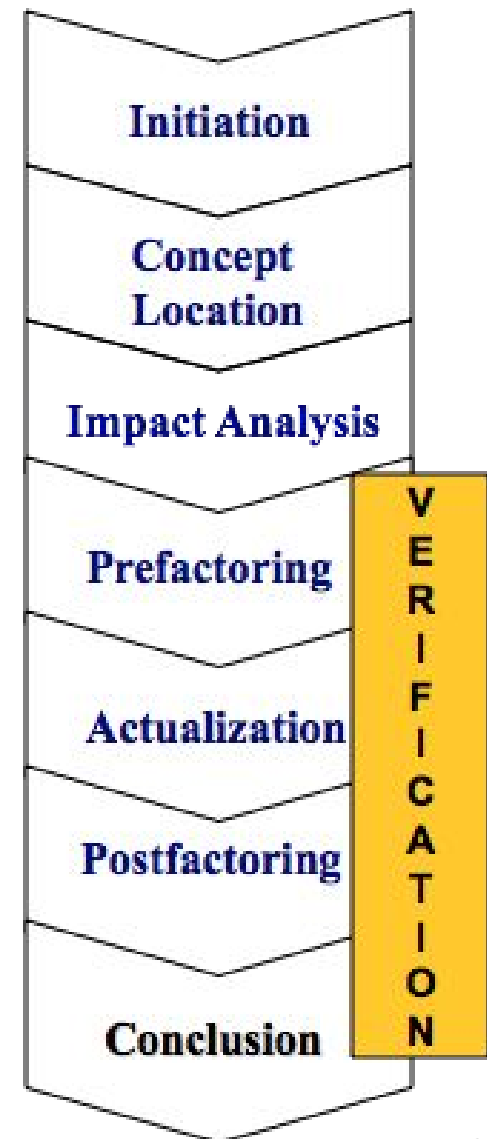
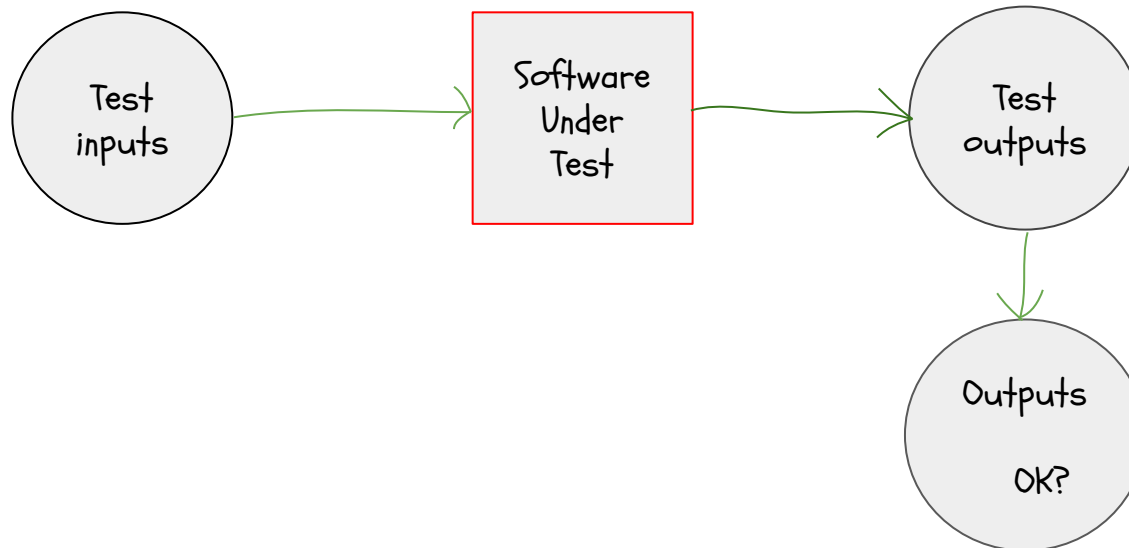


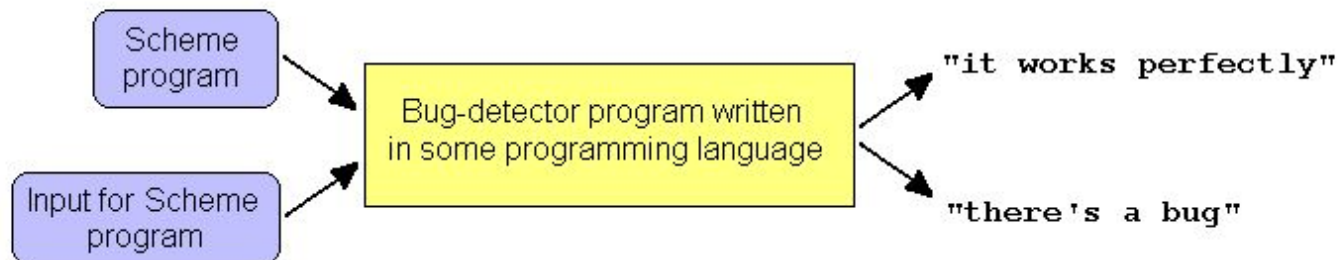
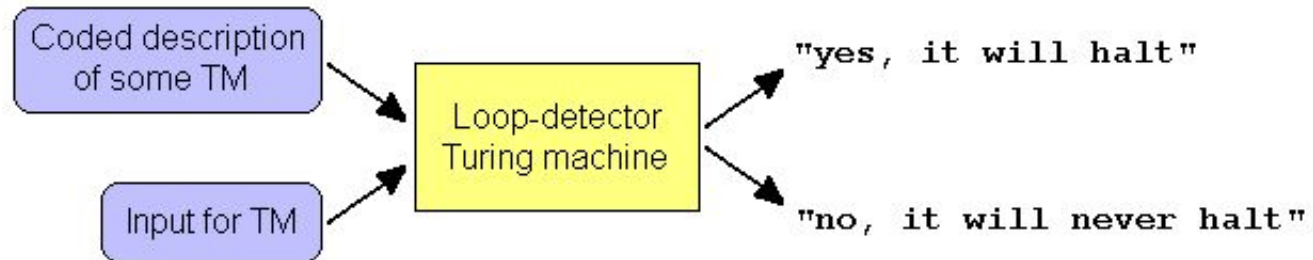
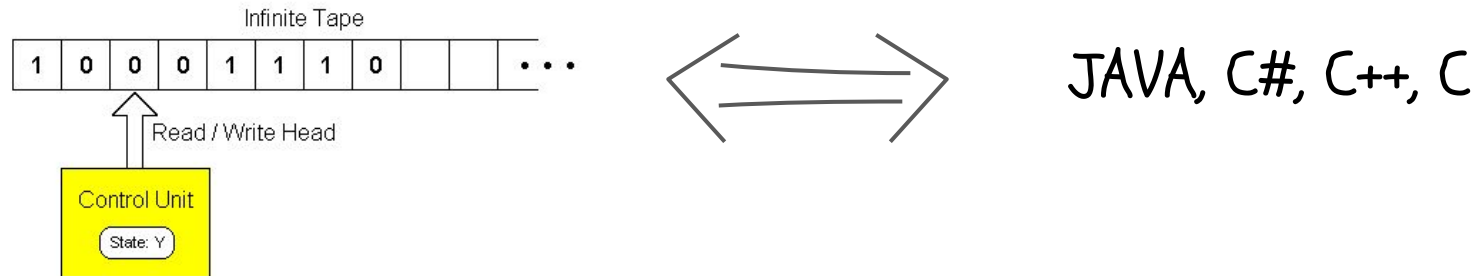
Software Testing

How to make software fail!

Introduction



Turing's halting problem



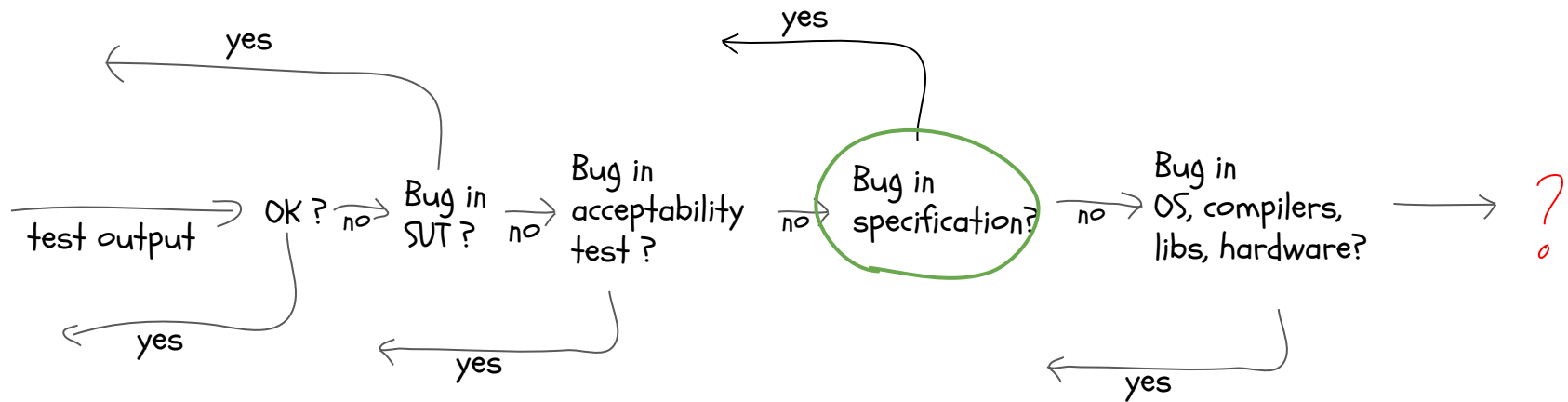
Turing's halting problem

- Theoretical reason for **testing incompleteness**
- It is **theoretically impossible** to create a perfect test suite
- Programmers have been trying to do the best under the circumstances
 - techniques of the testing cannot guarantee a complete correctness of software
 - well designed tests come close to be adequate

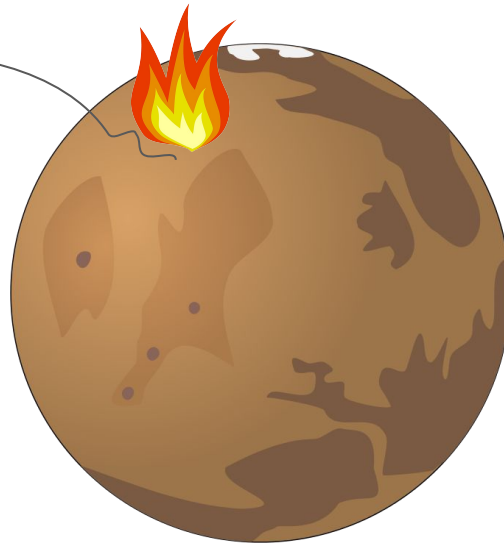
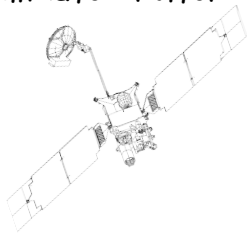
Incompleteness of testing

- “Testing can **demonstrate the presence** of the bugs, but **not their absence** “
- No matter how much testing has been done, residual **bugs** still can hide in the code:
 - they have not been revealed by any tests
 - no test suite can guarantee that the program runs without errors

What is going on?



