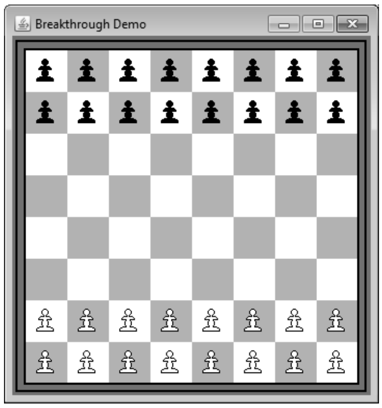
Exam Question Examples 2010

Henrik Bærbak Christensen

December 9, 2010

0.1 Test-driven development.

The Breakthrough game is played on a standard chess board, using 16 white and 16 black pawns that are initially arranged like in the figure on the right.



The rules of movement are simple. White player begins. A piece may move one square straight or diagonally forward if that square is empty. A piece, however, may only capture an opponent piece diagonally. When capturing, the opponent piece is removed from the board and the player's piece takes its position, as you do in chess.

Using a TDD process, the methods covering basic board and piece storage and turn handling have already been been developed in a class implementing the Breakthrough interface:

```java
public interface Breakthrough {
  /** Enumeration of the three types of 'pieces' that
      is possible on a given location on the chess board:
      black, white, or no piece */
  public static enum PieceType { BLACK, WHITE, NONE};
  /** Enumeration of the two types of players in the game,
      either white or black */
  public static enum PlayerType { BLACK, WHITE };

  /** Return the type of piece on a given (row,column) on
      the chess board.
      @return the type of piece on the location.*/
  public PieceType getPieceAt( int row, int column );

  /** Return the player that is in turn, i.e. allowed
      to move.
      @return the player that may move a piece next */
  public PlayerType getPlayerInTurn();

  /** Validate a move from a given location (fromRow, fromColumn) to a
      new location (toRow, toColumn). A move is invalid if you try to
      move your opponent's pieces or the move does not follow the
      rules, see the exercise specification.  PRECONDITION: the
      (row,column) coordinates are valid posititions, that is, all
      between (0..7).
      @return true if the move is valid, false otherwise */
  public boolean isMoveValid(int fromRow, int fromColumn,
                             int toRow, int toColumn);

}
```

You are asked to start implementing the isMoveValid method using TDD. You can assume method getPlayerInTurn() and getPieceAt(row,column) are correctly implemented.

You are asked to describe a preliminary plan for a test-driven development effort. You should use terminology, techniques, and tools from the course to:

- Sketch a test list, and outline some plausible initial iterations.
- Cover steps and TDD principles in one or two initial interations, time permitting, including central Java code fragments.
- Broaden the discussion to include basic definitions, terminology, and techniques in the area.
- Relate to other topics in the course.

Do not try to cover all requirements if this removes the possibility of broadening the discussion.

0.2 Test-driven development.

Consider the following specification:

```java
public interface FanControl {
  /** Return the frequency of the cooling fan given the
      temperature of air and liquid in a chemical chamber.
      The ideal liquid temperature is around 75 degrees.

      The frequency (return value) is calculated as follows
      (in order of precedence):

      if TempLiquid > 90 return 9999 (ALERT)
      if TempLiquid > 80 return 500 (Max cooling)
      if TempLiquid < 70 return 0 (No cooling)
      otherwise return (TempLiquid-70)*50

      The air temperature may overrule the above calculation:
      if TempAir > 100 return 9999
      if TempAir > 90 return 500
   */
  public int fanControl(double TempAir, double TempLiquid);
}
```

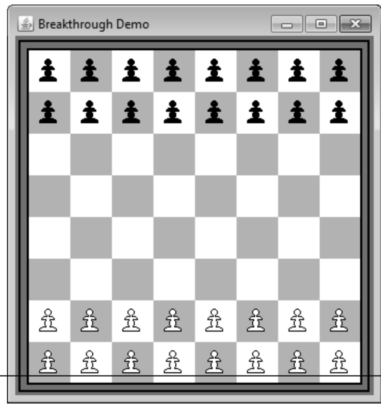
You are asked to describe a preliminary plan for a test-driven development effort. You should use terminology, techniques, and tools from the course to:

- Sketch a test list, and outline some plausible initial iterations.
- Cover steps and TDD principles in one or two initial interations, time permitting, including central Java code fragments.
- Broaden the discussion to include basic definitions, terminology, and techniques in the area.
- Relate to other topics in the course.

