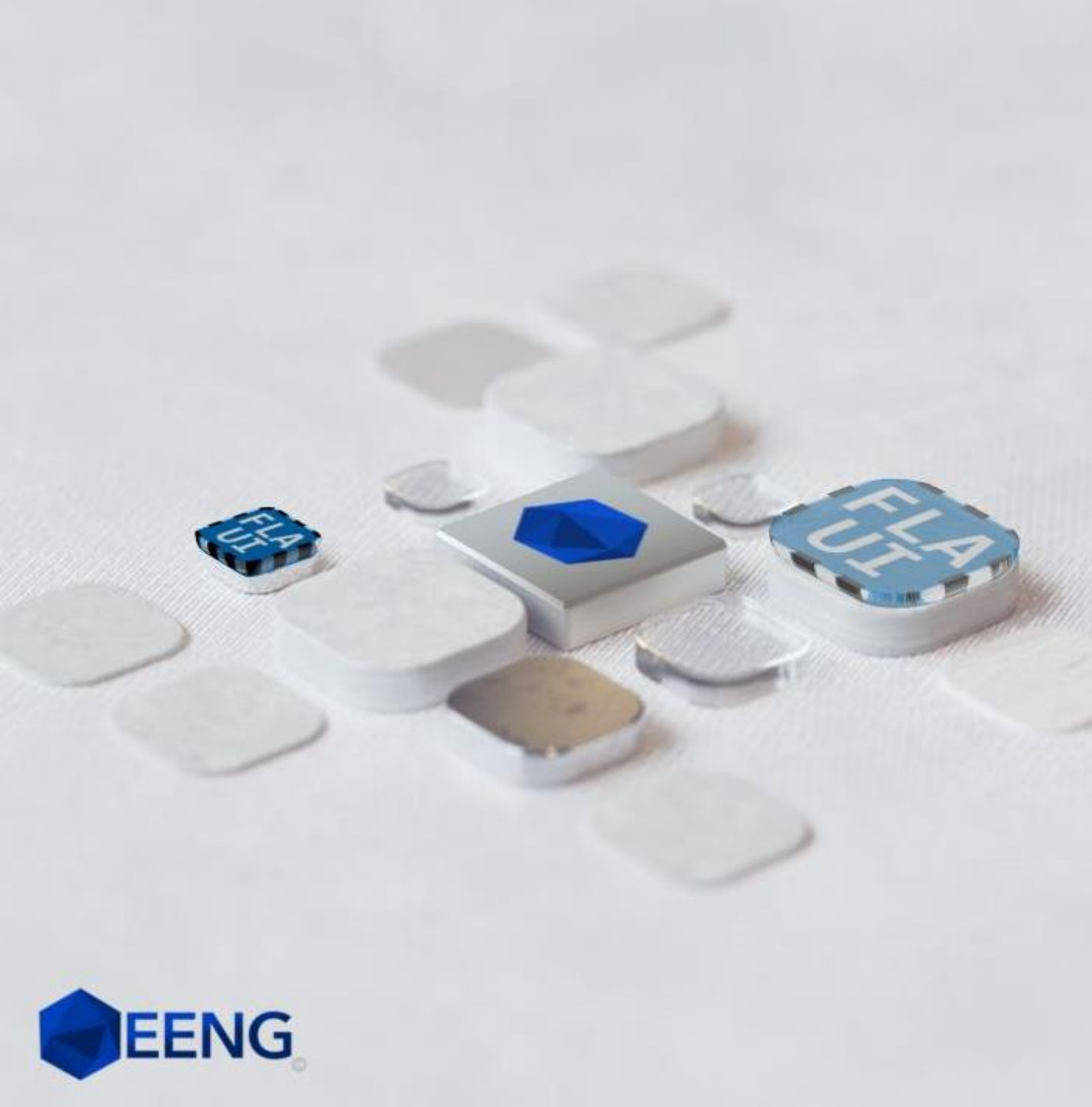
```
var buttonScreenshot = loginButton.Capture();  
buttonScreenshot.ToFile("LoginButton.png");
```



Capture Hirarchie

```
2 references | 0/1 passing
public string GetHierarchicalPath( AutomationElement element)
{
    var parent = element.Parent;
    var builder = "";
    if (parent != null)
    {
        builder+=GetHierarchicalPath(parent);
    }
    builder += "." + element.Name + "(" + element.ControlType + ")";

    return builder;
}
```



Kontrolltypen

AutomationElement

Ein AutomationElement repräsentiert ein einzelnes Benutzeroberflächenelement, das über die UI-Automatisierung (Microsoft UI Automation Framework oder FlaUI) getestet oder gesteuert werden kann.