Mars climate orbiter



Metric: m/s

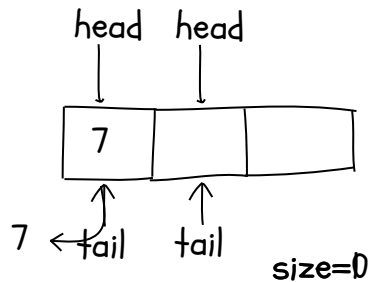
English: ft/s

Testing Example

Fixed Size Queue

- enqueue
- dequeue
- FIFO order

enqueue(7); enqueue(8);
dequeue(); ->
7 then 8



```
q=Queue(2)
r1=q.enqueue(6);
r2=q.enqueue(7);
r3=q.enqueue(8);
r4=q.dequeue();
r5=q.dequeue();
r6=q.dequeue();
```

```
public class Queue {
    public Queue(int sizeMax) {
        assert sizeMax > 0;
        this.max = sizeMax;
        this.head = 0;
        this.tail = 0;
        this.size = 0;
        this.data =
            new int[sizeMax];
    }
    public boolean empty() {
        return size == 0;
    }
    public boolean full() {
        return size == max;
    }
}
```

→ true, true, false, 6, 7, null

```
public boolean enqueue(int x) {
    → if (size == max)
        return false;
    data[tail] = x;
    → size += 1;
    → tail += 1;
    if (tail == max)
        tail = 0;
    → return true;
}

public Integer dequeue() {
    → if (size == 0)
        return null;
    int x = data[head];
    → size -= 1;
    → head += 1;
    if (head == max) {
        head = 0;
    }
    → return x;
}
}
```

Equivalent Tests

```
q.enqueue(7);  
x=q.dequeue();  
if( x==7 )  
    success
```

```
q.enqueue(?);  
x=q.dequeue();  
if( x== ? )  
    success
```

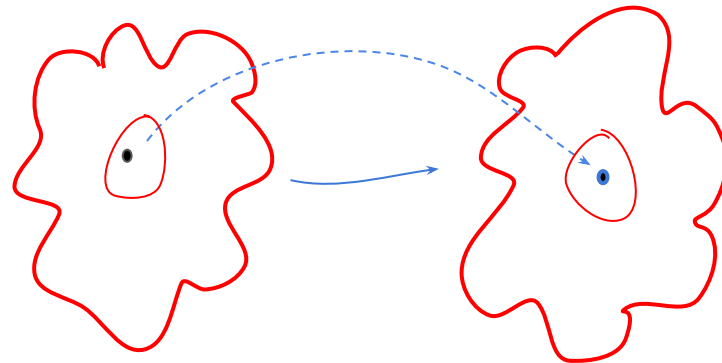
Large Integers?

?

```
q.enqueue(7);  
sleep(100);  
x=q.dequeue();  
if( x==7 )  
    success
```

Input space

Output Space

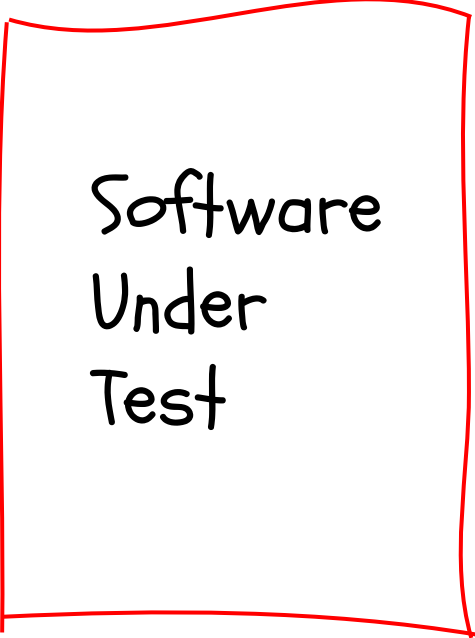


Testing the Queue

```
public static boolean testEquality() {
    IQueue q = new Queue(2);
    boolean res = q.empty();
    if (!res) {
        out.println("test1 NOT OK");
        return false;
    }
    res = q.enqueue(10);
    if (!res) {
        out.println("test1 NOT OK");
        return false;
    }
    res = q.enqueue(11);
    if (!res) {
        out.println("test1 NOT OK");
        return false;
    }
    int x = q.dequeue();
    if (x != 10) {
        out.println("test1 NOT OK");
        return false;
    }
    x = q.dequeue();
    if (x != 11) {
        out.println("test1 NOT OK");
        return false;
    }
    res = q.empty();
    if (!res) {
        out.println("test1 NOT OK");
        return false;
    }
    out.println("test1 OK");
    return true;
}
```

```
public static boolean testEnqueueTail() {
    Queue q = new Queue(2);
    boolean res = q.empty();
    if (!res) {
        out.println("test2 NOT OK");
        return false;
    }
    res = q.enqueue(1);
    if (!res) {
        out.println("test2 NOT OK");
        return false;
    }
    res = q.enqueue(2);
    if (!res) {
        out.println("test2 NOT OK");
        return false;
    }
    q.enqueue(3);
    if (q.getTail() != 0) {
        out.println("test2 NOT OK");
        return false;
    }
    out.println("test2 OK");
    return true;
}
```

Creating Testable Software



Software
Under
Test

- Clean Code
- Refactor
- Describe what it **does** and how it **interacts**
- No extra Threads
- No swap of global variables
- No pointer soup
- Module unit tests
- Support fault injection
- Assertions, Assertions
Assertions !!!

Assertions

Assertions

Assertion: Executable check for a property that must be true (invariant)

```
double sqrt(arg){  
    //...compute result...  
    assert result > 0;  
    return result;  
}
```

~~`assert foo0==0;`
// where foo0 changes a global variable~~

`assert |+|==2;`

Rule1: Assertions are not for error handling

Rule2: NO SIDE EFFECTS

Rule3: No silly assertions

Queue Assertions

```
public void checkRep() {  
    // Size  
    assert this.getSize() >= 0 && this.getSize() <= this.getMax();  
  
    if (this.getTail() > this.getHead()) {  
        assert (this.getTail() - this.getHead()) == this.getSize();  
    }  
  
    if (this.getTail() < this.getHead()) {  
        assert (this.getHead() - this.getTail()) == (this.getMax() - this.getSize());  
    }  
  
    if (this.getTail() == this.getHead()) {  
        assert (this.getSize() == 0) || (this.getSize() == this.getMax());  
    }  
}
```

Why Assertions?

- Make code self-checking, leading to effective testing
- Make code fail early, closer to the bug
- Assign blame
- Document assumptions, preconditions, postconditions and invariants

In production?

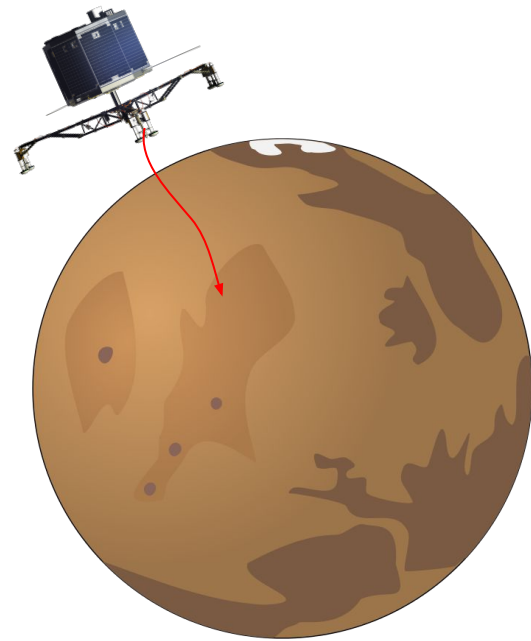
- GCC: ~9000 assertions
- LLVM: ~ 13,000 assertions
- One assertion per 110 L.O.C

Disable Assertions?

- Advantages:
 - Code runs faster
 - Code keeps running
- Disadvantages:
 - Code can rely on a side-effect assertion?
 - Even in production code, may be better to fail early

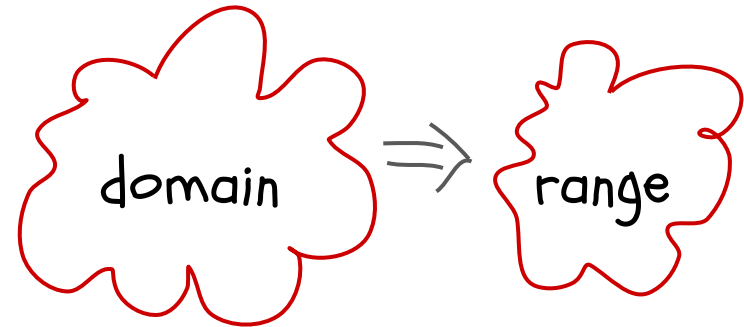
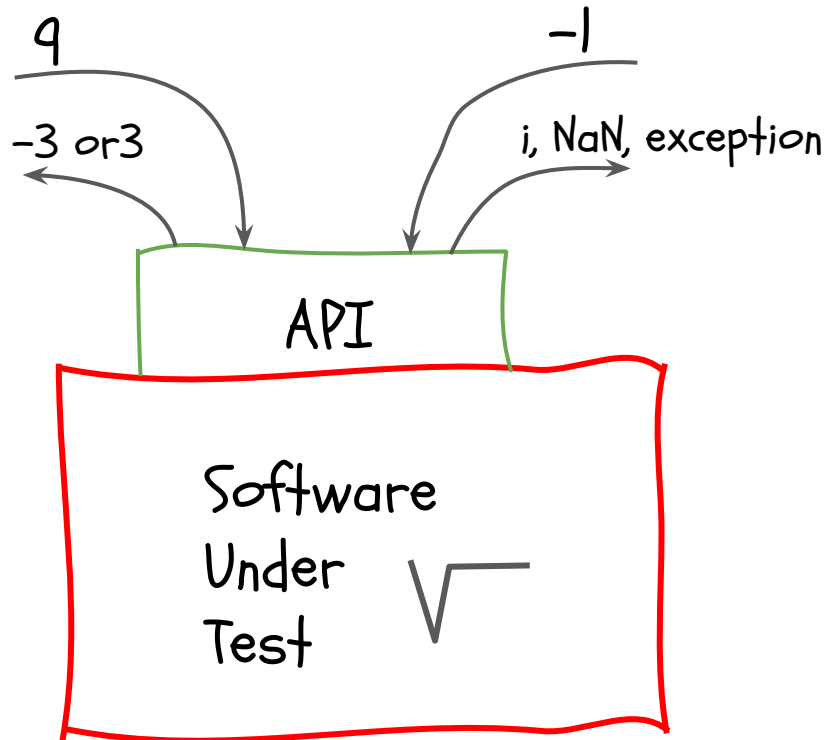
When to use Assertions?

