Software Engineering I (02161)

Week 4

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2017

Recap

- week 1: Introduction
- week 2: Requirements: Domain model, Use Cases
- week 3:
  - Requirements: User stories, from use cases to user stories
  - Test: Acceptance tests

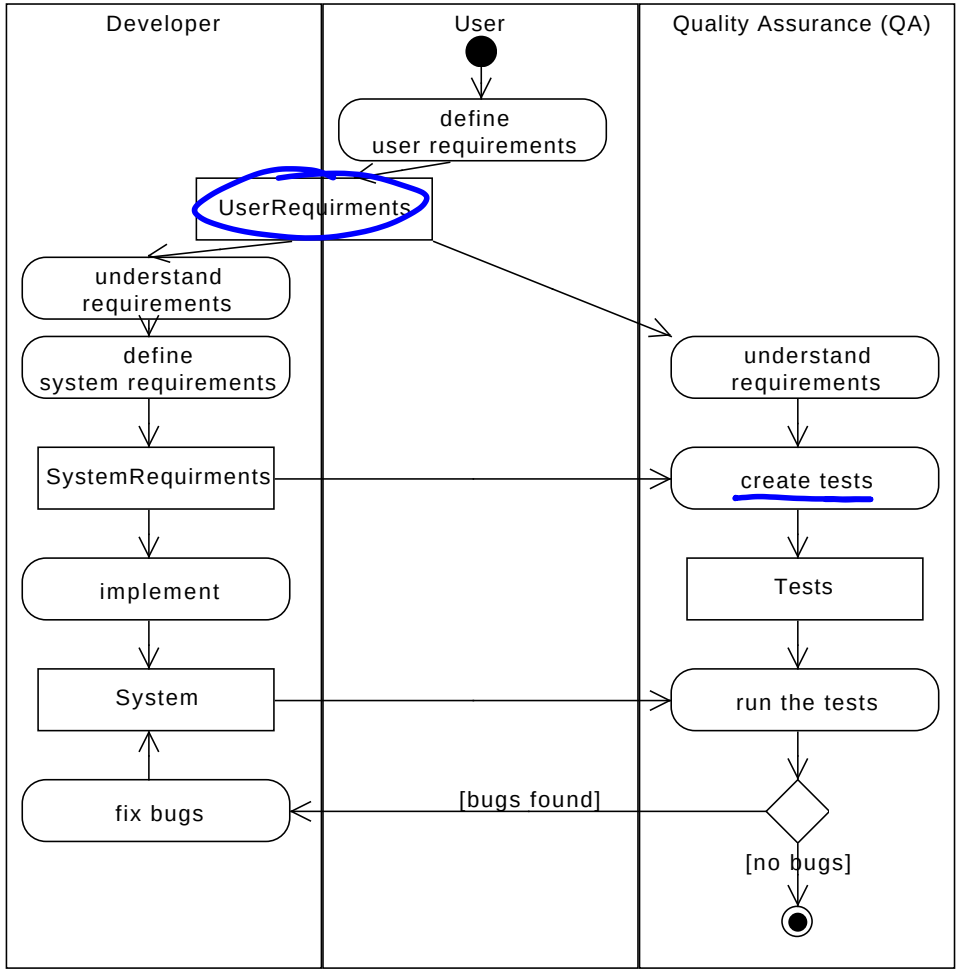
Contents

Test-Driven Development