Do not try to cover all requirements if this removes the possibility of broadening the discussion.

0.3 Systematic black-box testing.

The Breakthrough game is played on a standard chess board, using 16 white and 16 black pawns that are initially arranged like in the figure on the right.



The rules of movement are simple. White player begins. A piece may move one square straight or diagonally forward if that square is empty. A piece, however, may only capture an opponent piece diagonally. When capturing, the opponent piece is removed from the board and the player's piece takes its position, as you do in chess.

The interface of a FACADE for the game is shown below.

```
public interface Breakthrough {
  /** Enumeration of the three types of 'pieces' that
      is possible on a given location on the chess board:
      black, white, or no piece */
  public static enum PieceType { BLACK, WHITE, NONE};
  /** Enumeration of the two types of players in the game,
      either white or black */
  public static enum PlayerType { BLACK, WHITE };

  /** Return the type of piece on a given (row,column) on
      the chess board.
      @return the type of piece on the location.*/
  public PieceType getPieceAt( int row, int column );

  /** Return the player that is in turn, i.e. allowed
      to move.
      @return the player that may move a piece next */
  public PlayerType getPlayerInTurn();

  /** Validate a move from a given location (fromRow, fromColumn) to a
      new location (toRow, toColumn). A move is invalid if you try to
      move your opponent's pieces or the move does not follow the
      rules, see the exercise specification. PRECONDITION: the
      (row,column) coordinates are valid posititions, that is, all
      between (0..7).
      @return true if the move is valid, false otherwise */
  public boolean isMoveValid(int fromRow, int fromColumn,
                             int toRow, int toColumn);

}
```

You are asked to develop a set of test cases using the equivalence class technique of method isMoveValid.

You are asked use terminology, techniques, and tools from the course to:

- Outline the conditions in the specification and derive an equivalence class table and generate test cases for a systematic black-box testing.
- Argue for your equivalence classes and choices.
- Broaden the discussion to include basic definitions, terminology, and techniques in the area.
- Relate to other topics in the course.

Do not try to cover all requirements if this removes the possibility of broadening the discussion.

0.4 Systematic black-box testing.

Consider the following specification:

```java
public interface FanControl {
  /** Return the frequency of the cooling fan given the
      temperature of air and liquid in a chemical chamber.
      The ideal liquid temperature is around 75 degrees.

      The frequency (return value) is calculated as follows
      (in order of precedence):

      if TempLiquid > 90 return 9999 (ALERT)
      if TempLiquid > 80 return 500 (Max cooling)
      if TempLiquid < 70 return 0 (No cooling)
      otherwise return (TempLiquid-70)*50

      The air temperature may overrule the above calculation:
      if TempAir > 100 return 9999
      if TempAir > 90 return 500
   */
  public int fanControl(double TempAir, double TempLiquid);
}
```

You are asked use terminology, techniques, and tools from the course to:

- Outline the conditions in the specification and derive an equivalence class table and generate test cases for a systematic black-box testing.
- Argue for your equivalence classes and choices.
- Broaden the discussion to include basic definitions, terminology, and techniques in the area.
- Relate to other topics in the course.

Do not try to cover all requirements if this removes the possibility of broadening the discussion.

0.5 Variability management

The pilots on in-bound and out-bound flights from an airport need precise information about the wind on the runway. One important information is *the 10 minute mean wind* that describes speed, direction, and characteristics of the wind in the last 10 minutes. This information is coded in different types of "reports", MET REPORT, METAR, and SYNOP, that each format the information in their own way. Furthermore the algorithms to calculate the 10 minute mean from observed values vary from one country to another.