- Running Software that can be recovered by failing early
- Disable in mission critical stage when it is better to continue then recover



Software Under Test

Specifications



$\text{sqrt}: \mathbb{R}^+ \rightarrow \mathbb{R}^+$

$\text{sqrt}: \mathbb{R} \rightarrow \mathbb{C}$

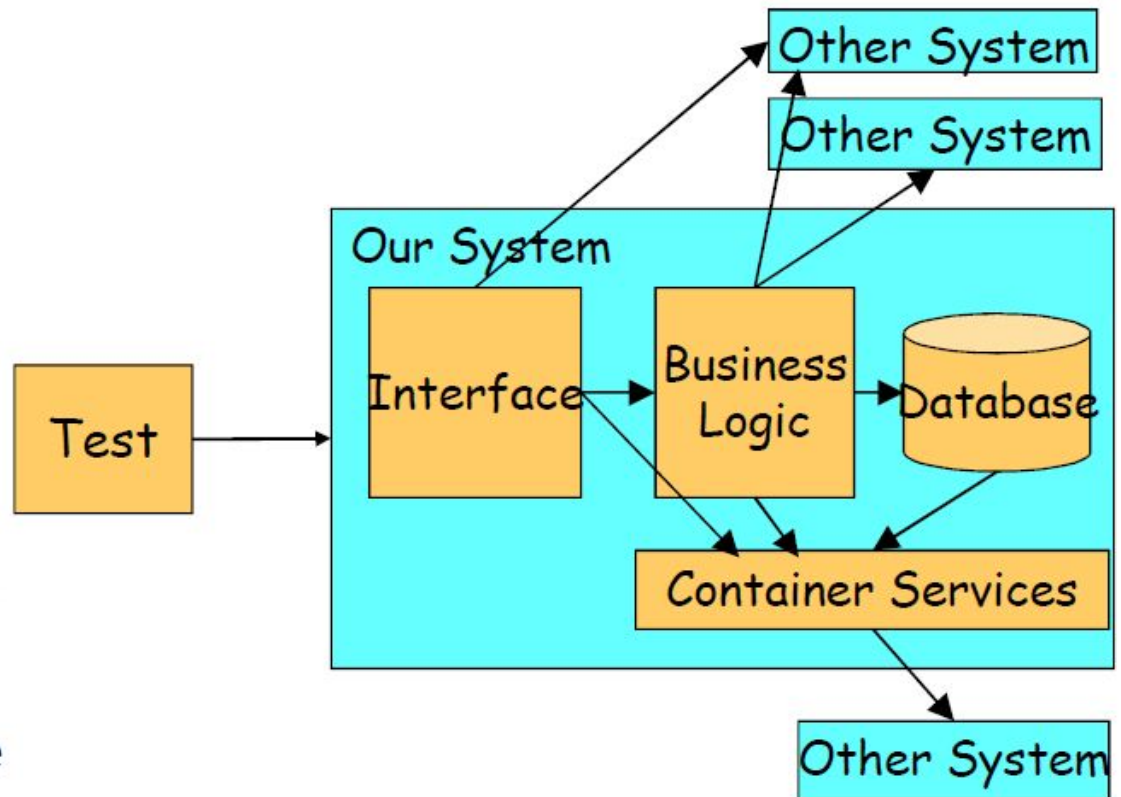
$\text{sqrt}: \mathbb{F}^+ \rightarrow \mathbb{F}^+$

$\text{sqrt}: \mathbb{F} \rightarrow \mathbb{F}^+ \cup \text{exception}$

The Fragile Test Problem

What, when changed,
may break our tests
accidentally:

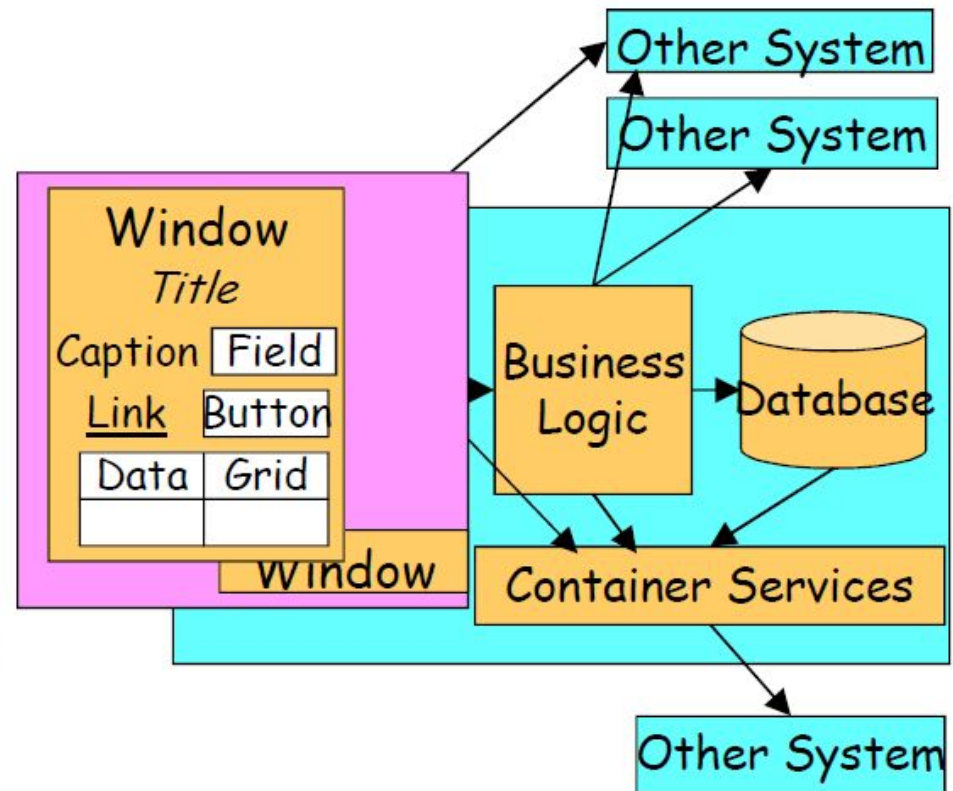
- Behavior Sensitivity
 - » Business logic
- Interface Sensitivity
 - » User or system
- Data Sensitivity
 - » Database contents
- Context Sensitivity
 - » Other system state



In Agile, these are all changing all the time!

Interface Sensitivity

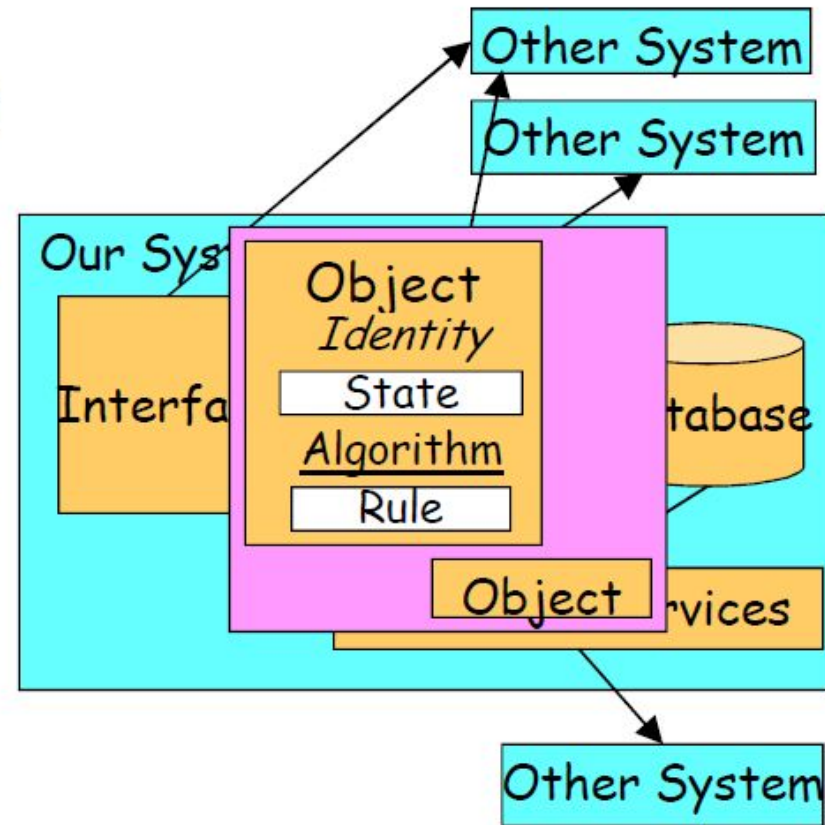
- Tests must interact with the SUT through some interface
- Any changes to interface may cause tests to fail.
 - User Interfaces:
 - » Renamed/deleted windows or messages
 - » New/renamed/deleted fields
 - » New/renamed/deleted data values in lists
 - Machine-Machine Interfaces:
 - » Renamed/deleted functions in API
 - » Renamed/deleted messages
 - » New/changed/deleted function parameters or message fields



