EPFL

CS-311: Advanced Testing

Prof. George Candea
School of Computer & Communication Sciences

Outline

- Mock testing and dependency injection
- Regression testing
- UI testing

Mock Testing and Dependency Injection

MainActivity

```
class MainActivity : AppCompatActivity() {

    private val authService = CloudAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)  //  ... our UI layout

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

External cloud service used for authentication

UI has two text fields for entering a username and a password

Helper function for logging in

Login outcome displayed as text

- How do we test MainActivity ?

Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

MainActivity

External cloud service used for authentication

UI has two text fields for entering a username and a password

Helper function for logging in

Login outcome displayed as text

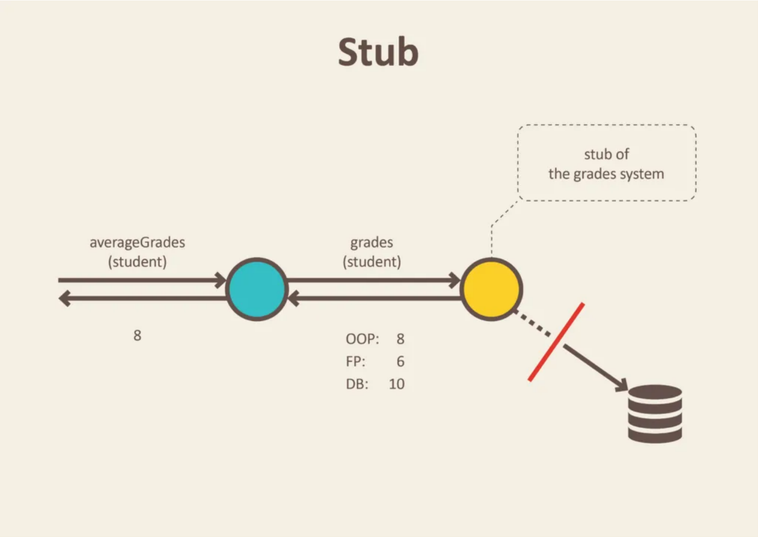
```
class MainActivity : AppCompatActivity() {

    private val authService = CloudAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)  //  ... our UI layout

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

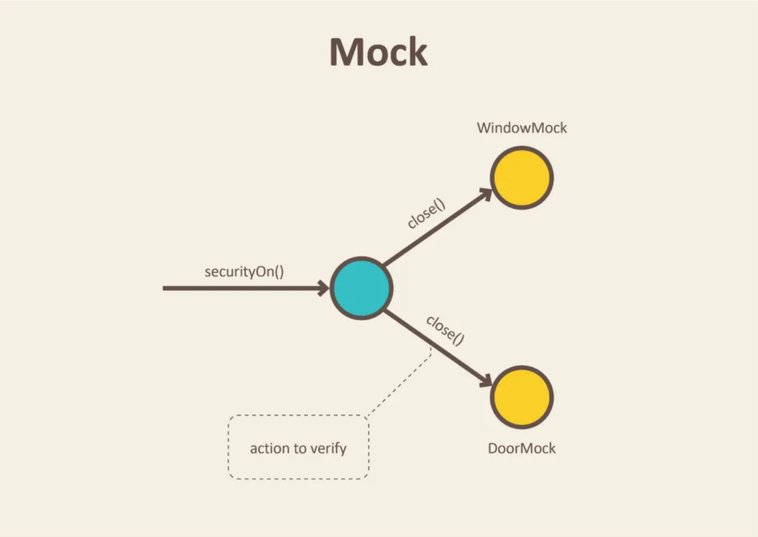
Mock authentication service

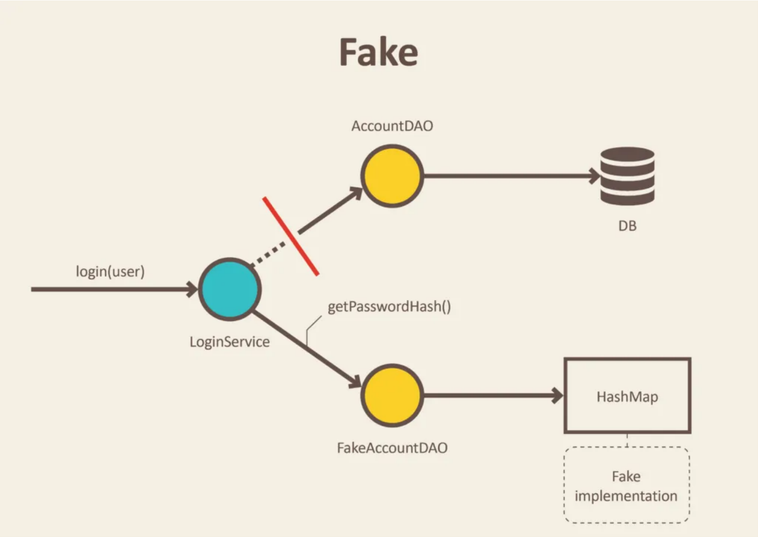
```
class MockAuthService {
    fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        // Directly invoke the callback with predefined results suitable for testing
        // Here we always return true for successful authentication in tests
        callback(true)
    }
}
```