So far, we have sold a wind computation system to Denmark, France, and Germany, and have a design like the code shown below (Only METAR report variant is shown, and many data values are faked to reduce the code size.)

```java
public abstract class WindCalculator {
  public static void main(String[] args) {
    // Example: METAR Wind after Danish regulations.
    WindCalculator calculator = new METARWindCalculator();
    int[] values = new int[] {230,7,245,8,234,7}; // fake-it
    String METAR =
      calculator.calculateFormatted10MinWind( values,
                                              WindCalculator.DANISH );
  }
  /** calculate a formatted 10 minute mean string to insert into a
   * specific meterological report and calculated according to
   * national algorithms. */
  public String calculateFormatted10MinWind(int[] datavalues,
                                            int algorithmType ) {
    int meanSpeed = 7, meanDirection = 234; // fake-it
    boolean vrb = false;                    // fake-it
    switch ( algorithmType ) {
    case DANISH:
      /* calculate means speed, direction, and vrb condition according
         to Danish regulations (omitted) */
      break;
    case FRENCH: /* French algorithm (omitted) */ break;
    case GERMAN: /* German algorithm (omitted) */ break;
    }
    return format(meanSpeed,meanDirection,vrb);
  }
  public abstract String format(int s, int d, boolean vrb);

  /* constants defining which national calculation algorithm to use */
  public static final int DANISH = 100;
  public static final int FRENCH = 101;
  public static final int GERMAN = 102;
}
class METARWindCalculator extends WindCalculator {
  public String format(int s, int d, boolean vrb) {
    String result = "23407"; // fake-it
    return result;
  }
}
```

You are asked use terminology, techniques, and tools for designing for variability to:

- Analyze the code fragment with respect to benefits and liabilities.
- Classify the techniques used to handle variability.
- Present an alternative design that improves maintainability and flexibility; and sketch central refactorings in Java.
- Discuss concepts introduced at a theoretical level.
- Relate to other topics.

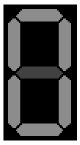
0.6 Test stubs and unit/integration testing

The hardware producer of a *seven segment LED display* provides a very low-level interface for turning on each of the seven LED (light-emitting diode) segments on or of by a Java interface:

```
public interface SevenSegment {
  /** turn a LED on or off.
   * @param led the number of the LED. Range is 0 to 6. The LEDs are
   * numbered top to bottom, left to right. That is, the top,
   * horizontal, LED is 0, the top left LED is 1, etc.
   * @param on if true the LED is turned on otherwise it is turned
   * off.
   */
  void setLED(int led, boolean on);
}
```

As an example, to display “0” as in the figure below, we would have to write:

```
d.setLED(0,true); d.setLED(1,true); d.setLED(2,true); d.setLED(3,false);
d.setLED(4,true); d.setLED(5,true); d.setLED(6,true);
```



Clearly, this is much too cumbersome in practice, so it is much better to define an abstraction that can turn on and off the proper LEDs for our ten numbers 0 to 9:

```
public interface NumberDisplay {
  /** display a number on a seven segment.
   * @param number the number to display.
   * Precondition: number should be in the range 0 to 9.
   */
  void display(int number);
}
```

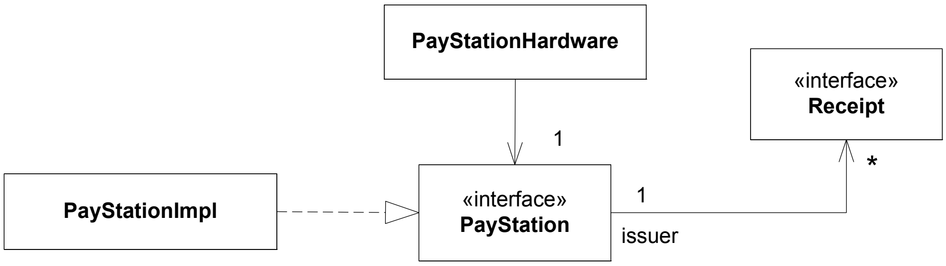
Thus, the code above would simply become: `d2.display(0);`

You are asked to use the terminology and techniques of test stubs to:

- Sketch a design that allows TDD and automated testing of the implementation of the NumberDisplay interface, using UML and Java.
- Discuss concepts introduced from a theoretical viewpoint.
- Classify and discuss the developed stub(s) according to the classification of Meszaros (Sidebar 12.1, page 192, in FRS).
- Relate to other topics.

0.7 Design patterns

Consider the the simplified design of the pay station, here described as a UML class diagram:



You are now faced with a new customer requirement: *In the four southern pay stations on our parking lot, the pay stations should not accept payment from 19:00 evening until 7:00 morning.* Rephrasing this, the addPayment method of interface PayStation (FRS p. 46) of these pay stations should throw an IllegalConException no matter what coinValue is entered.

You are asked to identify a design pattern that will solve this requirement such that a flexible, reliable, and maintainable design emerge.

- Describe the design using UML and Java and emphasize the design pattern identified.
- Discuss alternatives to the proposed design, and argue for benefits and liabilities.
- Discuss the design pattern concept from a theoretical point of view, including the various definitions.
- Relate to other topics.

