

# Exam Question Examples 2018

Henrik Bærbak Christensen

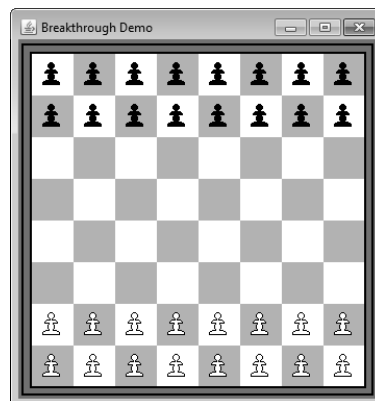
December 7, 2018

## 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 developed in a class implementing the Breakthrough interface:



```
1 public interface Breakthrough {
2     /** Enumeration of the three types of 'pieces' that
3      * is possible on a given location on the chess board:
4      * black, white, or no piece */
5     public static enum PieceType { BLACK, WHITE, NONE};
6     /** Enumeration of the two types of players in the game,
7      * either white or black */
8     public static enum PlayerType { BLACK, WHITE };
9
10    /** Return the type of piece on a given (row,column) on
11     * the chess board.
12     * @return the type of piece on the location.*/
13    public PieceType getPieceAt( int row, int column );
14
15    /** Return the player that is in turn, i.e. allowed
16     * to move.
17     * @return the player that may move a piece next */
18    public PlayerType getPlayerInTurn();
19
20    /** Validate a move from a given location (fromRow, fromColumn) to a
21     * new location (toRow, toColumn). A move is invalid if you try to
22     * move your opponent's pieces or the move does not follow the
23     * rules, see the exercise specification. PRECONDITION: the
24     * (row,column) coordinates are valid positions, that is, all
25     * between (0..7).
26     * @return true if the move is valid, false otherwise */
27    public boolean isMoveValid(int fromRow, int fromColumn,
28                               int toRow, int toColumn);
29
30 }
```

**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 iterations, 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:

```
1 public interface FanControl {
2     /** Return the frequency of the cooling fan given the
3     temperature of air and liquid in a chemical chamber.
4     The ideal liquid temperature is around 75 degrees.
5
6     The frequency (return value) is calculated as follows
7     (in order of precedence):
8
9     if TempLiquid > 90 return 9999 (ALERT)
10    if TempLiquid > 80 return 500 (Max cooling)
11    if TempLiquid < 70 return 0 (No cooling)
12    otherwise return (TempLiquid - 70) * 50
13
14    The air temperature may overrule the above calculation:
15    if TempAir > 100 return 9999
16    if TempAir > 90 return 500
17    */
18    public int fanControl(double TempAir, double TempLiquid);
19 }
```

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 iterations, 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.

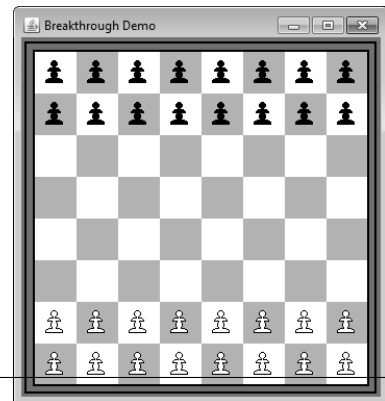
Do not try to cover all requirements if this removes the possibility of broadening the discussion.

### 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.



```
1 public interface Breakthrough {
2     /** Enumeration of the three types of 'pieces' that
3     is possible on a given location on the chess board:
4     black, white, or no piece */
5     public static enum PieceType { BLACK, WHITE, NONE};
6     /** Enumeration of the two types of players in the game,
7     either white or black */
8     public static enum PlayerType { BLACK, WHITE };
9
10    /** Return the type of piece on a given (row, column) on
11    the chess board.
12    @return the type of piece on the location.*/
13    public PieceType getPieceAt( int row, int column );
14
15    /** Return the player that is in turn, i.e. allowed
16    to move.
17    @return the player that may move a piece next */
18    public PlayerType getPlayerInTurn();
19
20    /** Validate a move from a given location (fromRow, fromColumn) to a
21    new location (toRow, toColumn). A move is invalid if you try to
22    move your opponent's pieces or the move does not follow the
23    rules, see the exercise specification. PRECONDITION: the
24    (row, column) coordinates are valid positions, that is, all
25    between (0..7).
26    @return true if the move is valid, false otherwise */
27    public boolean isMoveValid(int fromRow, int fromColumn,
28                               int toRow, int toColumn);
29
30 }
```