- Mock vs. Stub vs. Fake ?

Mock vs. Stub vs. Fake








```
class MockAuthService {
    private var lastUsername: String? = null
    private var lastPassword: String? = null

    fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        lastUsername = username
        lastPassword = password

       callback(true)
    }

    fun getLastUsername(): String? = lastUsername
    fun getLastPassword(): String? = lastPassword
}
```

https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

MainActivity

```
class MainActivity : AppCompatActivity() {

    private val authService = CloudAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)  //  ... our UI layout

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

MainActivity using the mock auth service

```
class MainActivity : AppCompatActivity() {

    // Use MockAuthService instead of real service (for testing)
    private val authService = MockAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```


Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

MainActivity using the mock auth service

```
class MainActivity : AppCompatActivity() {

    // Use MockAuthService instead of real service (for testing)
    private val authService = MockAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

Test for MainActivity

```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        launch(MainActivity::class.java)

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Launch main activity

Make up username and password for testing

Simulate click to submit login

Check that UI indicates success

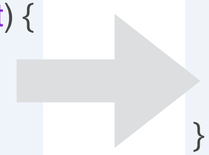
Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

Interface to unify real / mock service

```
interface AuthenticationService {
    fun signIn(username: String, password: String, callback: (Boolean) -> Unit)
}
```

Mock authentication service

```
class MockAuthService {
    fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        callback(true)
    }
}
```



Mock service prepared for injection

```
class MockAuthService : AuthenticationService {
    override fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        callback(true)
    }
}
```

Adapt real service to fit the common interface

```
class CloudAuthServiceAdapter(
    private val cloudAuthService: CloudAuthService
) : AuthenticationService {

    override fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        // Delegate and adapt
        cloudAuthService.signIn(username, password, callback)
    }
}
```


Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

MainActivity using the mock auth service

```
class MainActivity : AppCompatActivity() {

    // Use MockAuthService instead of real service (for testing)
    private val authService = MockAuthService()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

MainActivity prepared for injection

```
class MainActivity : AppCompatActivity() {

    lateinit var authService: AuthenticationService

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    fun setAuthenticationService(service: AuthenticationService) {
        authService = service
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

Setter enables external entity to inject the auth svc

Test for MainActivity

```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        launch(MainActivity::class.java)

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Test that injects the MockAuthService

```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        val mockAuthService = MockAuthService()
        launch(MainActivity::class.java).setAuthenticationService(mockAuthService)

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Inject the auth svc into the main activity

Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

Test for MainActivity

```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        launch(MainActivity::class.java)

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Test that injects the MockAuthService

```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        val scenario = ActivityScenario.launch(MainActivity::class.java)
        scenario.onActivity { activity ->
            val mockAuthService = MockAuthService()
            activity.setAuthenticationService(mockAuthService)
        }

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Inject the auth svc into the main activity

Note: The code examples skip certain details that distract from the purpose of the slide (such as Android specifics)

MainActivity prepared for injection

```
class MainActivity : AppCompatActivity() {

    lateinit var authService: AuthenticationService

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    fun setAuthenticationService(service: AuthenticationService) {
        authService = service
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

Manual means of injection no longer needed

MainActivity: injectable with Hilt

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject lateinit var authService: AuthenticationService

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        findViewById<Button>(R.id.login_button).setOnClickListener {
            val username = findViewById<EditText>(R.id.username).text.toString()
            val password = findViewById<EditText>(R.id.password).text.toString()
            signInUser(username, password)
        }
    }

    private fun signInUser(username: String, password: String) {
        authService.signIn(username, password) { isAuthenticated ->
            runOnUiThread {
                val message = if (isAuthenticated) {
                    "Authentication successful"
                } else {
                    "Authentication failed"
                }
                findViewById<TextView>(R.id.text_view_status).text = message
            }
        }
    }
}
```