0.8 Design patterns

An image/picture is basically just a matrix of pixels, each pixel having a luminosity value for the three basic colors RGB (red, green, blue). However, any image processing program needs to store images in many different formats: GIF, PNG, JPEG, TIFF, and many others. While the individual formats vary greatly, they generally have a similar structure with the binary file divided into "chunks" following each other:

- Header chunk: identifying the particular format, e.g. "GIF89a".
- Image property chunk: encodes data like (width, height, aspect ratio).
- Color table chunk: encodes pixel colors in a tabular format to reduce raw image size.
- Raw image data chunks: the actual values of the pixels of the image.
- Trailer chunk: identifies the end of the file.

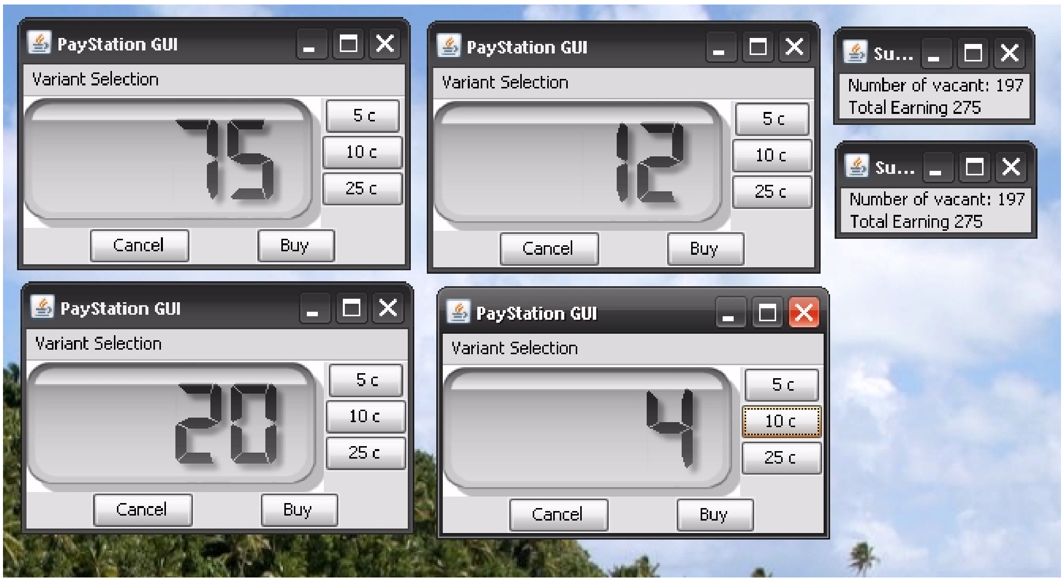
In our image processing application, we want a flexible and maintainable design that allows us to store images in a large range of image formats, and furthermore is "open for extension" to include future new image formats while staying "closed for modification"

You are asked to identify a design pattern that will solve this requirement such that a flexible, reliable, and maintainable design emerge.

- Describe the design using UML and Java and emphasize the design pattern identified.
- Discuss alternatives to the proposed design, and argue for benefits and liabilities.
- Discuss the design pattern concept from a theoretical point of view, including the various definitions.
- Relate to other topics.

0.9 Design patterns

Alphatown approaches us with a new requirement. They want to monitor the pay stations on a given parking lot for two purposes: A) they want a digital sign at the entrance stating the number of vacant slots for cars at the parking lot, and B) they want to monitor the total earning of the parking lot. Below is shown an early prototype of such a system where four pay stations are monitored by two "monitor" applications.



We realize that a monitor application can calculate the two properties (vacant slots and earning) if they are informed of "number of minutes bought" and "amount of cents entered" in every buy transaction from every pay station in the parking lot.

You are asked to identify a design pattern that will solve this requirement such that a flexible, reliable, and maintainable design emerge.

- Describe the design using UML and Java and emphasize the design pattern identified.
- Discuss alternatives to the proposed design, and argue for benefits and liabilities.
- Discuss the design pattern concept from a theoretical point of view, including the various definitions.
- Relate to other topics.

(You should focus on the exchange of information between pay stations and monitor applications, not on the algorithm to calculate number of vacant slot.)

0.10 Compositional Design

The pilots on in-bound and out-bound flights from an airport need precise information about the wind on the runway. One important information is *the 10 minute mean wind* that describes speed, direction, and characteristics of the wind in the last 10 minutes. This information is coded in different types of "reports", MET REPORT, METAR, and SYNOP, that each format the information in their own way. Furthermore the algorithms to calculate the 10 minute mean from observed values vary from one country to another.