E.g.: Move tax field
to new popup window

Behavior Sensitivity

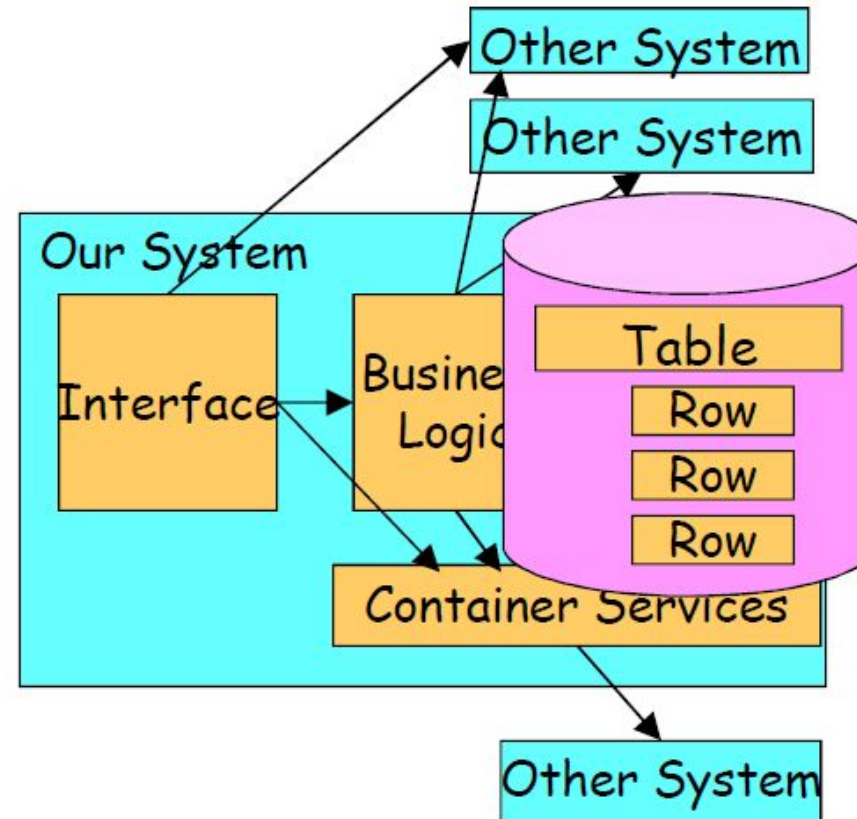
- **Tests must verify the behavior of the system.**
 - Behavior also involved in test set up & tear down
- **Any changes to business logic may cause tests to fail.**
 - New/renamed/deleted states
 - New/changed/removed business rules
 - Changes to business algorithms
 - Additional data requirements



E.g.: Change from
GST+PST to HST

Data Sensitivity

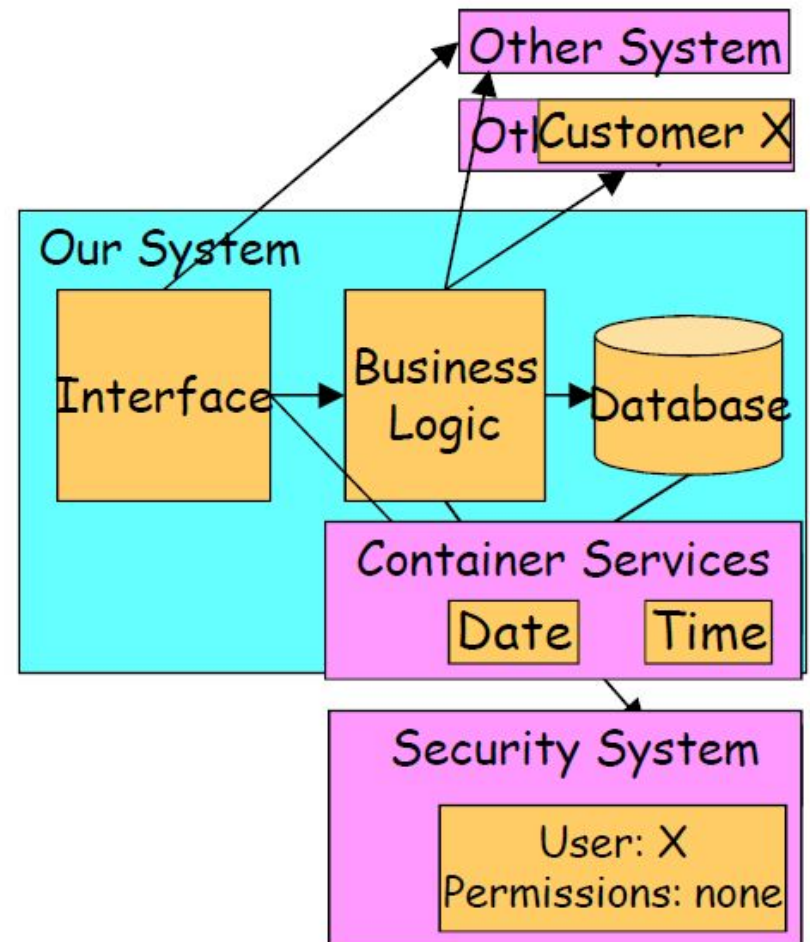
- **All tests depend on “test data” which are:**
 - Preconditions of test
 - Often stored in databases
 - May be in other systems
- **Changing the contents of the database may cause tests to fail.**
 - Added/changed/deleted records
 - Changed Schema



E.g.: Change customer's billing terms

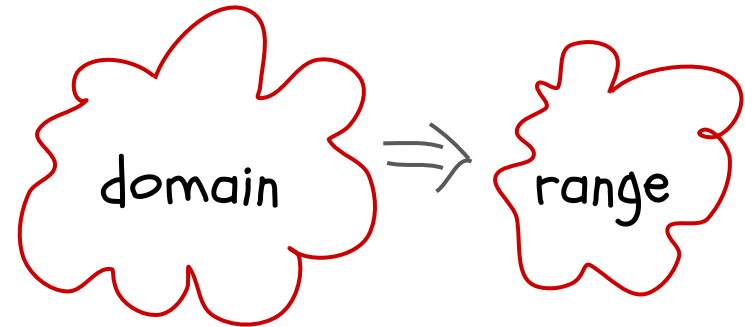
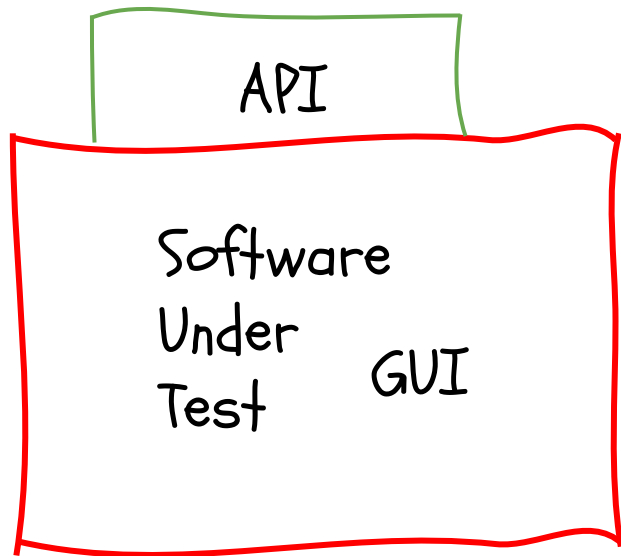
Context Sensitivity

- **Tests may depend on inputs from another system**
 - State stored outside the application being tested
 - Logic which may change independently of our system
- **Changing the state of the context may cause tests to fail.**
 - State of the container
 - » e.g. time/date
 - State of related systems
 - » Availability, data contents



E.g.: Run test in a shorter/longer month

Testing a GUI

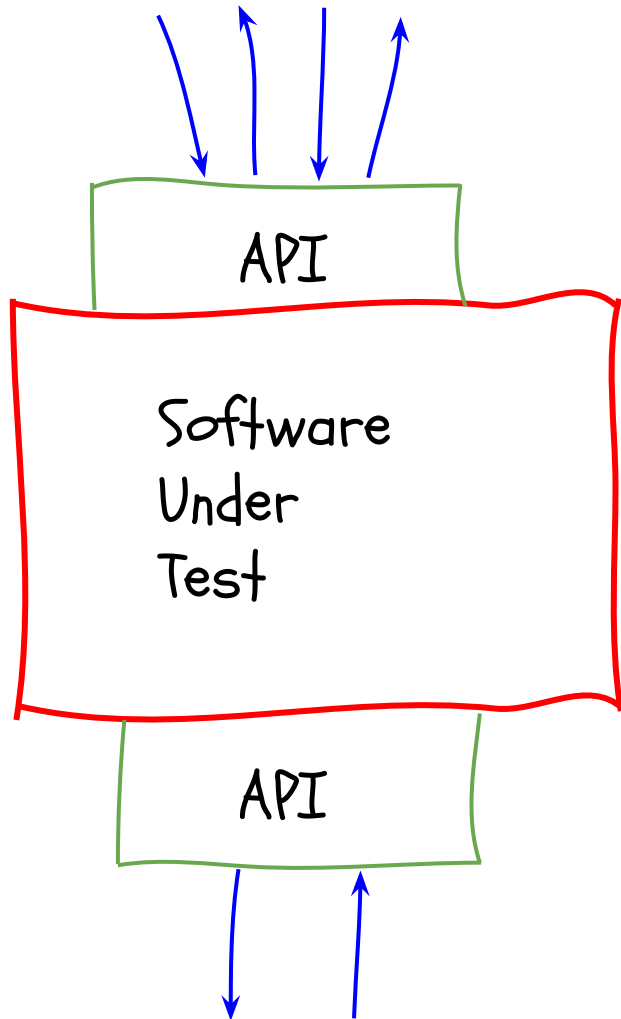


GUI: clicks,
events,
swipes...