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:

```
1 public interface FanControl {
2     /** Return the frequency of the cooling fan given the
3     temperature of air and liquid in a chemical chamber.
4     The ideal liquid temperature is around 75 degrees.
5
6     The frequency (return value) is calculated as follows
7     (in order of precedence):
8
9     if TempLiquid > 90 return 9999 (ALERT)
10    if TempLiquid > 80 return 500 (Max cooling)
11    if TempLiquid < 70 return 0 (No cooling)
12    otherwise return (TempLiquid - 70) * 50
13
14    The air temperature may overrule the above calculation:
15    if TempAir > 100 return 9999
16    if TempAir > 90 return 500
17    */
18    public int fanControl(double TempAir, double TempLiquid);
19 }
```

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.

A cooling fan control system (Da: blæser-baseret kølesystem) is controlling the temperature in a cold store (Da: kølehus) by measuring the temperature and adjusting a cooling fan's rotation frequency (=speed): higher frequency means better cooling. The control system is available in several variants. **Display variants:** It comes in a cheap version whose user interface is three LEDs ("lamps") that light up if fan frequency is low, medium or high respectively, and a more expensive version with a single-line 32 character LCD display showing the frequency. **Sensor variations:** The control system also handles three variants of temperature measurement sensors: Philips, Textronic, or Texas.

Consider the following Java code skeleton:

```
1 public abstract class CoolingFanControlB {
2     public static void main(String args[]) {
3         CoolingFanControlB ctrl = new CoolingFanControl.LED.Phillips();
4         ctrl.controlTemperature();
5     }
6     /** The main controller loop: read sensor and control fan. */
7     */
8     public void controlTemperature() {
9         while (true) {
10            double reading = readTemperature();
11            double fanFrequency = controlAlgorithm( reading );
12            displayFrequency( fanFrequency );
13            // [control the cooling fan]
14        }
15    }
16    abstract void displayFrequency( double f );
17    abstract double readTemperature();
18
19    double controlAlgorithm( double T ) {
20        /* [calculate frequency based on T] */
21        return 250.0; // Fake 'it
22    }
23 }
24
25 class CoolingFanControl_LED_Phillips extends CoolingFanControlB {
26     void displayFrequency( double f ) {
27         /* [Turn off all LEDs] */
28         if ( f < 100 ) { /* [LowSpeedLED.turnOn()] */ }
29         if ( f >= 100 && f < 500 ) { /* [MediumSpeedLED.turnOn()] */ }
30         if ( f >= 500 ) { /* [HighSpeedLED.turnOn()] */ }
31     }
32     double readTemperature() {
33         double reading; // the temperature measured, assigned in code below
34         /* [measure temperature using PHILIPS sensor] */;
35         return reading;
36     }
37 }
```

(Only the subclass for the LED plus Philips combination is shown. The [...] parts in comments are code to be filled in.)