Field to be injected

Note: The code examples skip certain details that distract from the purpose of the slide (such as Android, JUnit, or Hilt setup specifics)

Provide instances of real auth svc to Hilt

```
object AppModule {

    @Singleton
    @Provides
    fun provideAuthenticationService(): AuthenticationService {
        // Real implementation provided
        return CloudAuthService()
    }
}
```

- AppModule vs. Factory

```
class OAuth2AuthService @Inject constructor(
    private val oauthClient: OAuth2Client
) : AuthenticationService {

    override fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        // OAuth2 sign-in logic
        oauthClient.requestToken(username, password) { token ->
            val isAuthenticated = token != null
            callback(isAuthenticated)
        }
    }
}
```

- Dependency injection used for production code

```
object OAuth2Module {

    @Singleton
    @Provides
    fun provideAuthenticationService(oauthClient: OAuth2Client): AuthenticationService {
        // Provide OAuth2AuthService
        return OAuth2AuthService(oauthClient)
    }

    // Provide OAuth2Client
    @Singleton
    @Provides
    fun provideOAuth2Client(): OAuth2Client {
        return OAuth2Client()
    }
}
```


Note: The code examples skip certain details that distract from the purpose of the slide (such as Android, JUnit, or Hilt setup specifics)

Provide instances of real auth svc to Hilt

```
object AppModule {

    @Singleton
    @Provides
    fun provideAuthenticationService(): AuthenticationService {
        // Real implementation provided
        return CloudAuthService()
    }
}
```

- AppModule vs. Factory

```
class OAuth2AuthService @Inject constructor(
    private val oauthClient: OAuth2Client
) : AuthenticationService {

    override fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        // OAuth2 sign-in logic
        oauthClient.requestToken(username, password) { token ->
            val isAuthenticated = token != null
            callback(isAuthenticated)
        }
    }
}
```

```
class BiometricAuthService @Inject constructor(
    private val biometricManager: BiometricManager
) : AuthenticationService {

    override fun signIn(username: String, password: String, callback: (Boolean) -> Unit) {
        // Biometric authentication logic
        biometricManager.authenticate { result ->
            callback(result)
        }
    }
}
```

- Dependency injection used for production code

```
object OAuth2Module {

    @Singleton
    @
    fu

    }

    //
    @
    @
    fu

    }
}
```

```
object BiometricModule {

    @Singleton
    @Provides
    fun provideAuthenticationService(biometricManager: BiometricManager): AuthenticationService {
        // Provide BiometricAuthService
        return BiometricAuthService(biometricManager)
    }

    @Singleton
    @Provides
    fun provideBiometricManager(): BiometricManager {
        // Construct and return a BiometricManager instance
        return BiometricManager()
    }
}
```


Note: The code examples skip certain details that distract from the purpose of the slide (such as Android, JUnit, or Hilt setup specifics)