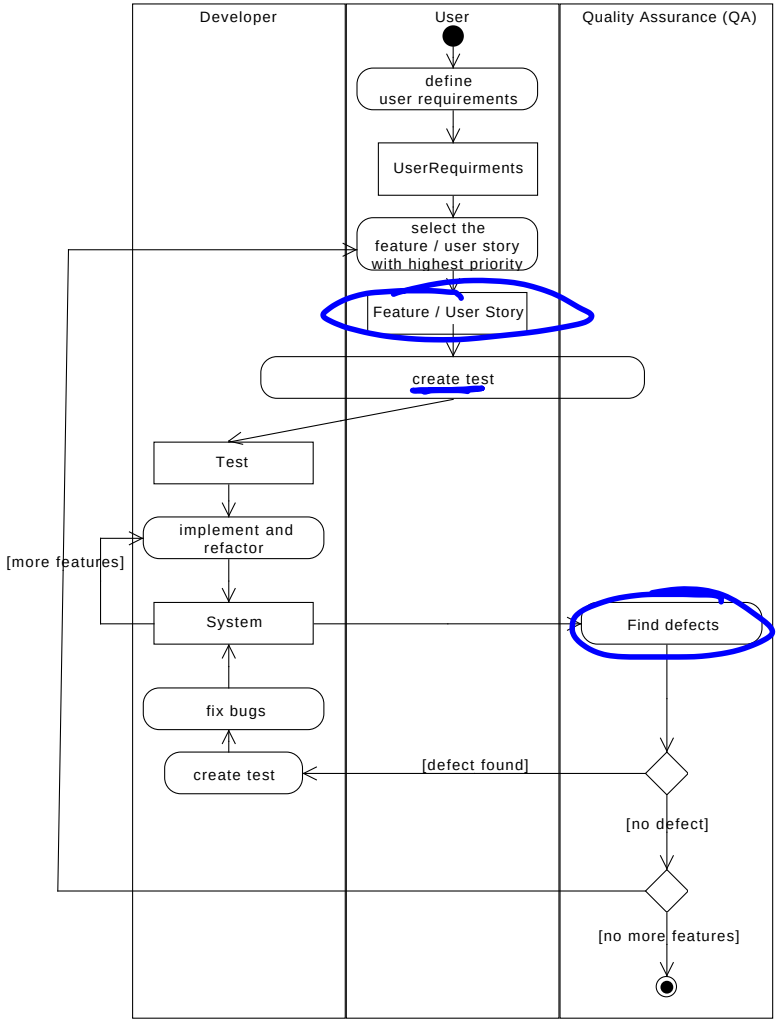
- Test *before* the implementation
- Tests = expectations on software
- All kind of tests: unit-, component-, system tests

Test-Driven Development

Traditional testing

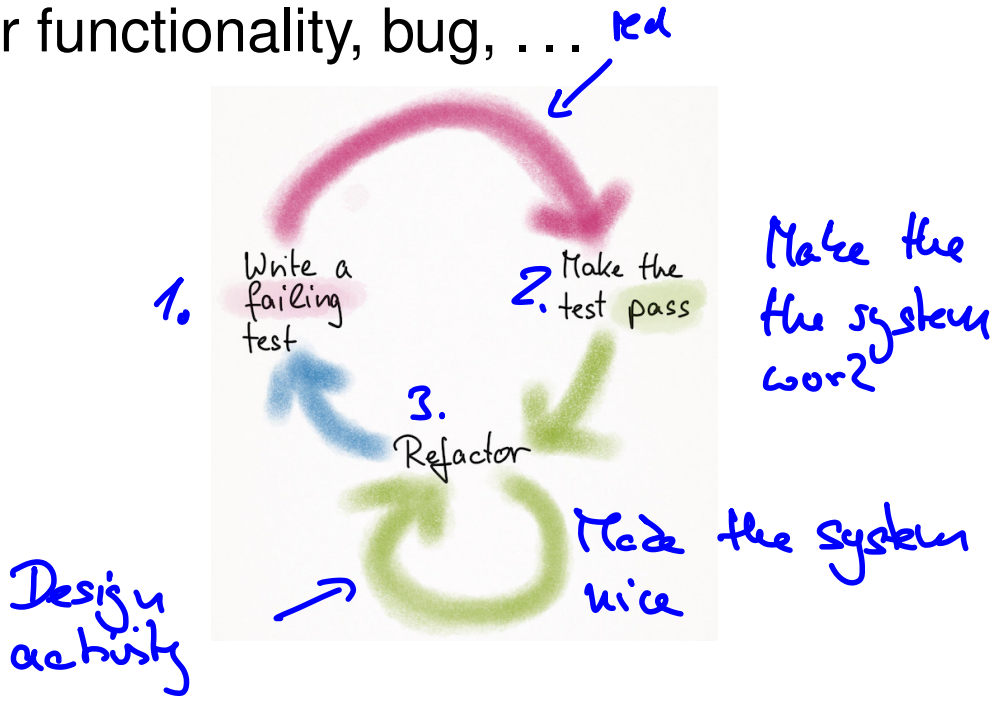


Test-Driven Development



TDD cycle

- Repeat for functionality, bug, . . .