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 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.)

```
1 public abstract class WindCalculator {
2     public static void main(String[] args) {
3         // Example: METAR Wind after Danish regulations.
4         WindCalculator calculator = new METARWindCalculator();
5         int[] values = new int[] {230,7,245,8,234,7}; // fake-it
6         String METAR =
7             calculator.calculateFormatted10MinWind( values,
8                                                     WindCalculator.DANISH );
9     }
10    /** calculate a formatted 10 minute mean string to insert into a
11     * specific meterological report and calculated according to
12     * national algorithms. */
13    public String calculateFormatted10MinWind(int[] datavalues,
14                                             int algorithmType ) {
15        int meanSpeed = 7, meanDirection = 234; // fake-it
16        boolean vrb = false; // fake-it
17        switch ( algorithmType ) {
18            case DANISH:
19                /* calculate means speed, direction, and vrb condition according
20                 to Danish regulations (omitted) */
21                break;
22            case FRENCH: /* French algorithm (omitted) */ break;
23            case GERMAN: /* German algorithm (omitted) */ break;
24        }
25        return format(meanSpeed, meanDirection, vrb);
26    }
27    public abstract String format(int s, int d, boolean vrb);
28
29    /* constants defining which national calculation algorithm to use */
30    public static final int DANISH = 100;
31    public static final int FRENCH = 101;
32    public static final int GERMAN = 102;
33 }
34 class METARWindCalculator extends WindCalculator {
35     public String format(int s, int d, boolean vrb) {
36         String result = "23407"; // fake-it
37         return result;
38     }
39 }
```

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.

## 0.7 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 off by a Java interface:

```
1 public interface SevenSegment {
2     /** turn a LED on or off.
3      * @param led the number of the LED. Range is 0 to 6. The LEDs are
4      * numbered top to bottom, left to right. That is, the top,
5      * horizontal, LED is 0, the top left LED is 1, etc.
6      * @param on if true the LED is turned on otherwise it is turned
7      * off.
8      */
9     void setLED(int led, boolean on);
10 }
```

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

```
1 d.setLED(0, true); d.setLED(1, true); d.setLED(2, true); d.setLED(3, false);
2 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:

```
1 public interface NumberDisplay {
2     /** display a number on a seven segment.
3      * @param number the number to display.
4      * Precondition: number should be in the range 0 to 9.
5      */
6     void display(int number);
7 }
```

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.



## 0.8 Test Stubs and Unit/Integration Testing.

A cooling fan control system (Da: blæser-baseret kølesystem) is controlling the temperature in a cold store (Da: kølehus) by measuring the temperature and adjusting a cooling fan's rotation frequency (=speed): higher frequency means better cooling. The control system is available in several variants. **Display variants:** It comes in a cheap version whose user interface is three LEDs ("lamps") that light up if fan frequency is low, medium or high respectively, and a more expensive version with a single-line 32 character LCD display showing the frequency. **Sensor variations:** The control system also handles three variants of temperature measurement sensors: Philips, Textronic, or Texas.

Consider the following Java code skeleton:

```
1 public abstract class CoolingFanControlB {
2     public static void main(String args[]) {
3         CoolingFanControlB ctrl = new CoolingFanControl.LED.Phillips();
4         ctrl.controlTemperature();
5     }
6     /** The main controller loop: read sensor and control fan. */
7     */
8     public void controlTemperature() {
9         while (true) {
10            double reading = readTemperature();
11            double fanFrequency = controlAlgorithm( reading );
12            displayFrequency( fanFrequency );
13            // [control the cooling fan]
14        }
15    }
16    abstract void displayFrequency( double f );
17    abstract double readTemperature();
18
19    double controlAlgorithm( double T ) {
20        /* [calculate frequency based on T] */
21        return 250.0; // Fake'it
22    }
23 }
24
25 class CoolingFanControl_LED_Phillips extends CoolingFanControlB {
26     void displayFrequency( double f ) {
27         /* [Turn off all LEDs] */
28         if ( f < 100 ) { /* [LowSpeedLED.turnOn()] */ }
29         if ( f >= 100 && f < 500 ) { /* [MediumSpeedLED.turnOn()] */ }
30         if ( f >= 500 ) { /* [HighSpeedLED.turnOn()] */ }
31     }
32     double readTemperature() {
33         double reading; // the temperature measured, assigned in code below
34         /* [measure temperature using PHILIPS sensor] */;
35         return reading;
36     }
37 }
```

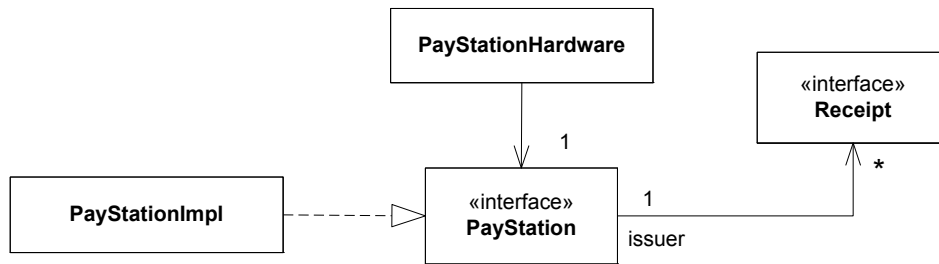
(Only the subclass for the LED plus Philips combination is shown. The [...] parts in comments are code to be filled in.)

You are asked to use terminology, techniques, and tools for test stubs and unit/integration testing to:

- Analyze the specification above with respect to its suitability for doing automatic testing.
- Use UML and Java code to present an alternative design and implementation sketch that improves its ability for doing automated testing.
- Discuss the concepts of test stubs and unit/integration tests in relation to the outlined case.
- Relate to other topics.

## 0.9 Design patterns

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



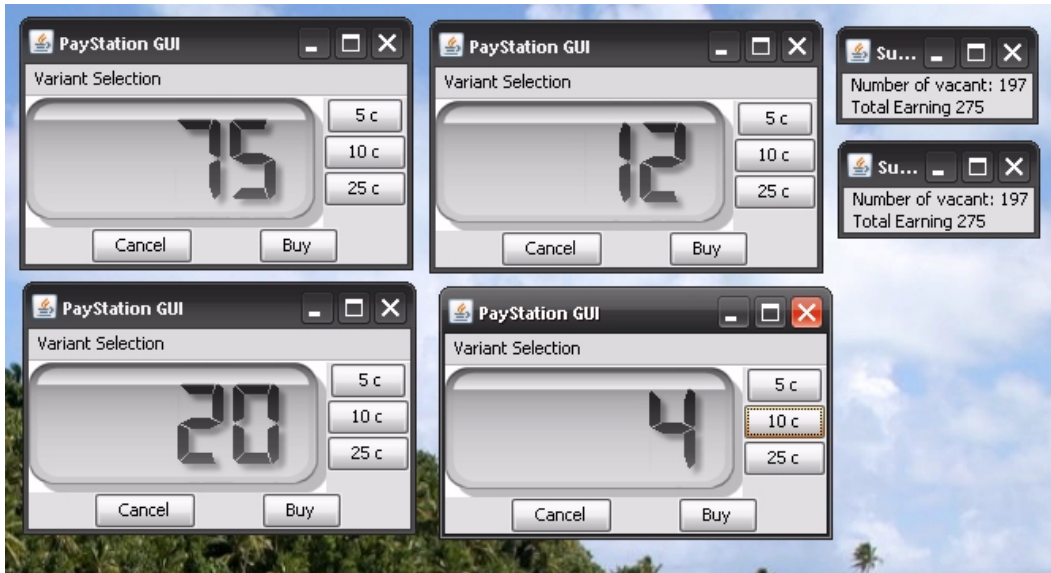
You are now faced with a new customer requirement: *In the four southern pay stations on our parking lot, the pay stations should not accept payment from 19:00 evening until 7:00 morning.* Rephrasing this, the `addPayment` method of interface `PayStation` (FRS p. 46) of these pay stations should throw 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.

## 0.10 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.)

## 0.11 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.)