Button

- interaktives UI-Element, das durch Benutzeraktionen (z. B. Klick) Ereignisse auslöst. FlaUI bietet umfassende Unterstützung, um Buttons in UI-Tests zu identifizieren, zu analysieren und zu steuern.
- AutomationId
 - Datentyp: string
 - Wird verwendet, um Buttons in UI-Tests eindeutig zu identifizieren.
- Name
 - Datentyp: string
 - Der sichtbare Text des Buttons, der häufig mit seiner Funktion korrespondiert (z. B. "Login", "Speichern").
- IsEnabled
 - Datentyp: bool
 - Zeigt an, ob der Button aktuell interagierbar ist (z. B. deaktivierte Buttons haben IsEnabled = false).

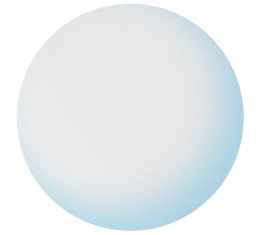
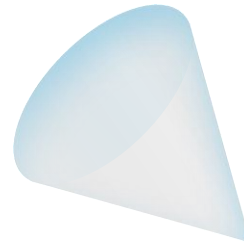
TextBox

- interaktives Element in grafischen Benutzeroberflächen, das Benutzern das Eingeben und Anzeigen von Text ermöglicht. FlaUI bietet spezielle Eigenschaften und Funktionen, um TextBoxen programmgesteuert zu testen und zu steuern.
- Text
 - Datentyp: string
 - Kann genutzt werden, um den Inhalt der TextBox zu überprüfen oder neuen Text einzufügen.
- IsReadOnly
 - Datentyp: bool
 - Zeigt an, ob die TextBox bearbeitet werden kann. Ideal für Validierungstests.

Weitere Kontrolltypen

- Button.cs
- Calendar.cs
- CheckBox.cs
- ComboBox.cs
- ComboBoxItem.cs
- DataGridView.cs
- DateTimePicker.cs
- Grid.cs
- Label.cs
- ListBox.cs
- ListBoxItem.cs
- Menu.cs
- MenuItem.cs
- MenuItems.cs

- MenuItems.cs
- ProgressBar.cs
- RadioButton.cs
- Slider.cs
- Spinner.cs
- Tab.cs
- TabItem.cs
- TextBox.cs
- Thumb.cs
- TitleBar.cs
- ToggleButton.cs
- Tree.cs
- TreeItem.cs
- Window.cs



Kontrolltypen Aufgaben

- Test Case 1
 - Login mit admin, password
 - Überprüfe ob man im Dashbord ist
 - End Process
-
- Test Case 2
 - Login mit admin, password
 - Überprüfe ob man im Dashbord ist
 - Logout
 - End Process