- 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

1. Use case scenarios (missing functions): Acceptance tests
2. Possibility for defects (missing code): Defect tests
3. You want to write *more* code than is necessary to pass the test
4. Complex behaviour of classes: Unit tests
5. Code experiments: ”How does the system behave, if . . . ”

→ Make a list of new test ideas

TDD example: Borrow Book

- Use case

  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

- test data:
  - a user with CPR ”1234651234” and book with signature ”Som001”
- Test case
  - 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

Only code that is needed to make the test pass

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

Create a test for the alternative scenario

- test data:
  - a user with CPR ”1234651234”, book with signature ”Som001”, and 10 books with signatures ”book1”, . . . , ”book10”
- Test case
  - 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

- What happens if book == null in borrowBook?
- Test Case:
  - 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

Final implementation so far

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

Another example

- Creating a program to generate the n-th Fibonacci number

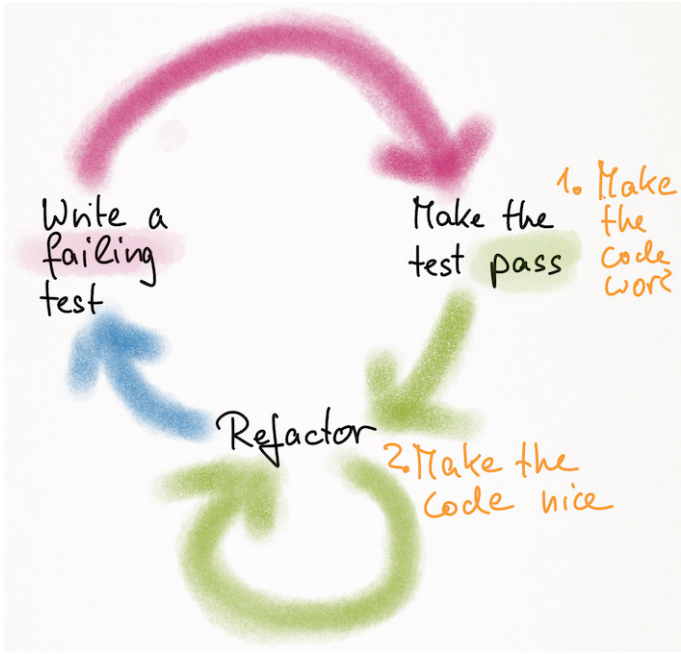
→ Codemanship's Test-driven Development in Java by Jason Gorman

`http://youtu.be/nt2KKUSSJsY`

- Note: The video uses JUnitMax to run JUnit tests automatically whenever the test files change (`junitmax.com`)
- A tool with similar functionality but free is Infinitest (`https://infinitest.github.io`)