Test that injects the MockAuthService

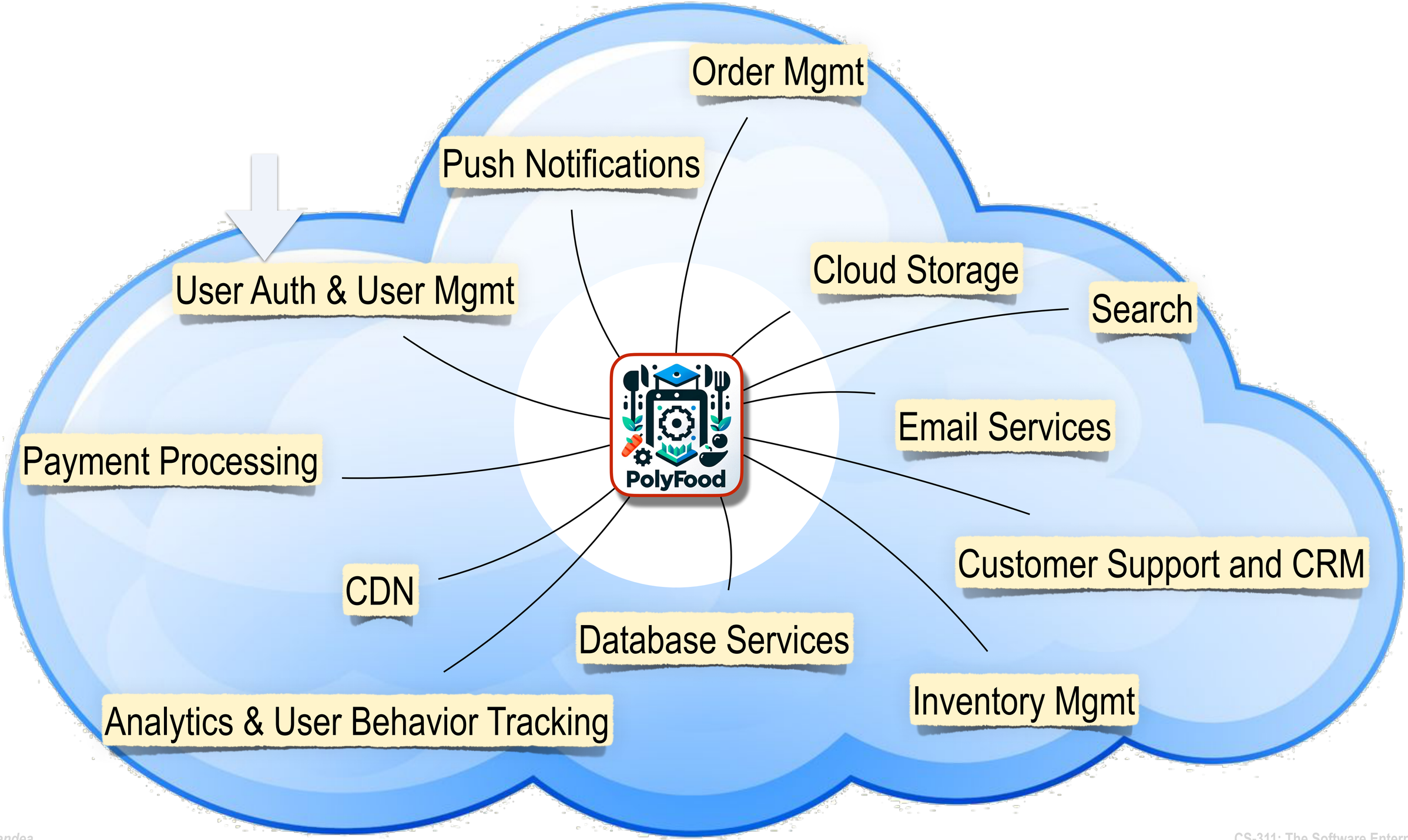
```
class MainActivityTest {

    @Test
    fun testAuthenticationSuccess() {
        val scenario = ActivityScenario.launch(MainActivity::class.java)
        scenario.onActivity { activity ->
            val mockAuthService = MockAuthService()
            activity.setAuthenticationService(mockAuthService)
        }

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

Test that uses Hilt to inject MockAuthService

```
class MainActivityTest {

    @Inject
    lateinit var mockAuthService: MockAuthService

    @Before
    fun init() {
        hiltRule.inject() // do the injection
    }

    @Test
    fun testAuthenticationSuccess() {
        launchActivity<MainActivity>()

        onView(withId(R.id.username)).perform(typeText("testUser"))
        onView(withId(R.id.password)).perform(typeText("testPass"))

        onView(withId(R.id.login_button)).perform(click())

        onView(withId(R.id.text_view_status))
            .check(matches(withText("Authentication successful")))
    }
}
```

```
object TestAppModule {

    @Provides
    fun provideAuthService(): AuthenticationService = MockAuthService()
}
```


Note: The code examples skip certain details that distract from the purpose of the slide (such as Android, JUnit, or Hilt setup specifics)


Order Mgmt
Push Notifications
User Auth & User Mgmt
Cloud Storage
Search
PolyFood
Payment Processing
Email Services
Customer Support and CRM
CDN
Database Services
Inventory Mgmt
Analytics & User Behavior Tracking

```
interface OrderRepository {
    fun saveOrder(order: Order): Boolean // Returns true if save was successful
}

// A simple in-memory mock for testing purposes
class MockOrderRepository : OrderRepository {
    private val savedOrders = mutableListOf<Order>()

    override fun saveOrder(order: Order): Boolean {
        savedOrders.add(order)
        return true // Simulate successful saving for now
    }

    fun orderSaved(order: Order): Boolean = savedOrders.contains(order)
}
```

- Mock all third-party services
- Inject them as dependencies