GUI
-> App.
states

Record and Play using scripts

Fault Injection

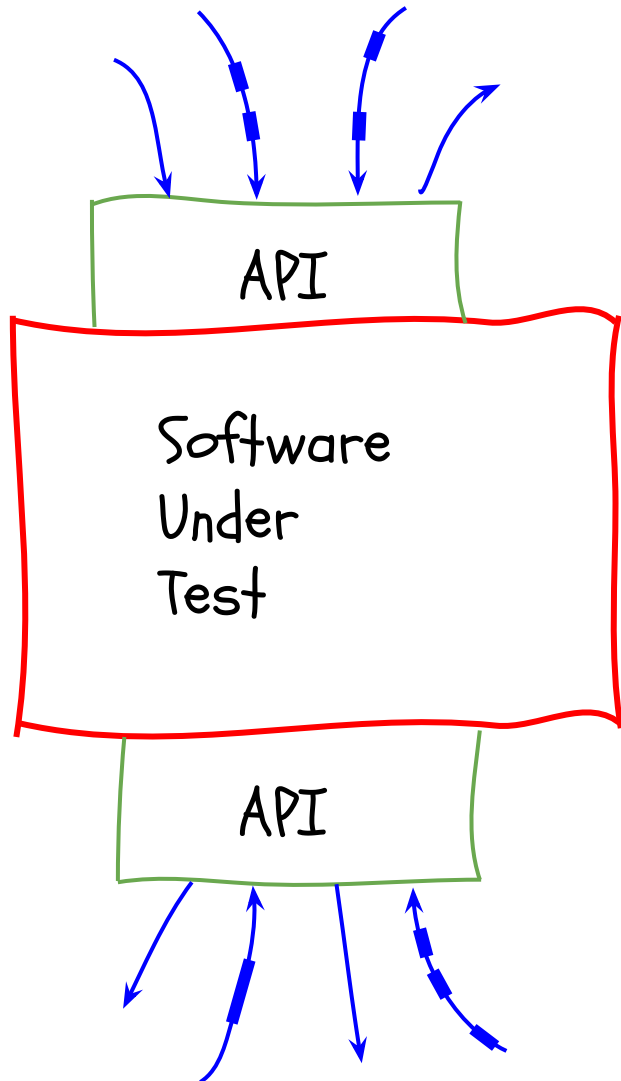


```
file=open("/tmp/foo", 'w')
```

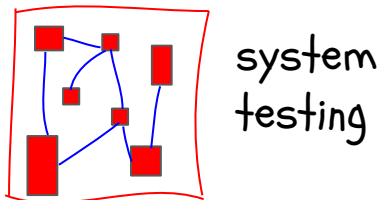
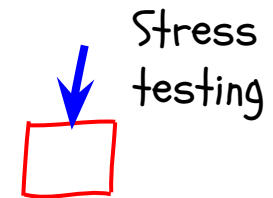
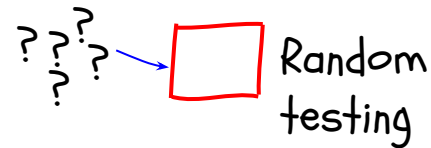
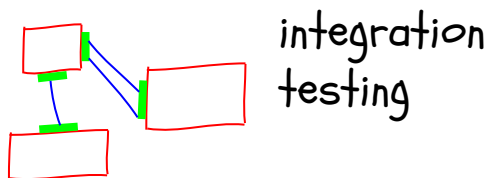
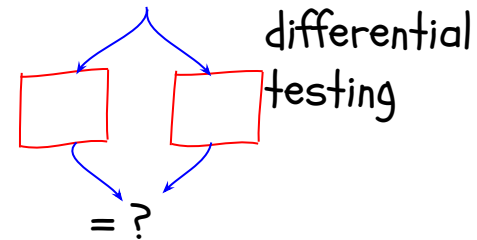
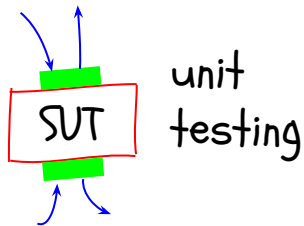
```
file=my-open("/tmp/foo", 'w')
```

my-open {
 succeed 100 times
 then fail 1% of calls

Time Dependent Problems



Kinds of Testing



Acceptance Tests

Acceptance Tests

- Tests defined by/with the help of the user based on the requirements
- Traditionally
 - manual tests
 - by the customer
 - after the software is delivered
 - based on use cases / user stories
- Agile software development
 - automatic tests: JUnit, Fit, . . .
 - created before the user story is implemented

Example of acceptance tests

Name: Login Admin

Actor: Admin

Precondition: Admin is not logged in

Main scenario:

1. Admin enters password
2. System responds true

Alternative scenarios:

- 1a. Admin enters wrong password
- 1b. The system reports that the password is wrong and the use case starts from the beginning

Postcondition: Admin is logged in

Manual tests

Prerequisite: the password for the administrator is “adminadmin”

Input	Step	Expected Output	Fail	OK
	Startup system	“0) Exit” “1) Login as administrator”		
“1”	Enter choice	“password”		
“adminadmin”	Enter string	“logged in”		

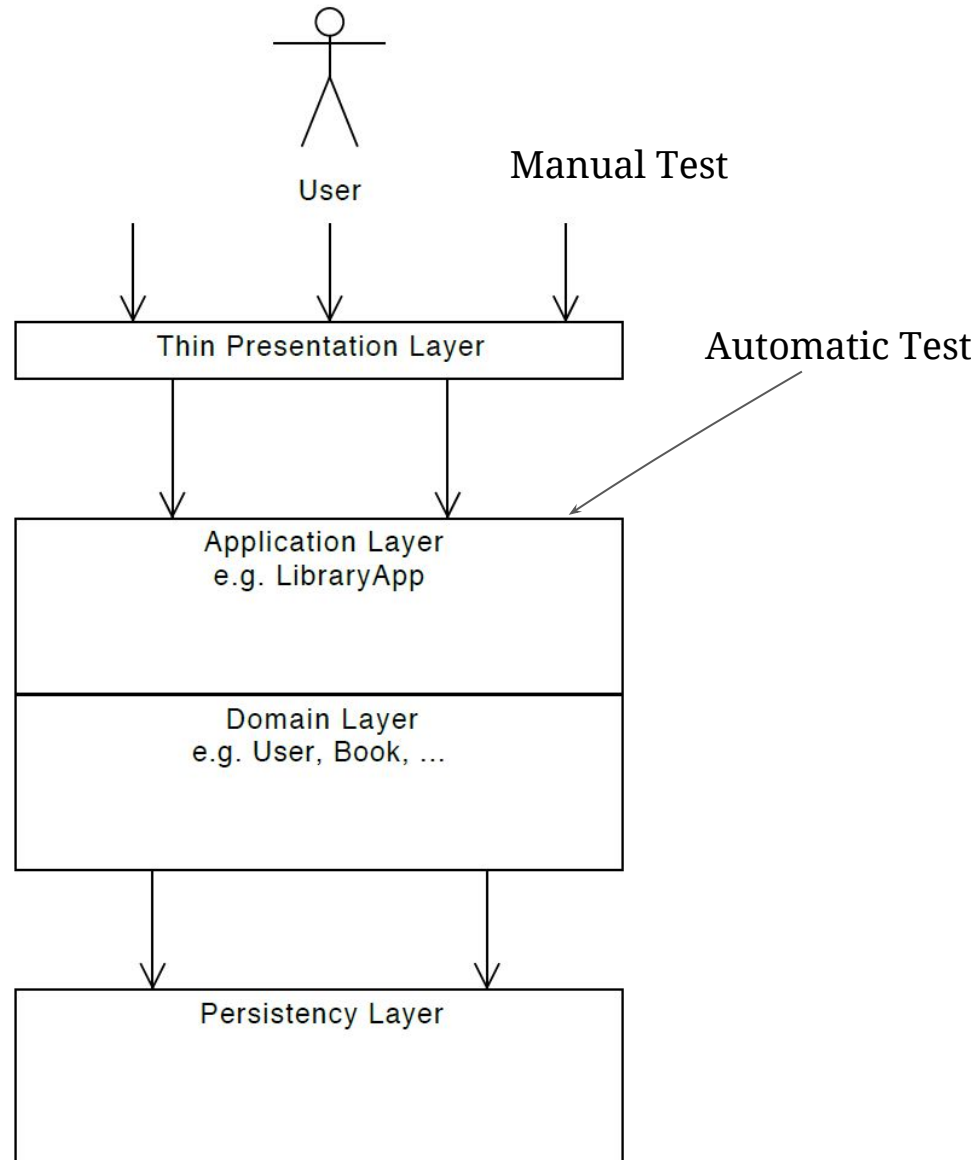
Prerequisite: the password for the administrator is “adminadmin”

Input	Step	Expected Output	Fail	OK
	Startup system	“0) Exit” “1) Login as administrator”		
“1”	Enter choice	“password”		
“admin”	Enter string	“Password incorrect” “0) Exit” “1) Login as administrator”		

Manual vs. automated tests

- Manual tests should be avoided
 - They are expensive (time and personal) to execute:
Can't be run often
- Automated tests
 - Are cheap (time and personal) to execute:
Can be run as soon something is changed in the system
 - immediate feedback if a code change introduced a bug →
Regression tests
 - More difficult (but not impossible) when they include the UI
 - Solution: Test under the UI

Testing under the UI



Automatic tests

Successful login

```
@Test
public void testLoginAdmin() {
    LibraryApp libApp = new LibraryApp();
    assertFalse(libApp.adminLoggedIn());

    boolean login = libApp.adminLogin("adminadmin");
    assertTrue(login);
    assertTrue(libApp.adminLoggedIn());
}
```

Failed login

```
@Test
public void testWrongPassword() {
    LibraryApp libApp = new LibraryApp();
    assertFalse(libApp.adminLoggedIn());

    boolean login = libApp.adminLogin("admin");
    assertFalse(login);
    assertFalse(libApp.adminLoggedIn());
}
```

JUnit

JUnit Intro

- JUnit is a framework for writing tests
- Written by **Erich Gamma** (Design Patterns) and **Kent Beck** (eXtreme Programming)
- Unit-, component-, and acceptance tests
- <http://www.junit.org>
- xUnit

JUnit Test Case

```
public class EnqueueTest {

    private IQueue q;

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        q = new CheckRepWrapper(new Queue(2));
    }

    @After
    public void tearDown() {
        q.empty();
    }
}
```

```
/**
 * Test of enqueue method, of Queue.
 */
@Test
public void testEnqueue() {
    boolean res = q.empty();
    assertTrue(res);

    res = q.enqueue(1);
    assertTrue(res);

    res = q.enqueue(2);
    assertTrue(res);

    q.enqueue(3);
    assertTrue(q.getTail() == 0);
}
```

JUnit assertions

General Assertions

```
import static org.junit.Assert.*;

assertTrue(bexp)
assertTrue(msg, bexp)
```

Specialised assertions for readability

```
assertFalse(bexp)
fail()
assertEquals(exp, act)
assertNull(obj)
assertNotNull(obj)
...
```


TestSuite

```
@RunWith(Suite.class)
@Suite.SuiteClasses({dk.sdu.mmmi.fixedsizequeue.EnqueueTest.class,
                    dk.sdu.mmmi.fixedsizequeue.DequeueTest.class})
public class QueueTestSuite {

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }
}
```

JUnit: exceptions

Test that method `m()` throws an exception

MyException

```
@Test
public void testMThrowsException() {
    ... try {
        m();
        fail(); // If we reach here, then the test fails because
                // no exception was thrown
    } catch(MyException e) {
        // Do something to test that e has the correct values
    }
}
```