Contents

Refactoring and TDD



Technical dept

- 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 test suite

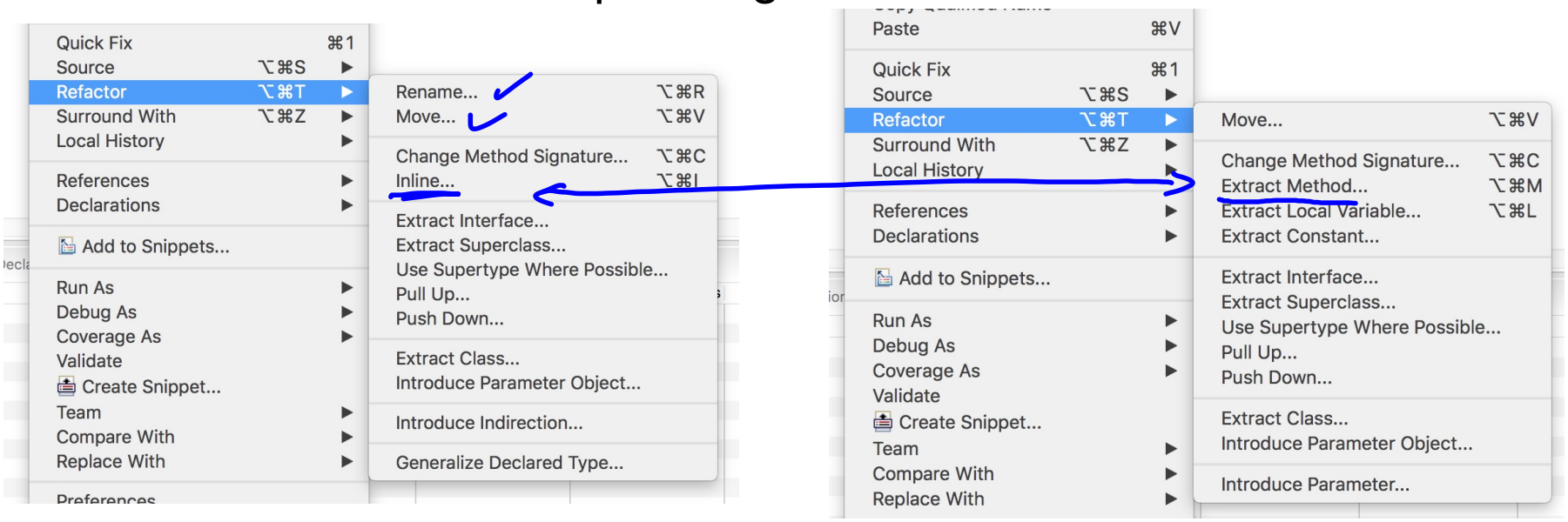
Refactoring

- PhD thesis by William Opdyke (1992): First tool: Smalltalk refactoring browser
- Book: *Refactoring: Improving the Design of Existing Code*, Martin Fowler, 1999
- Set of refactorings
  - e.g. **renameMethod**, **extractMethod**, **encapsulateField**, **encapsulateCollection**, ...
  - → complete list `http://www.refactoring.com/catalog/index.html`
- Set of code smells
  - e.g. Duplicate Code, Long Method, Large Class, ...
  - → `http://c2.com/cgi/wiki?CodeSmell`, or `http://www.codinghorror.com/blog/2006/05/code-smells.html`
  - How to write unmaintainable code `http://www2.imm.dtu.dk/courses/02161/2017/files/how_to_write_unmaintainable_code.pdf`
- Decompose large refactorings into several small refactorings
  - Each step: compiles and passes all tests

Eclipse Refactoring Support

Menu in the code editor

- Different menus depending on selection



Example of Code smells

If it stinks, change it Refactoring, Martin Fowler, 1999

- Duplicate Code
- Long Method
- Large Class (God class)
- Switch Statements
- Comments
- ...

`http://en.wikipedia.org/wiki/Code_smell`

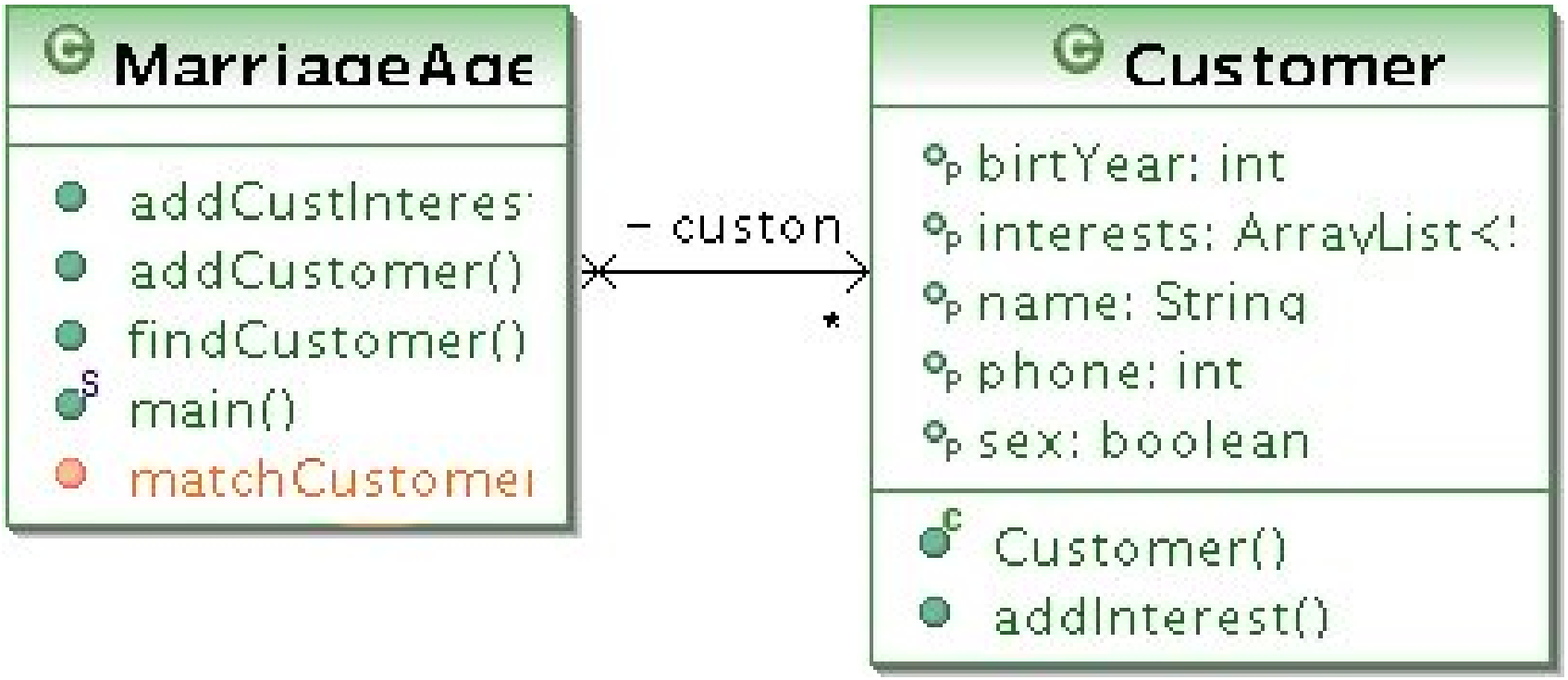
Code Smell: Switch Statement

```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Media m : borrowedMedia) {
         if (m.overdue) {
            switch (m.getType()) {
               case Media.BOOK : fine = fine + 10; break;
               case Media.DVD: fine = fine + 30; break;
               case Media.CD: fine = fine + 20; break;
               default fine = fine + 5; break;
            }
         }
      }
      return fine;
   }
}
```