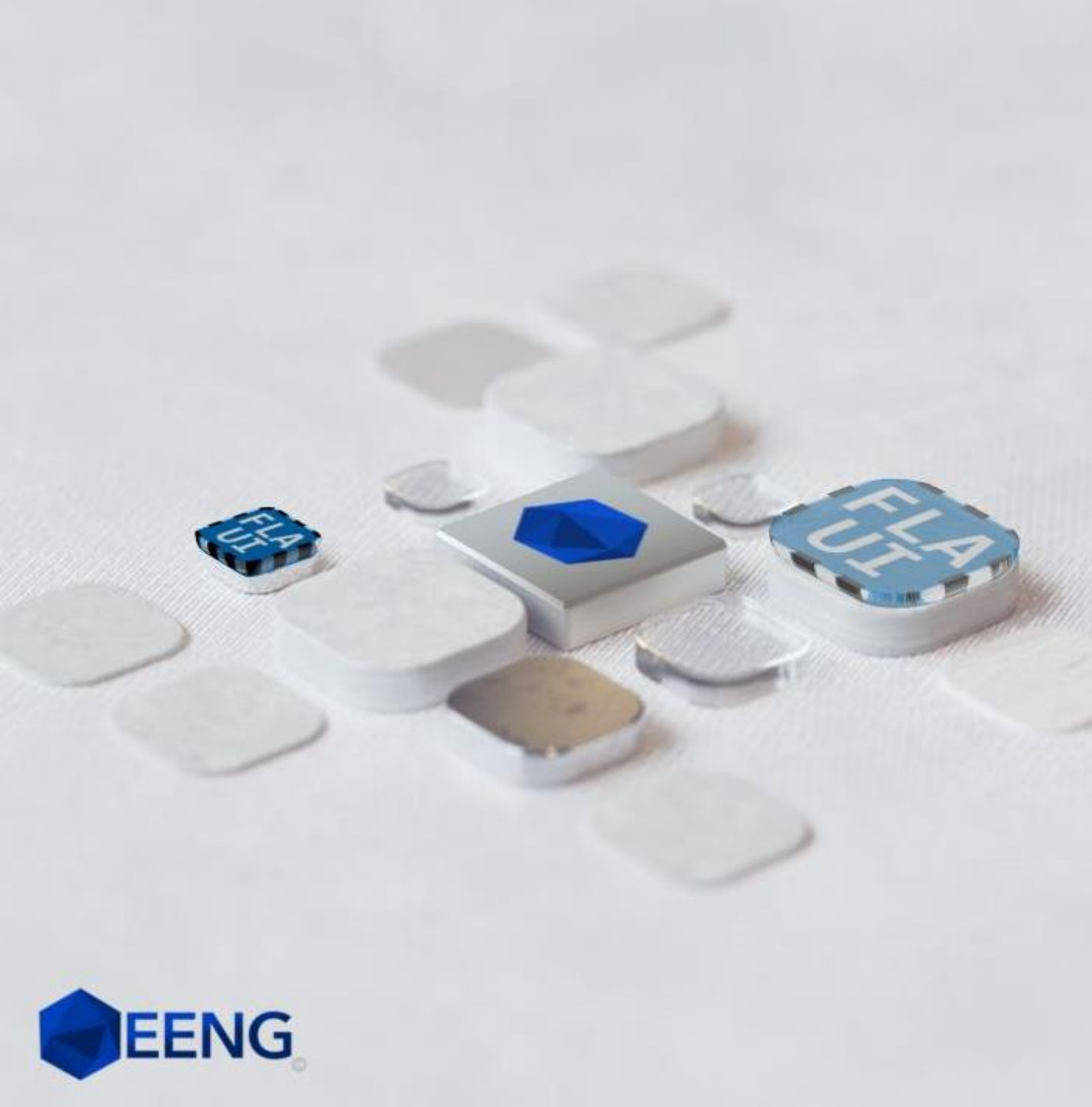
Kontrolltypen Aufgaben

- Test Case 3
- Login mit admin, password
- Element hinzufügen
- End Process

- Test Case 4
- Login mit admin, password
- Element hinzufügen
- Element löschen
- Logout
- End Process

Kontrolltypen Aufgaben

- Test Case 5
- Login mit falschem Daten
- Abfangen der Messagebox



Flaui Extensions

Extensions

- Extension-Methoden sind statische Methoden, die es ermöglichen, bestehende Klassen oder Schnittstellen mit zusätzlichen Funktionen zu erweitern, ohne sie direkt zu ändern.
- Sie werden mit dem Schlüsselwort `this` im ersten Parameter definiert.

Extensions

Push, Release, Hover:

1 reference | 1/1 passing

```
public static void Push(this Button button)
{
    Mouse.Position = button.BoundingBox.Center();
    Mouse.Down(MouseButton.Left);
}
```

1 reference | 1/1 passing

```
public static void Release(this Button button)
    => Mouse.Up(MouseButton.Left);
```

1 reference | 1/1 passing

```
public static void Hover(this Button button)
    => Mouse.MoveTo(button.BoundingBox.Center());
```

```
+ color | "{Name = ff707070, ARGB = (255, 112, 112, 112)}"
+ HoverColor | "{Name = ff3c7fb1, ARGB = (255, 60, 127, 177)}"
+ pushedColor | "{Name = ff2c628b, ARGB = (255, 44, 98, 139)}"
ick():
```

```
//loginButton.Click();
var color = loginButton.GetBackgroundColor();
loginButton.Hover();
var HoverColor = loginButton.GetBackgroundColor();
loginButton.Push();
var pushedColor = loginButton.GetBackgroundColor();
loginButton.Release();
```

Extensions

Hintergrundfarbe bekommen:

```
public static Color GetBackgroundColor(this Button button)
{
    var img = button.Capture();
    var colors = new List<Color>();

    for (int x = 0; x < img.Width; x++)
        for (int y = 0; y < img.Height; y++)
            colors.Add(img.GetPixel(x, y));

    var sortedColors = colors
        .GroupBy(c => c)
        .OrderByDescending(g => g.Count())
        .Select(g => g.Key)
        .ToList();

    var medianColor = sortedColors[1];


    return medianColor;
}
```