Alternative

```
@Test(expected=MyException.class)
public void testMThrowsException() {..}
```

Links

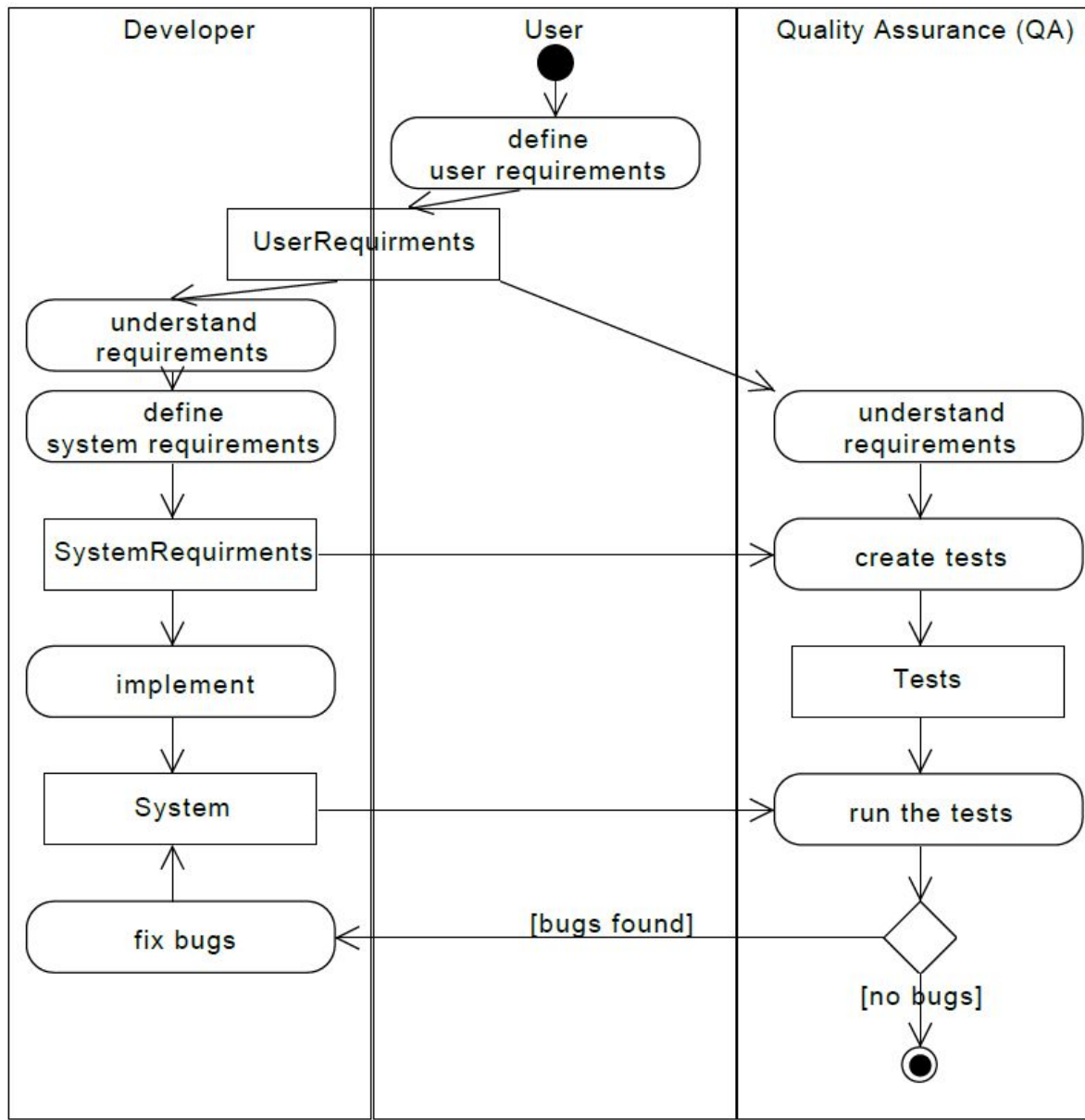
- [xUnit](#)
- [CppUnit](#)
- [JUnit](#)
- [Mockito](#)
- [Sikuli Script](#)
- [DbUnit](#)

Test-Driven Development

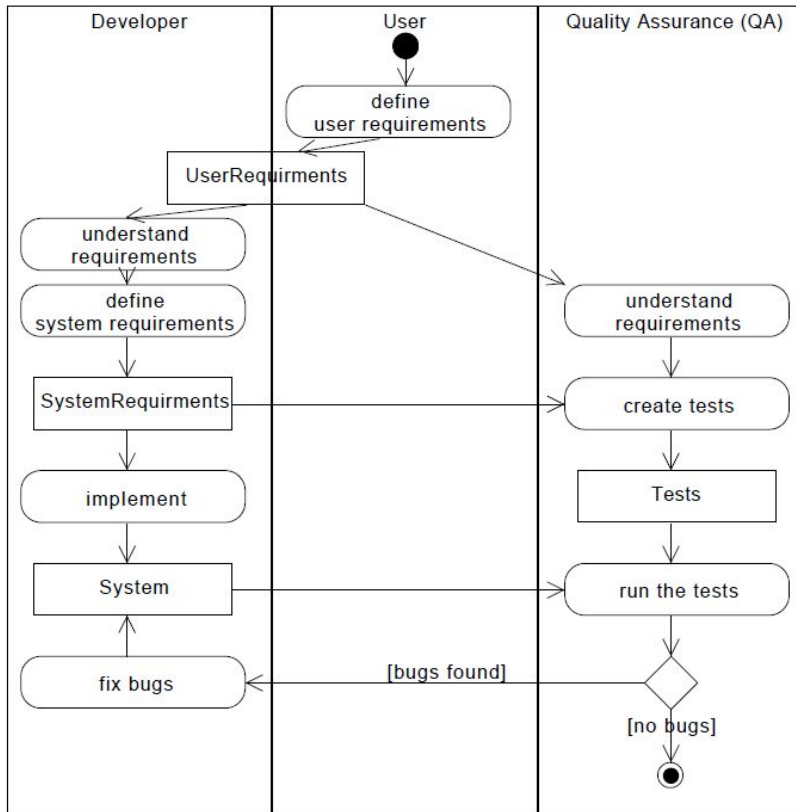
TDD

- Test before the implementation
- Tests = expectations on software
- All kind of tests:
unit, component, and system tests