Better Design

```
public class User {
   public double computeFine() {
      double fine = 0;
      for (Media m : borrowedMedia) {
         if (m.overdue) { fine = fine + m.getFine();}
      }
      return fine;
   }
}

abstract public class Media {
   abstract public double getFine();
}

public class Book extends Media {
   public double getFine() { return 10; }
}

public class DVD extends Media {
   public double getFine() { return 30; }
}

public class CD extends Media {
   public double getFine() { return 20; }
}
```

MarriageAgency class diagram



- Refactoring example in detail
  - → `http://www2.imm.dtu.dk/courses/02161/2017/slides/refactoring_example.pdf`
- Framework for running tests as soon as the code changes:
  - → Infinitest `http://infinitest.github.io/`

Remark on refactoring

- A refactoring takes a system with green tests to a system with green tests
- Decompose a large refactoring into small refactorings
  - $\rightarrow$ Don't have failing tests (or a broken system) for too long (e.g. days, weeks, ...)
    - Each small refactoring goes from a green test to a green test
    - Ideally, you can interrupt large refactorings to add some functionality and then continue with the refactoring

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

  $\rightarrow$ Testable system

Contents

How to use Date and calendar (I)

- Date class deprecated
- Calendar and GregorianCalendar classes
- An instance of Calendar is created by

```
new GregorianCalendar() // current date and time
new GregorianCalendar(2017, Calendar.JANUARY,10)
```

- Note that the month is 0 based (and not 1 based). Thus 1 = February.
- Best is to use the constants offered by Calendar, i.e. Calendar.JANUARY

How to use Date and calendar (I)

- One can assign a new calendar with the date of another calendar by

```
Calendar newCal = new GregorianCalendar();
newCal.setTime(oldCal.getTime())
```

- One can add years, months, days to a Calendar by using add: e.g.

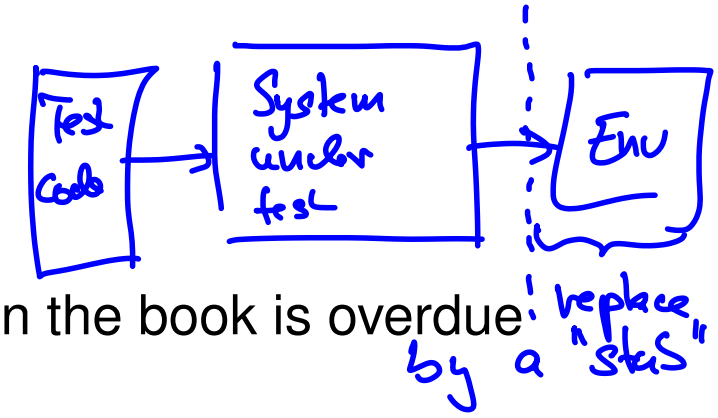
```
cal.add(Calendar.DAY_OF_YEAR,28)
```

- Note that the system roles over to the new year if the date is, e.g. 24.12.2017
- One can compare two dates represented as calendars using before and after, e.g.