Button Extensions

Button Pressed Verification

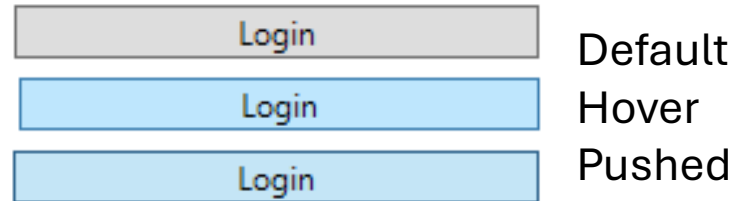
```
public static bool IsPressed(this Button button, Color pressedColor)  
    => button.GetBackgroundColor() == pressedColor;
```

```
bool isPressed = loginButton.IsPressed(Color.FromArgb(255, 44, 98, 139));  
loginButton.Release();
```

 isPressed | true

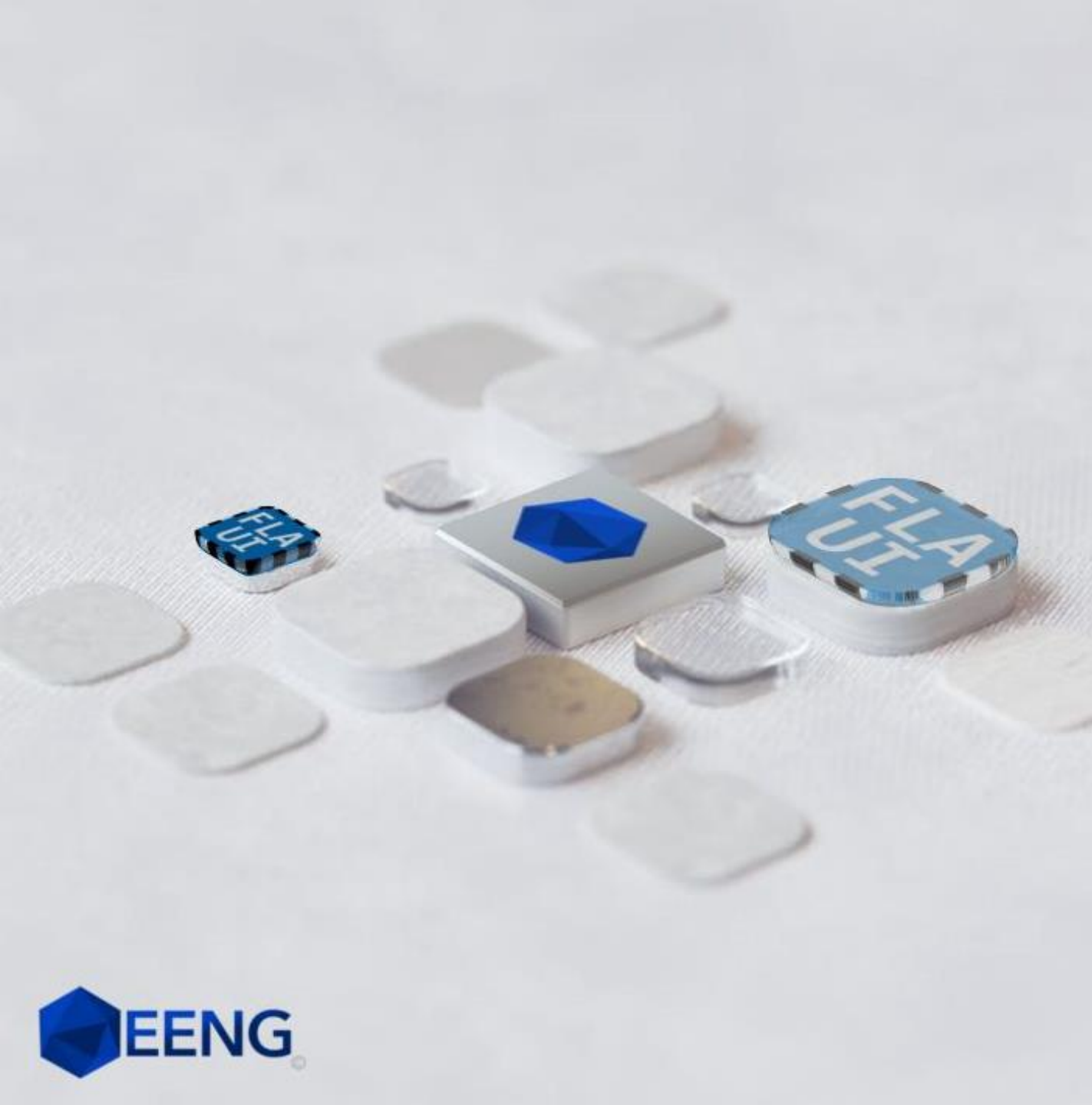
Extensions

- Weitere Möglichkeit um zu erkennen dass ein Button optisch gedrückt ist wäre eine Capture vergleich



```
1 reference | 1/1 passing
public bool ClickWithFeedback()
{
    Mouse.Position = Element.BoundingBox.Center();
    Mouse.Down(MouseButton.Left);
    Thread.Sleep(100);
    var currentButton = Element.Capture();
    var result = CompareBitmaps(currentButton, _buttonPressedBitmap);
    Mouse.Up(MouseButton.Left);

    var threshold = 6;
    return result <= threshold;
}
```



Kontrolltypen Accessoren

Button Accessor

Was sind Accessoren?