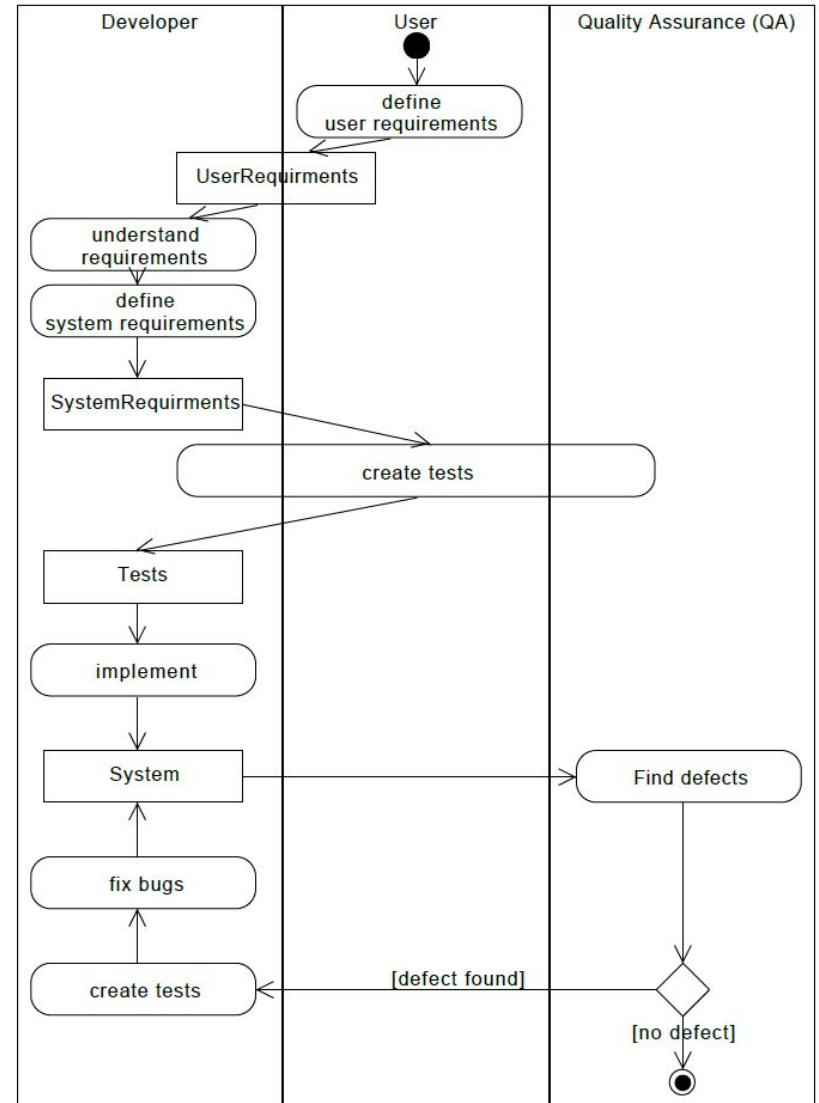
Traditional Testing



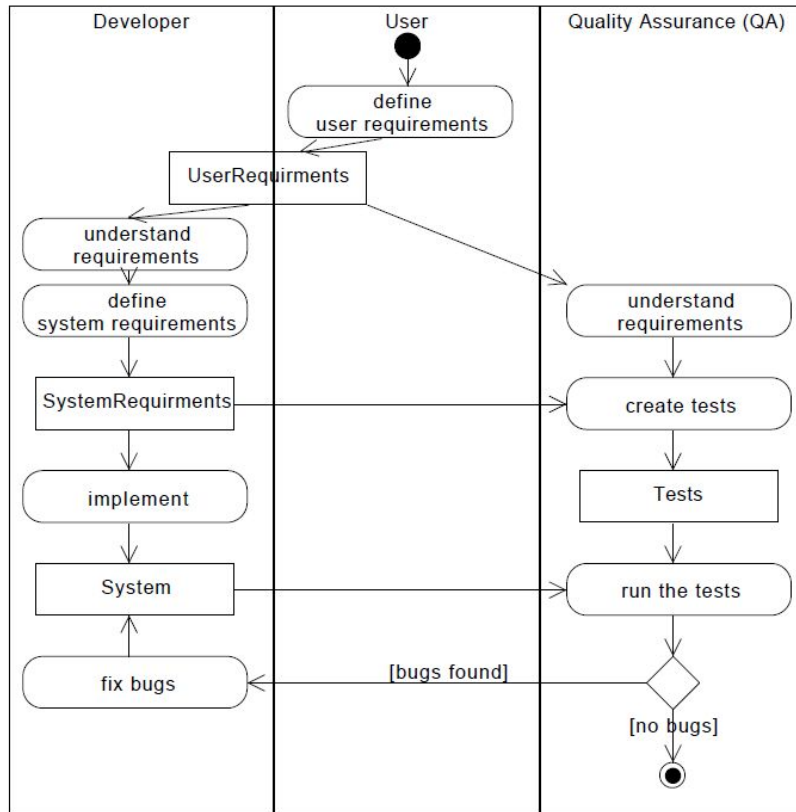
Traditional Testing



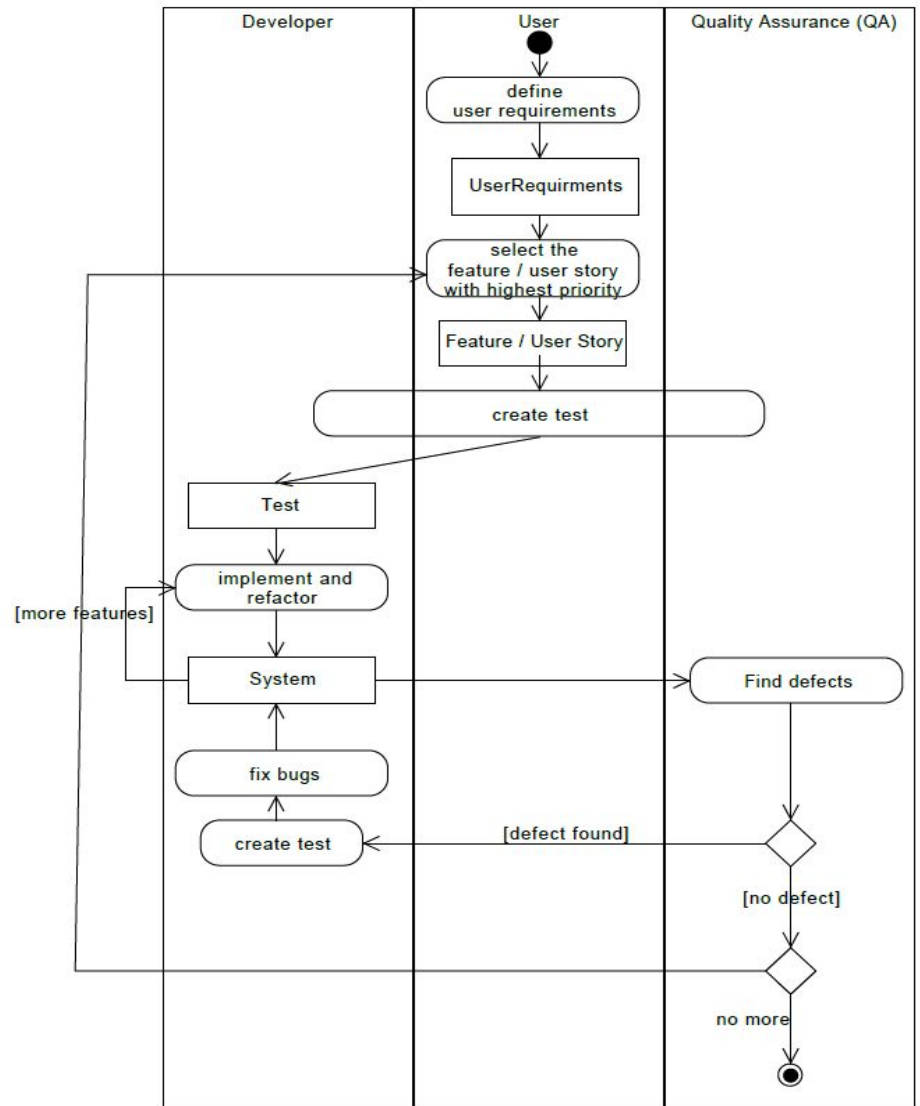
Moving to TDD



Traditional Testing



Real TDD



TDD cycle

- Repeat for functionality, bug, . . .
 - red: Create a failing test
 - green: Make the test pass
 - refactor : clean up your code
- Until: no more ideas for tests
- Important:
 - One test at a time
 - Implement only as much code so that the test does not fail.
 - If the method looks incomplete,
 - add more failing tests that force you to implement more code

Ideas for tests

- Use case scenarios (missing functions):
Acceptance tests
- Possibility for defects (missing code): Defect tests
- You want to write more code than is necessary to pass the test
- Complex behaviour of classes: Unit tests
- Code experiments: "How does the system behave, if . . ."
- Make a list of new test ideas

TDD example: Borrow Book

Name: borrow book

Description: the user borrows a book

Actor: user

Main scenario:

1. the user borrows a book

Alternative scenario:

1. the user wants to borrow a book, but has already 10 books borrowed
2. the system presents an error message

Create a test for the main scenario

Testdata:

- A user with CPR "1234651234" and book with signature "Som001"

Testcase:

- Retrieve the user with CPR number "1234651234"
- Retrieve the book by the signature "Som001"
- The user borrows the book
- The book is in the list of books borrowed by that user

Create a test for the main scenario

```
@Test
public void testBorrowBook() throws Exception {

    String cprNumber = "1234651234";
    User user = libApp.userByCprNumber(cprNumber);
    assertEquals(cprNumber, user.getCprNumber());

    String signature = "Som001";
    Book book = libApp.bookBySignature(signature);
    assertEquals(signature, book.getSignature());

    List<Book> borrowedBooks = user.getBorrowedBooks();
    assertFalse(borrowedBooks.contains(book));

    user.borrowBook(book);

    borrowedBooks = user.getBorrowedBooks();
    assertEquals(1, borrowedBooks.size());
    assertTrue(borrowedBooks.contains(book));
}
```

Implement the main scenario

```
public void borrowBook(Book book) {  
    borrowedBooks.add(book);  
}
```

Create a test for the alternative scenario

Testdata:

- a user with CPR "1234651234", book with signature "Som001", and 10 books with signatures "book1", . . . , "book10"

Testcase:

- Retrieve the user with CPR number "1234651234"
- Retrieve and borrow the books with signature "book1", . . . , "book10"
- Retrieve and borrow the book by the signature "Som001"
- Check that a `TooManyBooksException` is thrown

Implementation of the alternative scenario

```
public void borrowBook(Book book) throws  
TooManyBooksException  
    if (borrowedBooks.size() >= 10) {  
        throw new TooManyBooksException();  
    }  
    borrowedBooks.add(book);  
}
```


More test cases

Test Data:

- What happens if `book==null` in `borrowBook`?

TestCase:

- Retrieve the user with CPR number "1234651234"
- Call the `borrowBook` operation with the `null` value
- Check that the number of borrowed books has not changed

Refactoring and TDD

- Third step in TDD
- Restructure the system without changing its functionality
- **Goal:** improve the design of the system, e.g. remove code duplication (**DRY** principle)
- Necessary step
- Requires good testsuite:
 - later in the course more about refactoring mechanics

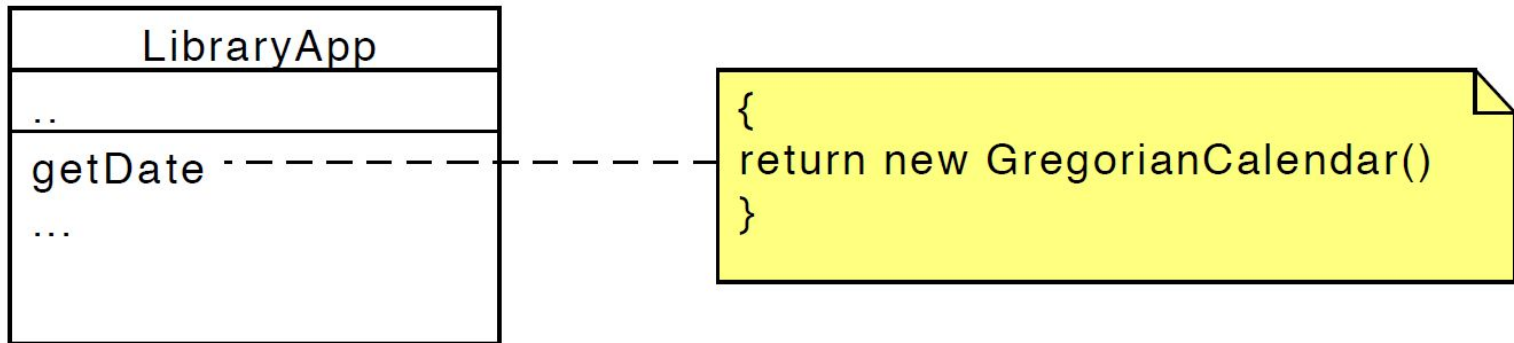
TDD: Advantages

- Test benefits
 - Good code coverage: Only write production code to make a failing test pass
- Design benefits
 - Helps design the system: defines usage of the system before the system is implemented
- Testable system