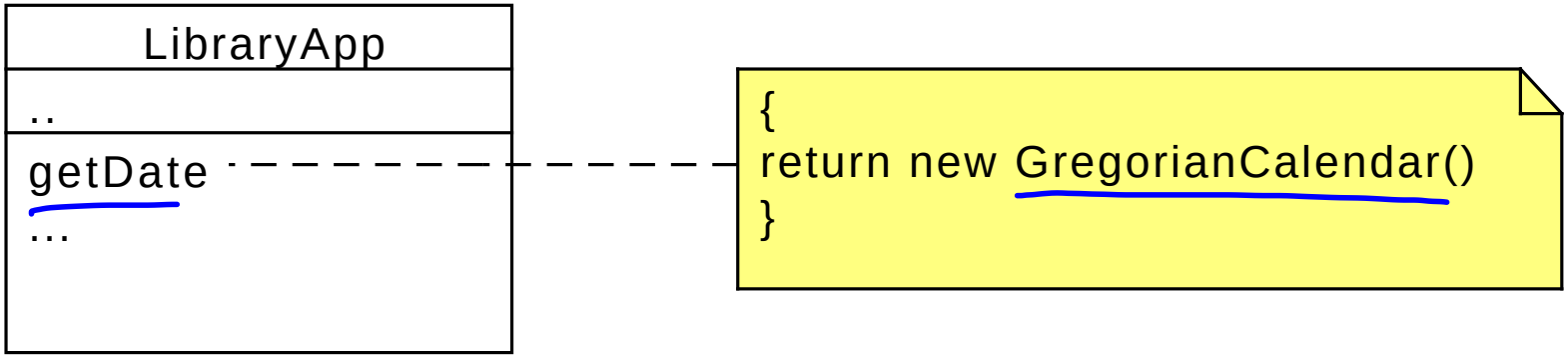
```
currentDate.after(dueDate)
```

Contents

Problems



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

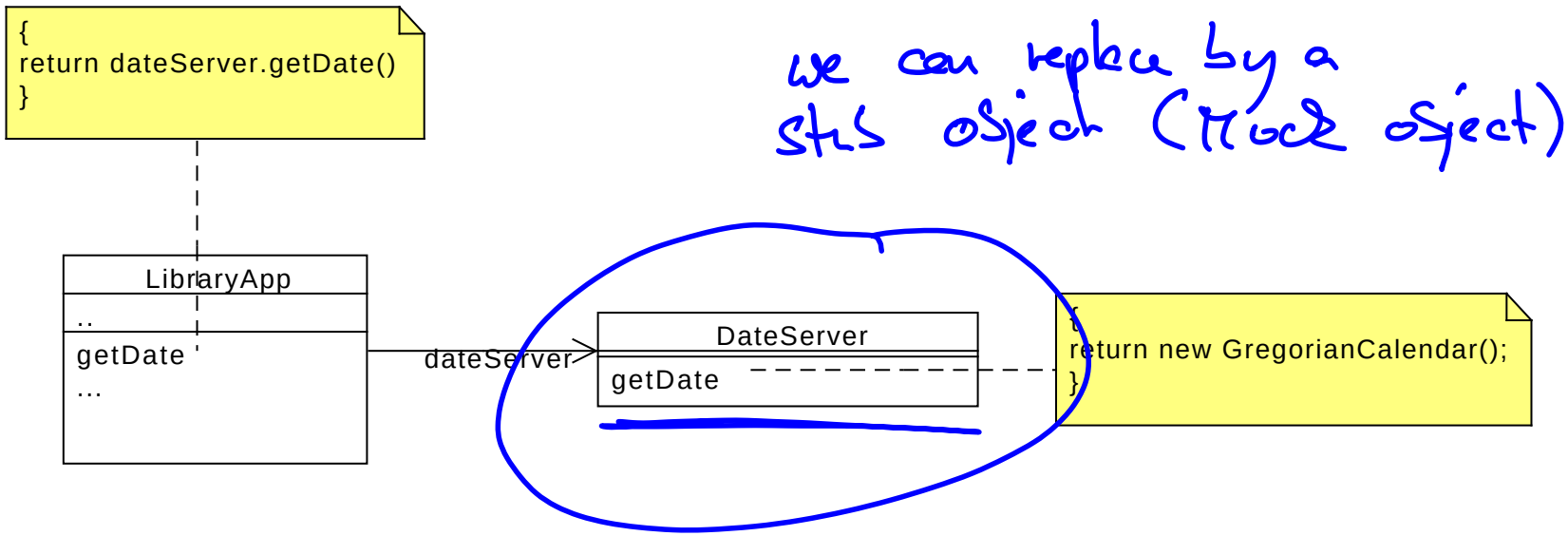


- How do we jump into the future?

