Objectives

- Course Notes
  - First Sprint Increment Due Friday
  - Notes on connecting the database
  - Homework 2
- Refactoring

Notes

- This Friday is the end of the first Sprint and each team needs to make a brief demonstration of their prototypes during the lab sessions
  - Aim for 8 minutes
  - We have a schedule of times that you should sign up for (due by 5:00 pm on 17/10)
  - I have also posted the rubric that we will use to evaluate your prototypes (scoring on the rubric is at the sole discretion of the evaluator)
- Homework 2, code of ethics writing assignment

Configuring a DB in a Container

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
<Resource name="jdbc/DistDB"
          auth="Container"
          type="javax.sql.DataSource"
          maxActive="100"
          maxIdle="30"
          maxWait="10000"
          username="username"
          password="password"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://10.10.3.14:3306/testdatabase361"/>
```

- Need to inform our server or servlet container where our database is
- In Tomcat we can specify this information as a JNDI resource in the `context.xml` file
- Use classes in the `java.sql` and `javax.sql` packages

Container Builds

- Software teams increasingly use container technology, like Docker, to package their applications
- Devs must be able to provision the container environment when working locally
- The packaging process results in the creation of a container image
- Tests have to interact with the container runtime environment
- Deployment environments may involve another layer of abstraction for the orchestration of containers (e.g. Kubernetes)

Containers[2]

- Software containers provide a sandboxed environment for running processes
- Only virtualize the operating system
- Containers can be quicker to spin up compared to virtual machines
- Docker community edition[1]
- Container images available for many common applications for example Tomcat and MySQL

---

[1] `https://docs.docker.com/install/`

[2] pictured: Vasco de Gama

Connections


```
private static Connection getDatabaseConnection() {
  Connection conn = null;
  try {
    Context initCtx = new InitialContext();
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    DataSource ds = (DataSource) envCtx.lookup("jdbc/DistDB");
    conn = ds.getConnection();
    } catch (Exception e) {
      e.printStackTrace();
    }
    return conn;
  }
```

- Getting the database connection in Java

Statements


```
try {
  Statement statement = conn.createStatement();
  ResultSet resultSet = statement.executeQuery(query);
  // do something with result

  conn.close();
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

- Making an SQL query from Java
- In this case the query is likely be some kind of selection
- The `Statement` class has different methods to make different kinds of queries
- Even better to work with the `PreparedStatement` class