```
interface PaymentProcessor {
    fun processPayment(order: Order, amount: Double): PaymentStatus
}

// A mock that can simulate success or failures
class MockPaymentProcessor(val success: Boolean) : PaymentProcessor {
    var paymentCalled = false // To track if it was invoked

    override fun processPayment(order: Order, amount: Double): PaymentStatus {
        paymentCalled = true
        return if (success) PaymentStatus.SUCCESS else PaymentStatus.FAILED
    }
}

enum class PaymentStatus { SUCCESS, FAILED }
```

Summary of Mocking and DI

- Mocking
  - *Isolate unit from its dependencies => improves focus of testing*
  - *Controlled behavior => improves test robustness and coverage*
  - *Lightweight, predictable mocks => faster, more stable tests*
- Dependency Injection (DI)
  - *Indirection => decoupling + modularity => maintainability, easy refactoring*
  - *Scalability: code remains flexible and adaptable as the project grows*
- Automated DI
  - *Less boilerplate code, better readability/maintainability*
  - *Powerful scoping and lifecycle management*
- Automated mocking (e.g., MockK)

Regression Testing

Rule #1: When you fix a bug, add a test to prove its absence forever

Group Ordering in PolyFood

- PolyFood rolls out a new feature for group ordering
- Dev team implements the feature
  - *Oops: payment authorizations for single-user orders stop working*
  - *But not noticed, because all testing is focused on the new functionality*

Group Ordering in PolyFood

- PolyFood rolls out a new feature for group ordering
- Dev team implements the feature
  - *Oops: payment authorizations for single-user orders stop working*
  - *But not noticed, because all testing is focused on the new functionality*
- Step 1
  - *devs write a regression test case for standard single-user order and payment*
- Step 2
  - *repeat process to write regression test suite*

Rule #2: SW products evolve constantly, and every change presents risk

Regression Suite — Step 1: Identify Core Features

- Core feature = enables the application's purpose and value proposition
  - *Features that are essential*
  - *Features that are frequently used by users*