→ Replace the GregorianCalendar class by a *mock* object that returns fixed dates

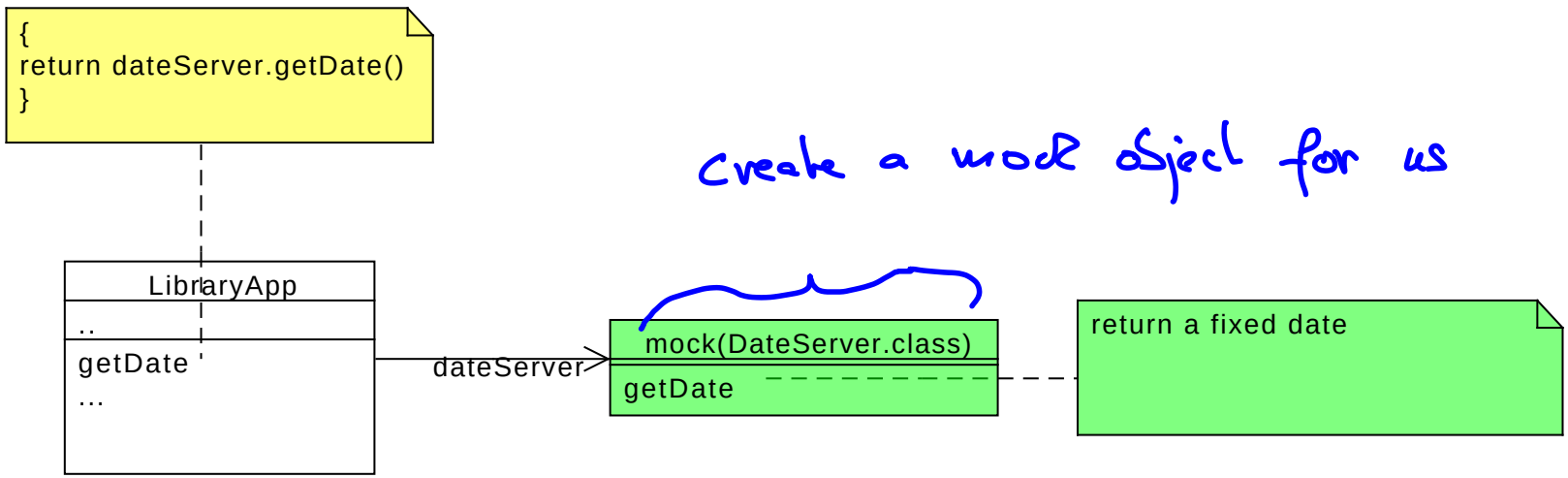
- Problem: Can't replace GregorianCalendar class

Creating a DateServer class



Creating a DateServer class

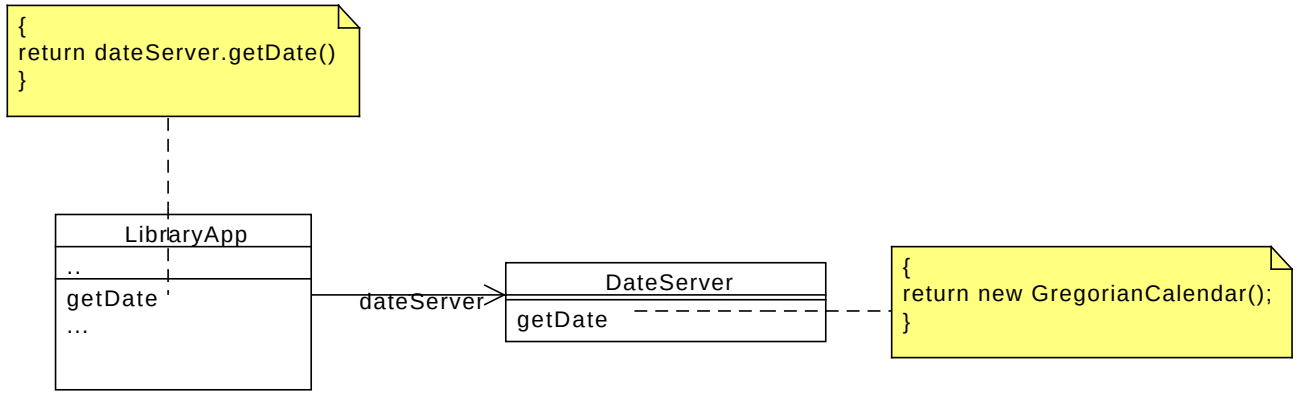
- The DateServer can be mocked



Mock Example: Overdue book

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

    Calendar cal = new GregorianCalendar(2017,Calendar.JANUARY,10);