- Accessoren sind Wrapper-Klassen, die spezifische Funktionalitäten für UI-Elemente bereitstellen.
- Sie kapseln die Logik für den Zugriff und die Interaktion mit einem bestimmten Control, wie z. B. einem Button.

Vorteile:

- Erhöhung der Wiederverwendbarkeit und Modularität.
- Reduktion von Code-Duplikation in Tests.
- Vereinfachung des Zugriffs auf UI-Elemente durch benutzerdefinierte Methoden.

Button Accessor

```
private readonly Button _button;
```

```
private readonly Color _pushedColor = Color.FromArgb(255, 44, 98, 139);
```

3 references

```
public ButtonAccessor(AutomationElement automationElement, string automationID, int timeout)
```

```
{
    _button = automationElement.FindFirstDescendant(cf
        => cf.ByAutomationId(automationID)).WaitUntilEnabled(new TimeSpan(0,0,0,0,timeout)).AsButton();
    if (_button == null)
    {
        throw new Exception($"Button with AutomationID '{automationID}' not found within timeout of {timeout}ms.");
    }
}
```

2 references | 0/1 passing

```
public void Click()
```

```
{
    if (_button == null)
    {
        throw new InvalidOperationException("Button not initialized.");
    }
    _button.Push();

    if(!_button.IsPressed(_pushedColor))
    {
        throw new InvalidOperationException("Button was not pressed.");
    }
    _button.Release();
}
```

Button Accessor

```
var loginButton = new ButtonAccessor(mainWindow, "LoginButton", 100);
```