JDBC Dependency

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.4</version>
</dependency>
```

- How to add the Java Database Connector with `pom.xml`

IT in Popular Culture[3]

- Media offers an important perspective on software and tech.
- Some milestones of Programmers/Devs in Media
  - *True Names* (1981)
  - *Wargames* (1983)
  - *Jurassic Park* (1993)
  - *The Matrix* (1999)
  - *The Social Network* (2010)

[3] pictured: Dennis Nedry in *Jurasic Park* (1993)

Homework 2

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Find an example of when a programmer (or computer scientist or IT specialist) has been portrayed in popular culture, for example in a movie or television series. Write a structured 5-paragraph essay reflecting on how the character was or was not a *professional* according to ACM/IEEE Code of Ethics and Professional Conduct.
- Each paragraph should be between 100 and 200 words
- **Paragraph 1**: introduce your example and explain how the character is related to computer science or IT
- **Paragraphs 2-4**: in each paragraph choose one of the ACM Code principles (for example, PRODUCT) and explain how the character did or did not uphold this particular principle
- **Paragraph 5**: explain why you would or would not want to be asssociated with the character you chose to analyze

Code of Ethics in Software Engineering

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- A joint task force of the ACM and IEEE created a guide to ethics consisting of 8 principles
- Think of the code as a *tool* rather than a *proscription*

ACM/IEEE Code of Ethics (Paraphrased)[4]

1. PUBLIC act in the public interest
2. CLIENT AND EMPLOYER act in the best interests of the client
3. PRODUCT maintain high quality standards in software products
4. JUDGEMENT integrity in professional judgments
5. MANAGEMENT ethical approach to management of software projects
6. PROFESSION maintain reputation of the profession
7. COLLEAGUES treat your peers and colleagues with respect
8. SELF continual learning over the lifetime in the profession

[4] D. Gotterbarn, K. Miller, and S. Rogerson, "Software engineering code of ethics," *Communications of the ACM*, vol. 40, no. 11, pp. 110–118, 1997.

Refactoring?

- *Refactoring* is discussed in Ch. 21 of the text [2]
- The idea has also been pioneered and promoted by Martin Fowler [3], defines both the noun and verb forms of the word in his book

Refactoring n. (Fowler)

a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

Refactoring v. (Fowler)

to restructure software by applying a series of refactorings without changing its observable behavior

Why Refactor

- Improve the design of software
  - Software is a living thing, can decay with age, especially when changes are made to achieve short-term goals (Technical Debt)
- Improves comprehension of code
  - When your focus is on making the code work, you are not necessarily considering the future developer
- Spot Bugs and Code Faster
  - Refactoring entails a deep engagement with the functionality that is embedded back into the code base
  - Prevent the situation where you are working on a patch of a patch of a patch...

Goals of Refactoring

- According to Larman [2] the goals of refactoring are the goals of good programming
  - remove duplicate code (don't repeat yourself)
  - improve clarity
  - make long methods shorter
  - remove uses of hard-coded literal constants

Code Smells [3] (scents)

- Code scents are inspirations for when to refactor (if you notice something, maybe it's a clue that something is wrong)
- *Mysterious name*: "naming is one of the two hard things in programming" (Karlton), trouble coming up with names could be a sign of deeper problems with the design
- *Duplicated Code*: need to read the duplicates carefully to check for differences, changes need to be made consistently to all duplicates
- *Long Method/Function*: prefer small functions to long functions; long functions are more difficult to understand; limited overhead to function calls

Code Smells [3] cont (scents)

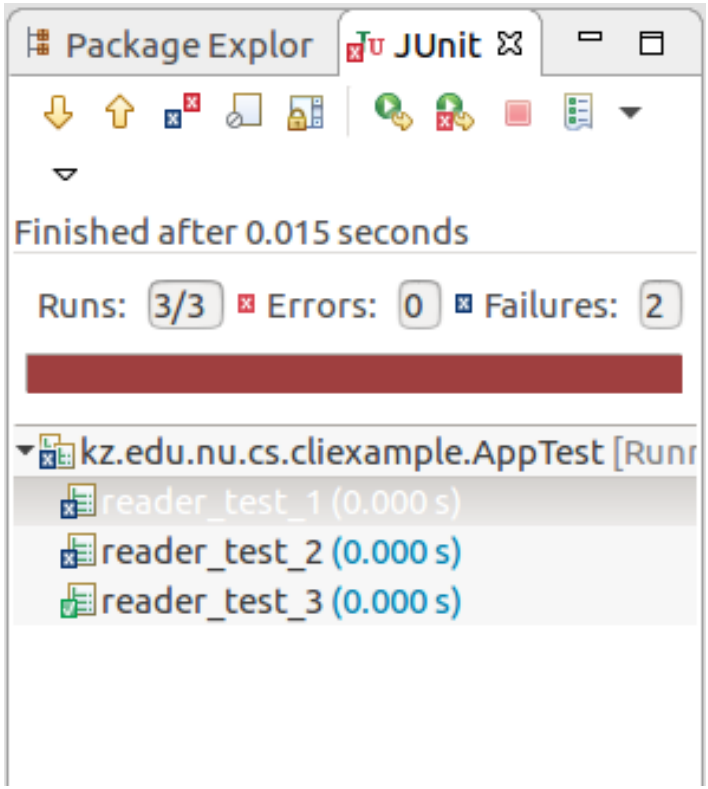
- *Long Parameter Lists*: better alternative to global data but can also be confusing; replace a parameter with a query or use an "options" object
- *Global Data/Mutable Data*: can be modified from anywhere in the codebase leading to hard to understand bugs; try to limit scope as much as possible
- *Divergent Change*: modifying a module in different ways for different reasons; indicates potential failure to follow the single-responsibility principle

Code Smells [3] cont (scents)

- *Feature Envy*: function in one module spends more time interacting with fields and methods in another module
- *Data Clumps*: data items that tend to be seen together; possibly refactor into an object
- *Primitive Obsession*: reluctance to create own fundamental types; for example, treating telephone numbers as strings rather than defining a type
- *Comments**: "When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous." [3]

Refactoring and TDD

- Aggressive refactoring and test-driven development are key practices of eXtreme Programming (XP)
- Clearly, TDD supports refactoring
- Robust test suite will make it more likely that we catch errors that might be introduced during refactoring, prevent a *regression*
- Tension with other principles of software development such as the Open-Closed Principle



Catalog of Refactorings

- In their book [3] Fowler and Beck develop a catalog of different refactorings
- Similar, in spirit, to the GoF book [4]
  - Name
  - Sketch (picture)
  - Motivation
  - Mechanics: step-by-step description
  - Examples
- Larman [2] also provides a short table of common refactorings
- Go through common refactorings in pairs that are mutually inverse
- Some examples are in Java some are in Javascript

Extract Method Example