    when(dateServer.getDate()).thenReturn(cal);

    libApp.setDateServer(dateServer);
    ...
    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

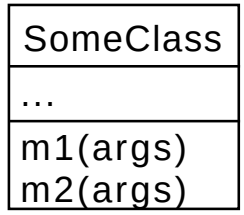



```
public class LibraryApp {
   private DateServer ds = new DateServer();
   public setDateServer(DateServer ds) { this.ds = ds;}
   ...
}

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

How to use

- Import helper methods

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



- Create a mock object on class SomeClass

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

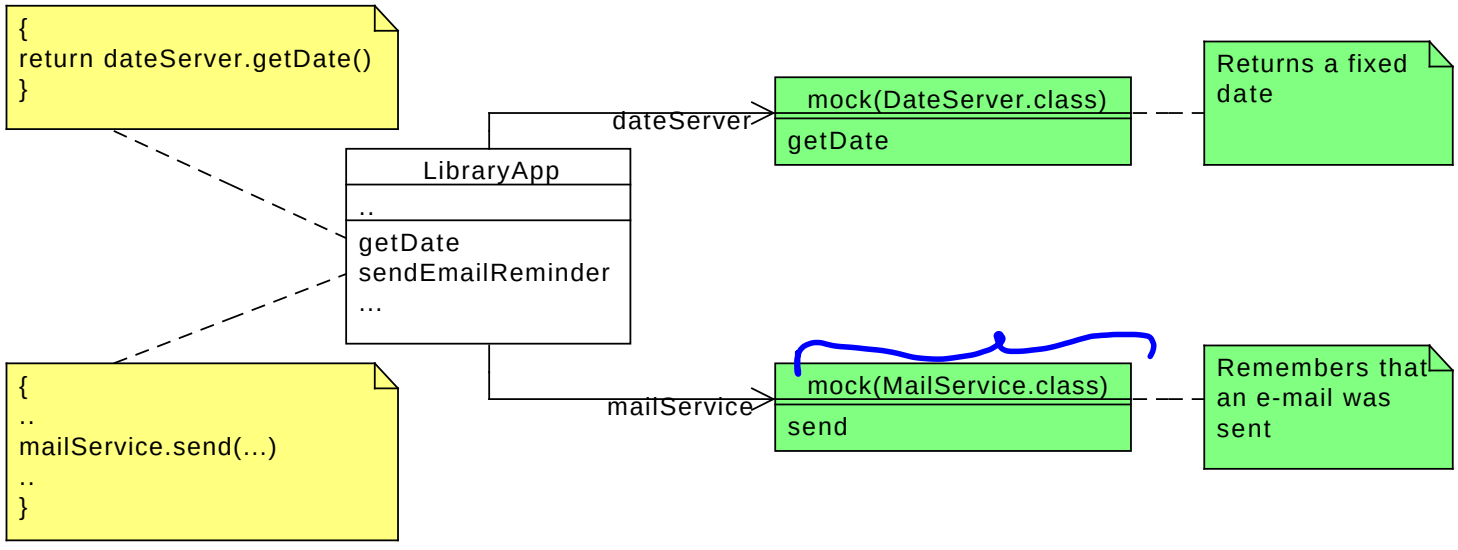
- return a predefined value for `m1(args)`

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

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

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

Testing for e-mails




```
@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("..","..","..");
}
```

appropriate arguments

Verify

Check that no messages have been sent

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

- Mokito homepage `http://mockito.org`
- Mockito tutorial `http://www.vogella.com/tutorials/Mockito/article.html`
- Look at the course Web page on how to use Mockito in your own Eclipse projects

Contents

Summary

- This week: Test-driven Development, Refactoring, Mock objects
- Next week: Continue with testing
  - Systematic tests: black box tests / white box tests
  - Code coverage