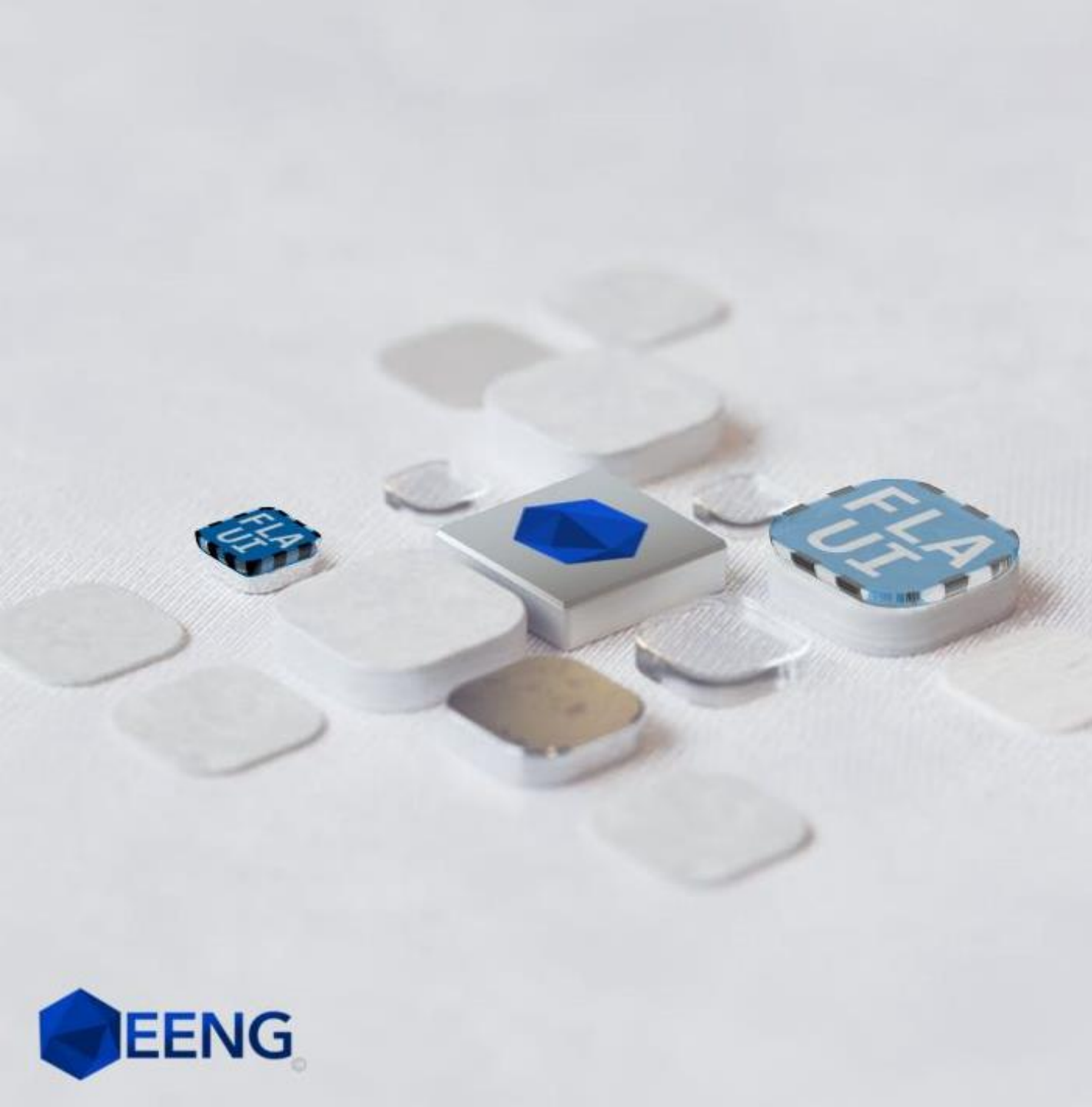

Accessor Mittels Bildverarbeitung

- Falls Keine Automation ID, Name nicht verfügbar ist, gibt es die Möglichkeit mittels Bildverarbeitung Elemente zu erkennen

```
Mat screenshotMat = BitmapToMat( mainImage);
Mat buttonMat = BitmapToMat(image);

// Template Matching durchführen
Mat result = new Mat();
CvInvoke.MatchTemplate(
    screenshotMat, buttonMat, result, TemplateMatchingType.CcoeffNormed);

// Variablen für MinMaxLoc
double minVal = 0, maxVal = 0;
System.Drawing.Point minLoc = new System.Drawing.Point();
System.Drawing.Point maxLoc = new System.Drawing.Point();
// Bestes Match finden
CvInvoke.MinMaxLoc(result, ref minVal, ref maxVal, ref minLoc, ref maxLoc);
if (maxVal > 0.8) // Schwellenwert für den Match
{
    // Maus bewegen und klicken
    Mouse.MoveTo(
        maxLoc.X + buttonMat.Width / 2,
        maxLoc.Y + buttonMat.Height / 2);
    Mouse.Click();
}
```



Erweiterungen Accessoren

Accessoren

- In komplexen UIs bestehen Gruppen von Elementen, wie Popups, Dialogfenster oder Seiten, aus mehreren Einzelkomponenten (z. B. Buttons, Textboxen).
- Accessoren ermöglichen eine logische Organisation dieser Elemente in einer strukturierten Hierarchie.

Vorteile:

- Übersichtlichkeit: UI-Elemente werden modular und wiederverwendbar.
- Klarheit: Zugriffe auf Unterelemente (z. B. Buttons in einem Popup) sind zentralisiert.
- Skalierbarkeit: Neue Elemente oder Funktionen können einfach hinzugefügt werden.

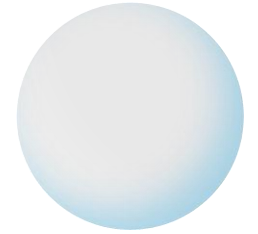
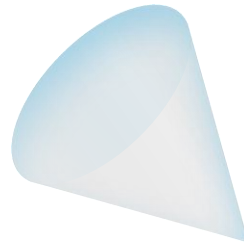
SaveConfirmDialog Accessor