  - *Features critical to the business model*
  - *Unique Selling Proposition (USP) features*

Regression Suite — Step 2: Core Features Matrix

Feature	Description	Importance	User Impact	Notes
User Registration	Allows new users to create an account	High	Essential for user engagement and personalization	Critical for first-time users
Login/Logout	Secure authentication mechanism for users	High	Essential for accessing personalized features	Must support social media logins
Restaurant Search	Enables users to search for restaurants by name, cuisine, or location	High	Core to the app's functionality	Should include filters for dietary restrictions
Menu Browsing	Displays available dishes from selected restaurants	High	Directly impacts ordering process	Include photos and descriptions
Order Customization	Allows users to customize orders (e.g., ingredient modifications)	Medium	Enhances user satisfaction	Critical for users with allergies
Shopping Cart Management	Supports adding, removing, and modifying items in the cart	High	Essential for order processing	Include an option to save the cart
Checkout Process	Processes orders with payment and delivery options	High	Critical for completing transactions	Must be secure and support multiple payment methods
Order Tracking	Enables users to track the status of their orders in real-time	Medium	Improves user experience	Push notifications for status updates
Ratings and Reviews	Allows users to rate and review restaurants and dishes	Low	Useful for community engagement	Filter and flag inappropriate content
User Profile Management	Lets users update personal information, payment methods, and order history	Medium	Important for personalization	Include privacy settings
Push Notifications	Sends notifications for order updates, promotions, and app updates	Medium	Enhances user engagement	User option to customize notification preferences
Help and Support	Provides access to FAQ, contact options, and support tickets	Low	Supports user inquiries and issues	Include chat support if possible

Regression Suite — Step 3: Manual Testing

- Do manual tests for each core feature
  - *document the steps and expected results (and note any issues found)*