So far, we have sold a wind computation system to Denmark, France, and Germany, and have a design like the code shown below (Only METAR report variant is shown, and many data values are faked to reduce the code size.)

```java
public abstract class WindCalculator {
  public static void main(String[] args) {
    // Example: METAR Wind after Danish regulations.
    WindCalculator calculator = new METARWindCalculator();
    int[] values = new int[] {230,7,245,8,234,7}; // fake-it
    String METAR =
      calculator.calculateFormatted10MinWind( values,
                                              WindCalculator.DANISH );
  }
  /** calculate a formatted 10 minute mean string to insert into a
   * specific meterological report and calculated according to
   * national algorithms. */
  public String calculateFormatted10MinWind(int[] datavalues,
                                            int algorithmType ) {
    int meanSpeed = 7, meanDirection = 234; // fake-it
    boolean vrb = false;                    // fake-it
    switch ( algorithmType ) {
    case DANISH:
      /* calculate means speed, direction, and vrb condition according
         to Danish regulations (omitted) */
      break;
    case FRENCH: /* French algorithm (omitted) */ break;
    case GERMAN: /* German algorithm (omitted) */ break;
    }
    return format(meanSpeed, meanDirection, vrb);
  }
  public abstract String format(int s, int d, boolean vrb);

  /* constants defining which national calculation algorithm to use */
  public static final int DANISH = 100;
  public static final int FRENCH = 101;
  public static final int GERMAN = 102;
}
class METARWindCalculator extends WindCalculator {
  public String format(int s, int d, boolean vrb) {
    String result = "23407"; // fake-it
    return result;
  }
}
```

You are asked to analyze the above design from a compositional design perspective:

- Describe the ③-①-②process and underlying principles, and apply it on the above system.
- Sketch UML and Java code for a refactored system.
- Relate the refactored design to multi-dimensional variance, and discuss benefits and liabilities of the original and refactored design.
- Relate to the concepts of behaviour, responsibility, roles, and protocol.
- Relate to other topics.

0.11 Frameworks

The pilots on in-bound and out-bound flights from an airport need precise information about the wind on the runway. One important information is *the 10 minute mean wind* that describes speed, direction, and characteristics of the wind in the last 10 minutes. This information is coded in different types of "reports", MET REPORT, METAR, and SYNOP, that each format the information in their own way. Furthermore the algorithms to calculate the 10 minute mean from observed values vary from one country to another.

So far, we have sold a wind computation system to Denmark, France, and Germany, and have a design like the code shown below (Only METAR report variant is shown, and many data values are faked to reduce the code size.)

```
public abstract class WindCalculator {
  public static void main(String[] args) {
    // Example: METAR Wind after Danish regulations.
    WindCalculator calculator = new METARWindCalculator();
    int[] values = new int[] {230,7,245,8,234,7}; // fake-it
    String METAR =
      calculator.calculateFormatted10MinWind( values,
                                              WindCalculator.DANISH );
  }
  /** calculate a formatted 10 minute mean string to insert into a
   * specific meterological report and calculated according to
   * national algorithms. */
  public String calculateFormatted10MinWind(int[] datavalues,
                                            int algorithmType ) {
    int meanSpeed = 7, meanDirection = 234; // fake-it
    boolean vrb = false;                    // fake-it
    switch ( algorithmType ) {
    case DANISH:
      /* calculate means speed, direction, and vrb condition according
         to Danish regulations (omitted) */
      break;
    case FRENCH: /* French algorithm (omitted) */ break;
    case GERMAN: /* German algorithm (omitted) */ break;
    }
    return format(meanSpeed,meanDirection,vrb);
  }
  public abstract String format(int s, int d, boolean vrb);

  /* constants defining which national calculation algorithm to use */
  public static final int DANISH = 100;
  public static final int FRENCH = 101;
  public static final int GERMAN = 102;
}
class METARWindCalculator extends WindCalculator {
  public String format(int s, int d, boolean vrb) {
    String result = "23407"; // fake-it
    return result;
  }
}
```

You are asked to use framework terminology, techniques, and tools to:

- Describe how the design could be refactored to become a framework.
- Sketch cental aspects in the refactored design using UML and Java.
- Discuss concepts introduced from a theoretical viewpoint.
- Relate to other topics.