Mock Objects

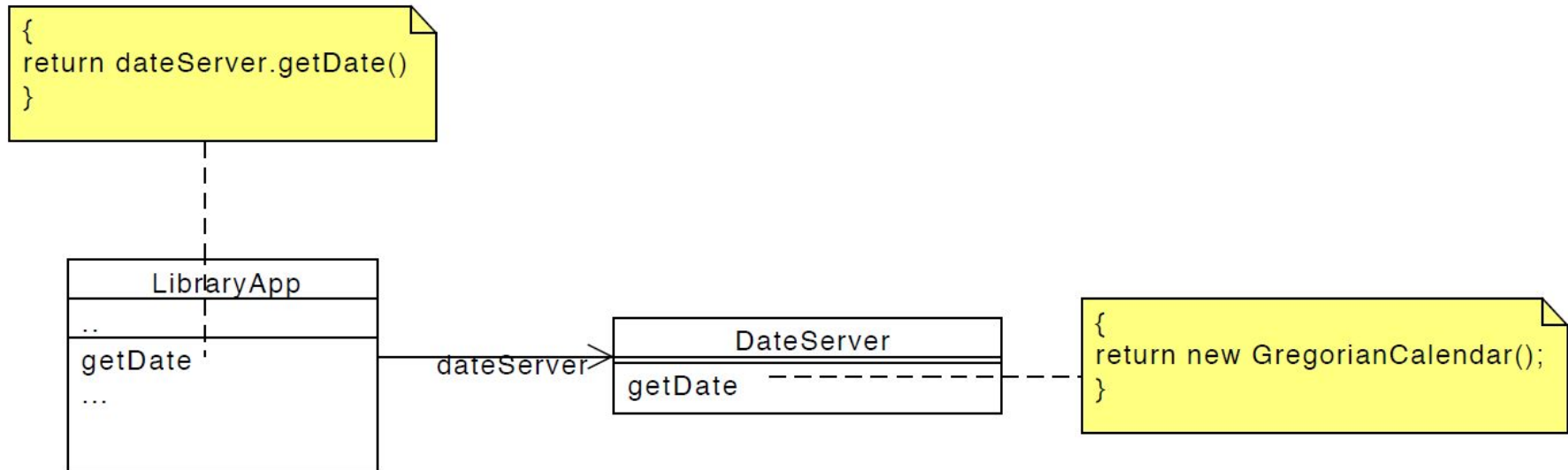
Problem

- ▶ How to test that a book is overdue?
 - ▶ Borrow the book today
 - ▶ Jump to the data in the future when the book is overdue
 - ▶ Check that the book is overdue

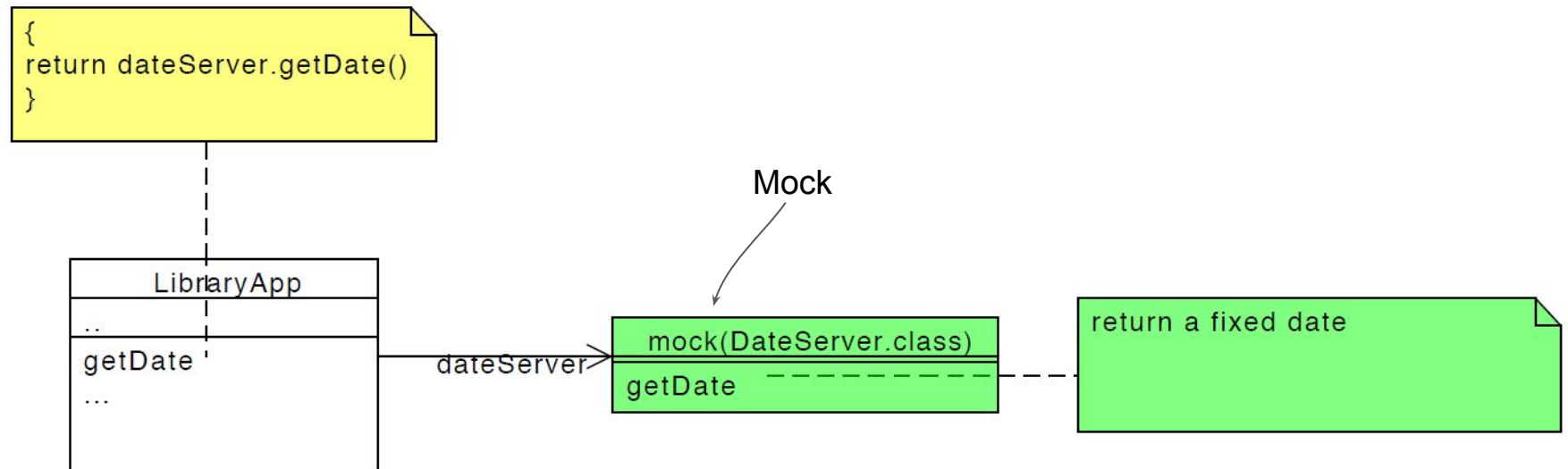


- ▶ How do we jump into the future?
 - Replace the `GregorianCalendar` class by a *mock* object that returns fixed dates
 - ▶ Problem: Can't replace `GregorianCalendar` class

Create DataServer



Create Mock



How?

- ▶ Import helper methods

```
import static org.mockito.Mockito.*;
```

- ▶ Create a mock object on a certain class

```
SomeClass mockObj = mock(SomeClass.class)
```

- ▶ return a predefined value for `m1 (args)`

```
when(mockObj.m1 (args) ).thenReturn(someObj) ;
```

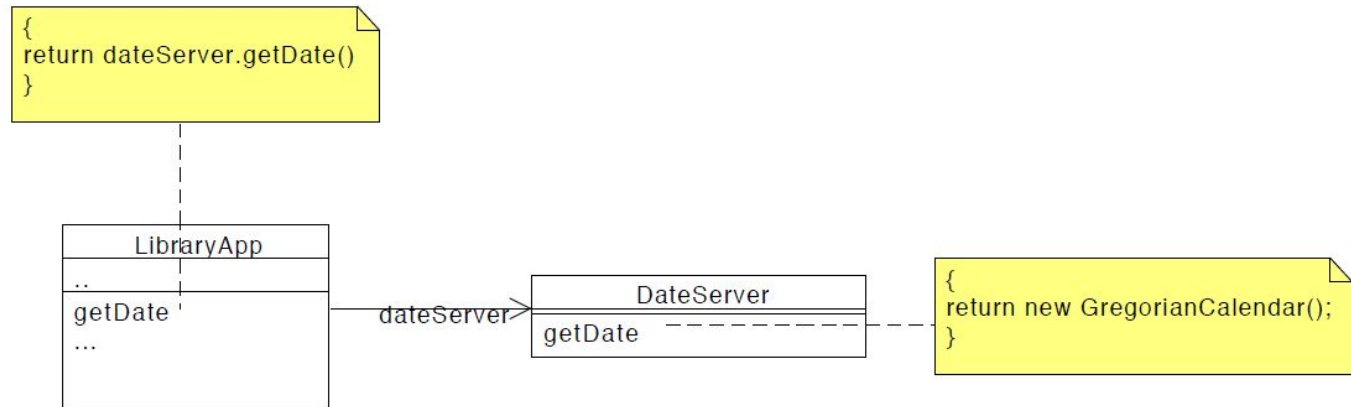
- ▶ **verify that message** `m2 (args)` **has been sent**

```
verify(mockObj) .m2 (args) ;
```


Mock Example 1: Overdue book

```
@Test
public void testOverdueBook() throws Exception {
    DateServer dateServer = mock(DateServer.class);
    libApp.setDateServer(dateServer);
    Calendar cal = new GregorianCalendar(2011, Calendar.JANUARY, 10);
    when(dateServer.getDate()).thenReturn(cal);
    ...
    user.borrowBook(book);
    newCal = new GregorianCalendar();
    newCal.setTime(cal.getTime());
    newCal.add(Calendar.DAY_OF_YEAR, MAX_DAYS_FOR_LOAN + 1);
    when(dateServer.getDate()).thenReturn(newCal);
    assertTrue(book.isOverdue());
}
```

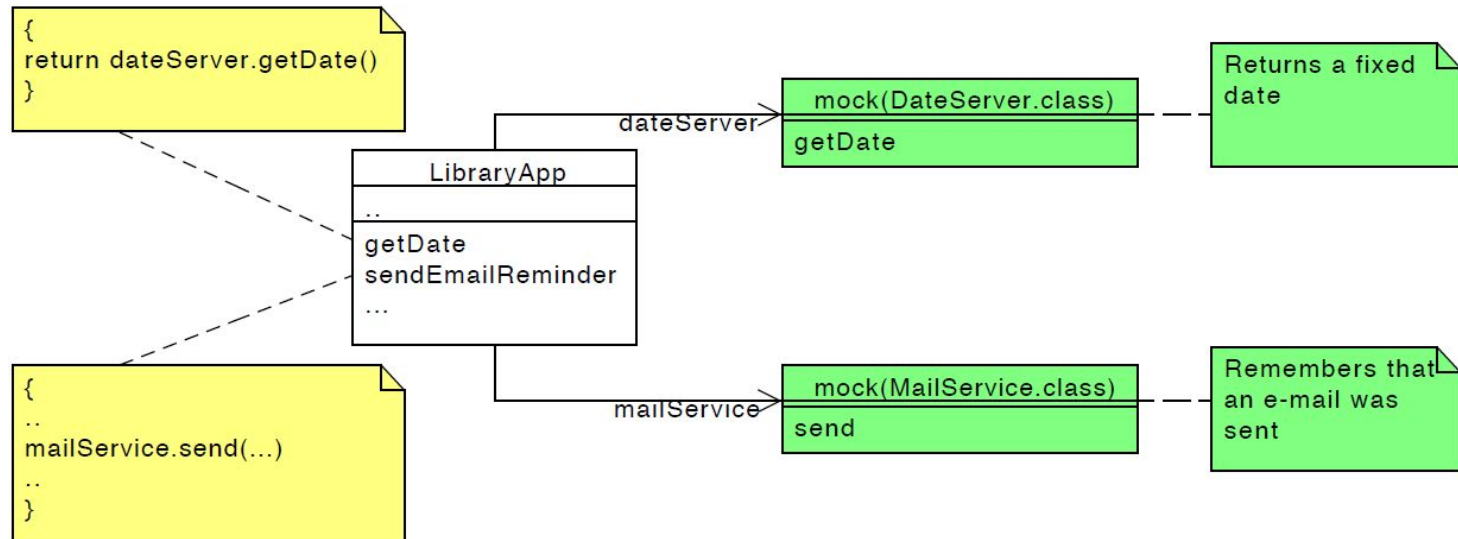
LibraryApp Code



```
public class LibraryApp {
    private DateServer ds = new DateServer();
    public setDateServer(DateServer ds) { this.ds = ds; }
    ...
}
```

```
public class DateServer {
    public Calendar getDate() {
        return new GreogorianCalendar();
    }
}
```

Testing E-mail



```
@Test
public void testEmailReminder() throws Exception {
    DateServer dateServer = mock(DateServer.class);
    libApp.setDateServer(dateServer);

    MailService mailService = mock(MailService.class);
    libApp.setMailService(mailService);
    ...
    libApp.sendEmailReminder();
    verify(mailService).send("..", "..", "..");
}
```

Verify

Check that no messages have been sent

```
verify(ms, never()).send(anyString(), anyString(), anyString());
```

Mockito documentation: <http://docs.mockito.googlecode.com/hg/org/mockito/Mockito.html>

Summary

Being Great at Testing

- testing + development are different
 - Developer: “I want this code to succeed.”
 - Tester: “I want this code to fail.”
 - Hybrid
- Test creatively
- Don't ignore the weird stuff



Fun

Profit