- Identify potential edge cases
- Identify user paths prone to errors

Regression Suite — Step 4: Write Baseline Suite

- Start writing automated tests
  - *follow the core feature matrix*
  - *possibly refactor code to make it easier to test (e.g., via dependency injection)*
- Write unit tests
- UI tests and user flow
- Mocking and Stubbing
- Integration testing
- Integrate tests into a CI pipeline

Steps 5+: Iteratively Augment, Enhance, and Maintain

- Enhance automated test suite with the finer points from manual testing
  - *Add tests for identified edge cases*
- Add non-functional requirements
  - *Performance, security code scans*
- Refine user flow and usability testing
  - *cover scenarios that manual testing says are crucial for a smooth user experience*
- Every time you fix a bug, add a test that would catch the bug
- Every time you add a new feature, expand regression test suite
- Regularly review and refine

Regression Test Suite — Summary

- Goal of regression test suite
  - *spot regressions on functional and non-functional requirements*
- Regression suite needs to run frequently
  - *need to be fully automated and fast*
- Use CI to automatically run when needed (e.g., on every push)

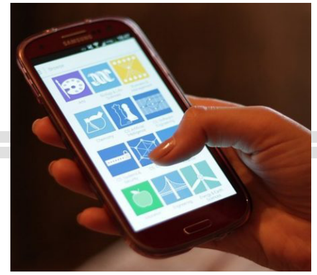
User Interface Testing

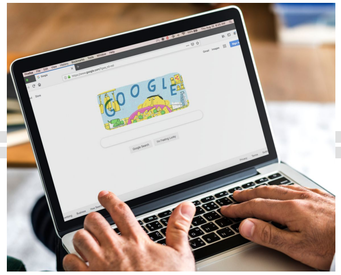
UX Testing is Broader than UI Testing

- Usability testing
- Emotional response testing
- Gather user feedback through surveys, interviews, direct observations
- Accessibility evaluation

Graphical User Interface (GUI)

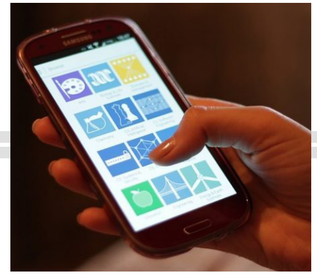
GUI: Testing Functional Properties

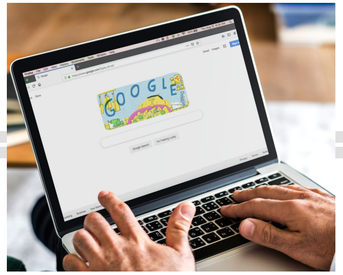




- Do inputs produce the expected outputs?
- Are users prevented from entering unreasonable data?
- Navigational elements
- User feedback elements

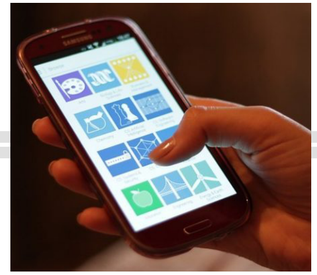
GUI: Testing Non-Functional Properties

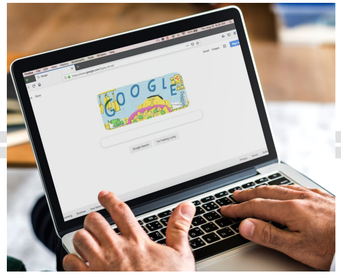




- Consistency (Cross-Browser and Cross-Device Testing)
- Responsiveness
- Performance
- Accessibility
- Security

Manual vs. Automated Testing

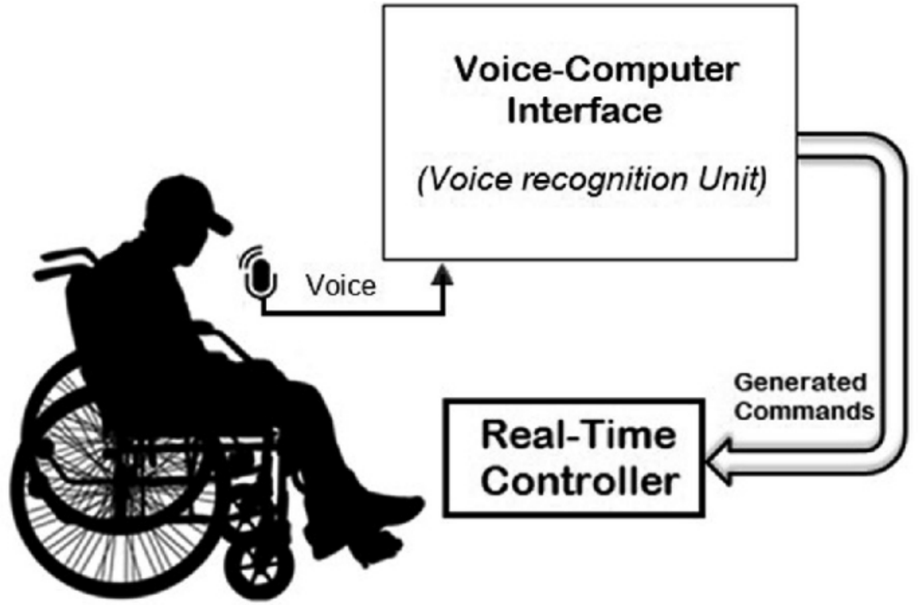




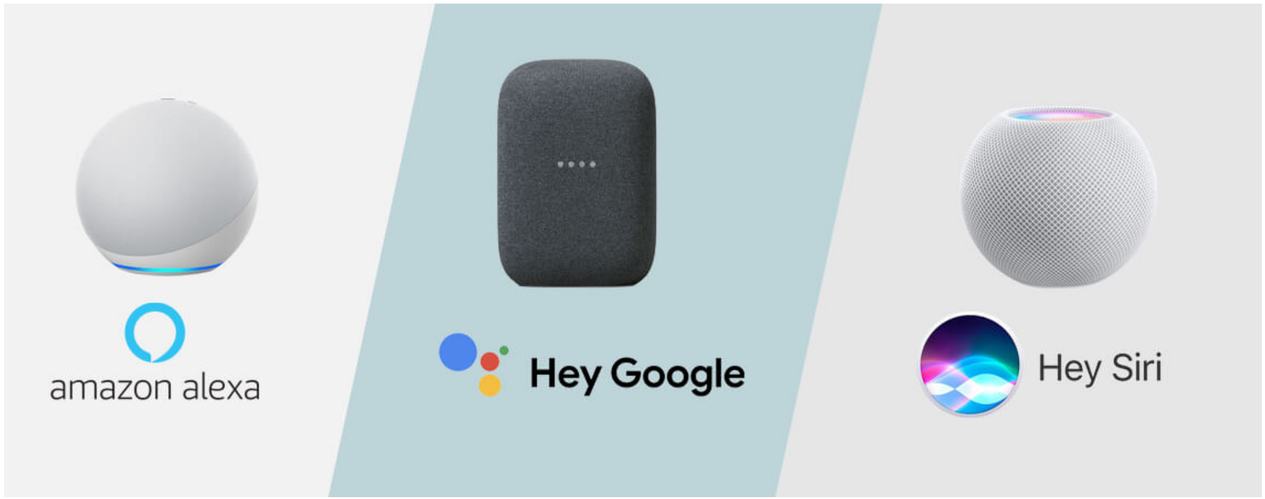