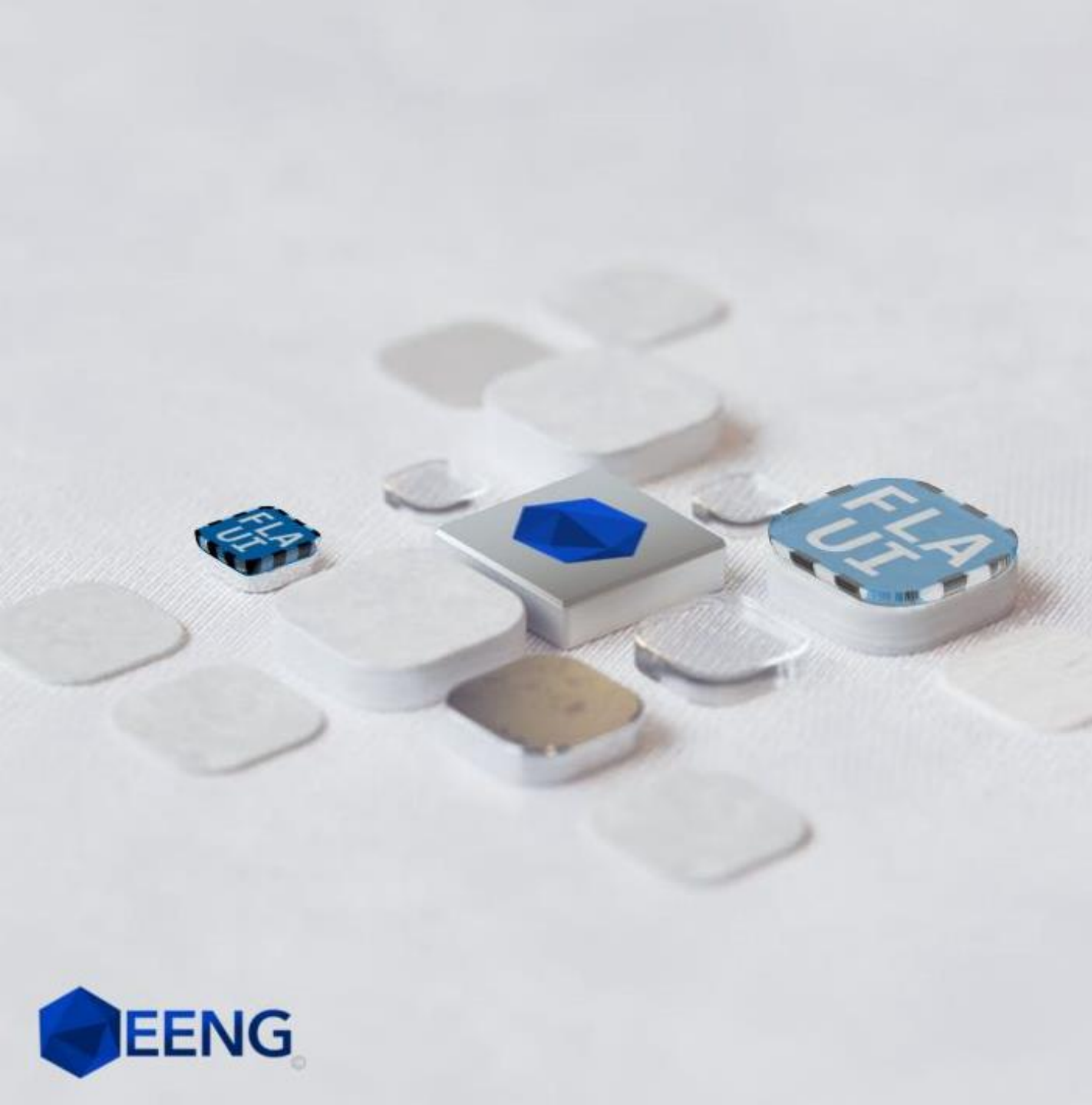
- ConfirmSaveDialog Accessor findet in der Regel eine mehrfache Verwendung in einer Applikation und könnte als eigenständiger Accessor definiert werden können

```
1 reference
internal class SavePopUpAccessor(Window mainWindow, int timeout)
    : PopupAccessor(mainWindow, "#32770", timeout)
{
    0 references
    public ButtonAccessor SaveButton
        => new ButtonAccessor(this.Popup, "CommandButton_6", 5000);
    1 reference | 0/1 passing
    public ButtonAccessor NotSaveButton
        => new ButtonAccessor(this.Popup, "CommandButton_7", 5000);
}
```

Accessor Übung

- Ergänze die Tests mit Accessoren
 - Login Accessor
 - Dashboard Accessor
 - Button Accessor
 - TextBoxAccessor
 - Listview Accessor





AccessorBase

AccessorBase

- In Testautomatisierungsprojekten wiederholen sich viele Schritte wie das Finden von Elementen, Klicken, oder die Interaktion mit Steuerelementen.
- Ohne eine zentrale Logik entstehen redundanter Code und Wartungsaufwand.
- **Lösung durch AccessorBase:**
 - **Zweck:** AccessorBase dient als Basisklasse, um wiederkehrende Logik zu kapseln und die Arbeit mit FlaUI zu vereinfachen.
 - **Funktionen:**
 - Standardisierte Methoden zur Suche und Interaktion mit UI-Elementen (z. B. durch AutomationId, Name, ControlType oder Bildvergleich).
 - Implementierung allgemeiner Aktionen wie Klicken oder Positionierung.
 - Aufnahme von Screenshots nach Aktionen zur Debugging-Unterstützung.

AccessorBase

Vorteile:

- **Wiederverwendbarkeit:** Jeder Accessor (z. B. ButtonAccessor) kann von der Basisklasse erben.
- **Konsistenz:** Standardisierte Interaktion mit der Oberfläche durch klar definierte Schnittstellen.
- **Erweiterbarkeit:** Neue Steuerelementtypen können einfach hinzugefügt werden, indem sie AccessorBase erweitern.