```
public class Player
{ private Piece  piece;
  private Board  board;
  private Die[]  dice;
  public void takeTurn()
  {
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++) {
      dice[i].roll();
      rollTotal += dice[i].getFaceValue();
    }
    Square newLoc = board.getSquare(piece.getLocation(),
                                    rollTotal);
    piece.setLocation(newLoc);
}} // end of class
```

- This is the original code for the *Extract Method* refactoring from the text [2]

Extract Method Example


```
public void takeTurn() {
        // the refactored helper method
    int rollTotal = rollDice();

    Square newLoc = board.getSquare(piece.getLocation(), rollTotal)
    piece.setLocation(newLoc);
}
```

- *Extract Method*, we shorten a long method by introducing helper methods
- The same method `takeTurn()` with the loop factored out
- Pay attention to what is in scope, and what the code you want to extract depends on

Extract Method with Local Variables [3]


```
function printOwing(invoice) {
  let outstanding = 0;
  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30)
  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

- More complicated *Extract Method* example in Javascript

Extract Method with Local Variables [3]


```
function printDetails(invoice, outstanding) {
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

- Extracting printing the details
- This is easy because we are only reading the variables; Extract Method will be trickier if we are also updating

Inline Function [3]


```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5;
}
// becomes
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

- Inverse of *Extract Method*
- Replace a function call with an inlined block of code

Extract Variable [2]


```
boolean isLeapYear( int year )
{
return( ( ( year % 400 ) == 0 ) ||
      ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
// becomes
boolean isLeapYear( int year )
{
   boolean isFourthYear = ( ( year % 4 ) == 0 );
   boolean isHundrethYear = ( ( year % 100 ) == 0);
   boolean is4HundrethYear = ( ( year % 400 ) == 0);
   return (
      is4HundrethYear
      || ( isFourthYear && ! isHundrethYear ) );
}
```

- *Introduce Explaining Variable*: put the result of an expression into a temporary variable

Change Declaration [3]


```
function circum(radius) {
  return 2 * Math.PI * radius;
}
// becomes
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

- *Change Function Declaration*: change the name or signature of a function or method
- Simple Mechanics
  - Verify that removed parameters aren't referenced in function body
  - Change to desired declaration
  - Update all references to old declaration
  - Test

Change Declaration [3] cont.


```
function inNewEngland(aCustomer) {
  return ["MA", "CT", "ME"].includes(aCustomer.address.state);
}
// caller
const newEnglanders = someCustomers.filter(c => inNewEngland(c));
// becomes
function inNewEngland(stateCode) {
  return ["MA", "CT", "ME"].includes(stateCode);
}
```

- The example above shows how a function is refactored so that that one of the properties of the old argument becomes the new argument
- Obviously, this requires a sequence of transformations involving both the function and callers

Extract Class [3]

- *Extract Class*: one class is doing the job of two classes
- Mechanics
  - Decide how to split responsibilities
  - Create a new class to express split-off responsibilities
  - *Move Field* and *Move Method* on all things that should change
  - Decide on visibility of the class
- Example: Person class with many fields and methods related to the personal telephone number
  - Refactor Person into Person and Telephone Number classes

Inline Class [3]

- *Inline Class*: a class is superfluous or not doing much. We have a *source* class and an *absorbing* class
  - Public methods of source class $\rightarrow$ absorbing class
  - Update all references
  - Use *Move Field* and *Move Method* on all elements of the source class until finished
- Example: Opposite of the previous example
  - Refactor Person and Telephone Number classes into the Person class

Hide Delegate [3]


```
Person john;

//

manager = john.getDepartment().getManager();
```

- *Hide Delegate*: clients are calling a method on the delegate class of some other object. In the example above [3] the client code is accessing the manager through the delegate object.
- Mechanics
  - Update the main class with the interface of the delegate
  - Update references in the clients
  - Make delegate private if no other code is accessing it

Conclusion

- Refactoring is a way to make improvements to your for clarity, ease of change without altering the behavior
  - May not be necessary if you can be sure that the software will never need to be updated
- Refactorings cataloged similarly to design patterns
- IDEs support common refactorings
- Automated testing is necessary to maintain invariants
- *Resource*: Catalog[5] of refactorings maintained by Martin Fowler

[5] `https://refactoring.com/catalog/`

References I

[1] D. Gotterbarn, K. Miller, and S. Rogerson, "Software engineering code of ethics," *Communications of the ACM*, vol. 40, no. 11, pp. 110–118, 1997.

[2] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[4] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.