- Manual testing — still lots of it in UI testing
  - *invaluable for capturing the human perspective on usability and aesthetics*
  - *but inefficient, time-consuming, and prone to human error*
- Automated testing
  - *can run automated UI tests on real devices/browsers using services*
  - *use specialized frameworks exist (different for web UI vs. mobile app UI testing)*
- BDD scenarios directly translate to automated UI tests
  - *BDD frameworks (e.g., Cucumber) integrate with UI testing frameworks*
  - *BDD specifications act as living documentation for the application's behaviorr*

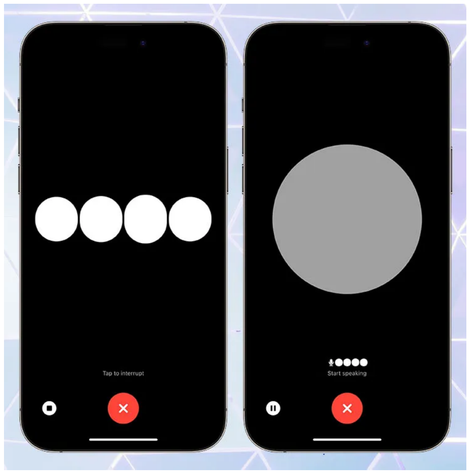
AI in GUI Testing

- Visual regression testing
- Self-healing tests
- Anomaly detection



Voice-Based UI (VUI)



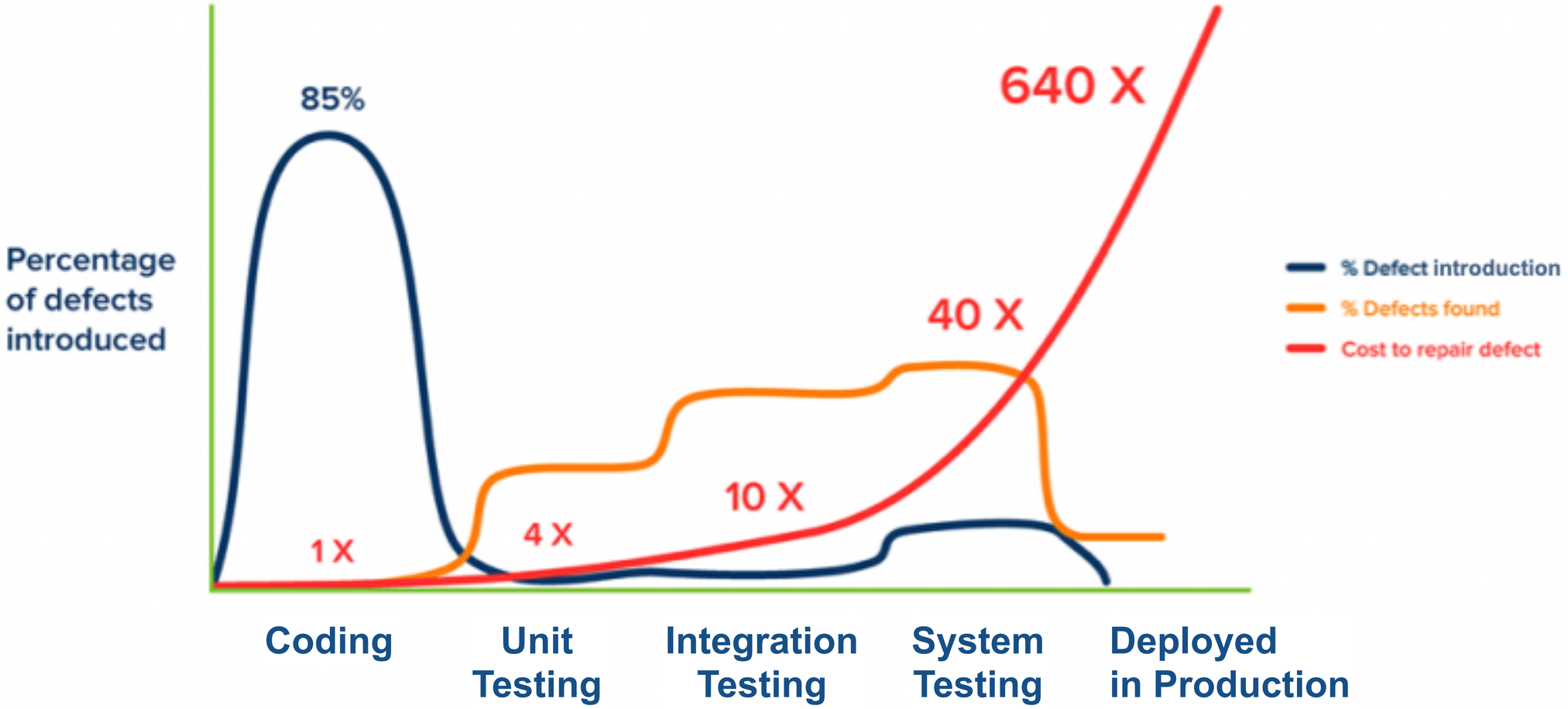


Gesture-Based UI (GbUI)

Testing User Interfaces — Highlights

- UI testing adds another dimension to testing
  - *testing UIs automatically is non-trivial*
  - *but several traditional testing aspects do carry over*
- Page Object Model for GUIs
  - *a way to represent GUI elements as objects*
- non-GUI interfaces
  - *Voice Interfaces*
  - *Gesture Interfaces*

Cost of Bugs (Revisited)



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

https://www.stickyminds.com/article/shift-left-approach-software-testing

Causes of Increasing Bug Costs

- Interdependencies => code becomes interwoven
  - => *longer bug hunting, ripple rffects*
- Reputation and opportunity costs
  - *Unfixed bugs lead to frustrated users and delays in delivering new features*
- Technical debt
  - *bug workarounds => codebase becomes more complex and fragile*
- Lost knowledge
  - *the longer a bug lives, the more likely the original developers have moved on*

Examples

- Boeing 737 MAX crashes (2018-2019)
  - *bugs in the Maneuvering Characteristics Augmentation System (MCAS)*
  - *cost of bug >> cost of fixing the bug*
- Y2K Bug: decades-long bugs led to massive, costly remediation efforts
- Ariane 5 Explosion (1996)
  - *bug stemmed from code reused from the Ariane 4*
- Heartbleed: urgent and widespread updates across the Internet
- Knight Capital: regression led to $440M lost in 45 minutes