AccessorBase

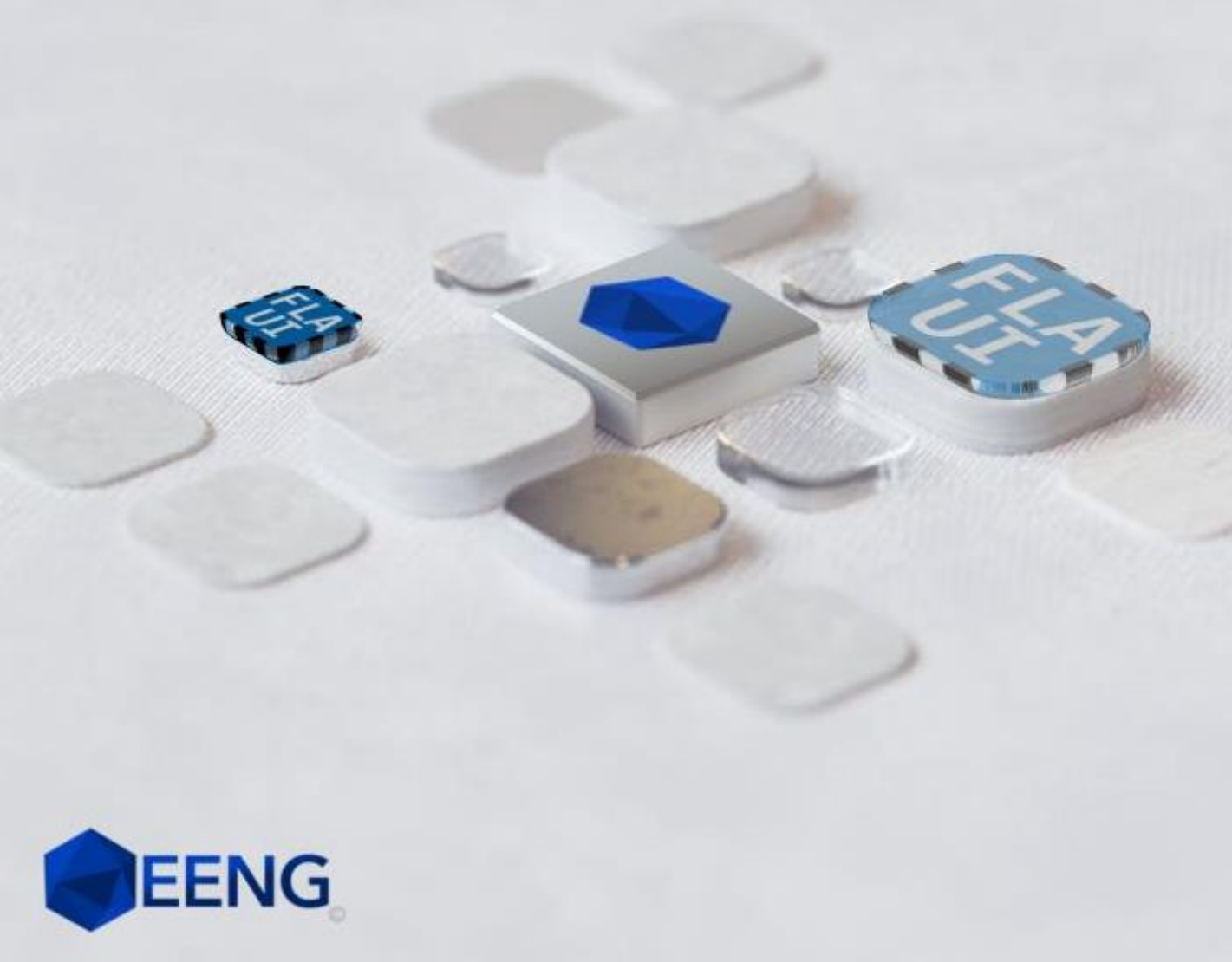
```
internal class ButtonAccessor:AccessorBase
{
    1 reference | ✔ 1/1 passing
    public ButtonAccessor(AutomationElement parent, string automationId) : base(parent, automationId)
    {
    }

    1 reference | ✔ 1/1 passing
    public ButtonAccessor(AutomationElement parent, string name, ControlType type) : base(parent, name, type)
    {
    }

    1 reference | ✔ 1/1 passing
    public ButtonAccessor(AutomationElement parent, Bitmap element) : base(parent, element)
    {
    }

    1 reference | ✔ 1/1 passing
    public void Click()
    {
        base.Click();
    }
}
```

Vielen Dank für Ihre Aufmerksamkeit!



B. Eng. Caglar Özdemir
Mobil: 0163 34050 13
info@oeeng.de
www.oeeng.de