```
1 public abstract class WindCalculator {
2     public static void main(String[] args) {
3         // Example: METAR Wind after Danish regulations.
4         WindCalculator calculator = new METARWindCalculator();
5         int[] values = new int[] {230,7,245,8,234,7}; // fake-it
6         String METAR =
7             calculator.calculateFormatted10MinWind( values,
8                 WindCalculator.DANISH );
9     }
10    /** calculate a formatted 10 minute mean string to insert into a
11     * specific meterological report and calculated according to
12     * national algorithms. */
13    public String calculateFormatted10MinWind(int[] datavalues,
14                                             int algorithmType ) {
15        int meanSpeed = 7, meanDirection = 234; // fake-it
16        boolean vrb = false; // fake-it
17        switch ( algorithmType ) {
18            case DANISH:
19                /* calculate means speed, direction, and vrb condition according
20                 to Danish regulations (omitted) */
21                break;
22            case FRENCH: /* French algorithm (omitted) */ break;
23            case GERMAN: /* German algorithm (omitted) */ break;
24        }
25        return format(meanSpeed, meanDirection, vrb);
26    }
27    public abstract String format(int s, int d, boolean vrb);
28
29    /* constants defining which national calculation algorithm to use */
30    public static final int DANISH = 100;
31    public static final int FRENCH = 101;
32    public static final int GERMAN = 102;
33 }
34 class METARWindCalculator extends WindCalculator {
35     public String format(int s, int d, boolean vrb) {
36         String result = "23407"; // fake-it
37         return result;
38     }
39 }
```

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

- Describe the ③-①-② process 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.

## 0.12 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.)

```
1 public abstract class WindCalculator {
2     public static void main(String[] args) {
3         // Example: METAR Wind after Danish regulations.
4         WindCalculator calculator = new METARWindCalculator();
5         int[] values = new int[] {230,7,245,8,234,7}; // fake-it
6         String METAR =
7             calculator.calculateFormatted10MinWind( values,
8                                                     WindCalculator.DANISH );
9     }
10    /** calculate a formatted 10 minute mean string to insert into a
11     * specific meterological report and calculated according to
12     * national algorithms. */
13    public String calculateFormatted10MinWind(int[] datavalues,
14                                             int algorithmType ) {
15        int meanSpeed = 7, meanDirection = 234; // fake-it
16        boolean vrb = false; // fake-it
17        switch ( algorithmType ) {
18            case DANISH:
19                /* calculate means speed, direction, and vrb condition according
20                 to Danish regulations (omitted) */
21                break;
22            case FRENCH: /* French algorithm (omitted) */ break;
23            case GERMAN: /* German algorithm (omitted) */ break;
24        }
25        return format(meanSpeed, meanDirection, vrb);
26    }
27    public abstract String format(int s, int d, boolean vrb);
28
29    /* constants defining which national calculation algorithm to use */
30    public static final int DANISH = 100;
31    public static final int FRENCH = 101;
32    public static final int GERMAN = 102;
33 }
34 class METARWindCalculator extends WindCalculator {
35     public String format(int s, int d, boolean vrb) {
36         String result = "23407"; // fake-it
37         return result;
38     }
39 }
```

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

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

## 0.13 Frameworks

A cooling fan control system (Da: blæser-baseret kølesystem) is controlling the temperature in a cold store (Da: kølehus) by measuring the temperature and adjusting a cooling fan's rotation frequency (=speed): higher frequency means better cooling. The control system is available in several variants. **Display variants:** It comes in a cheap version whose user interface is three LEDs ("lamps") that light up if fan frequency is low, medium or high respectively, and a more expensive version with a single-line 32 character LCD display showing the frequency. **Sensor variations:** The control system also handles three variants of temperature measurement sensors: Philips, Textronic, or Texas.

Consider the following Java code skeleton:

```
1 public abstract class CoolingFanControlB {
2     public static void main(String args[]) {
3         CoolingFanControlB ctrl = new CoolingFanControl.LED.Phillips();
4         ctrl.controlTemperature();
5     }
6     /** The main controller loop: read sensor and control fan. */
7     */
8     public void controlTemperature() {
9         while (true) {
10            double reading = readTemperature();
11            double fanFrequency = controlAlgorithm( reading );
12            displayFrequency( fanFrequency );
13            // [control the cooling fan]
14        }
15    }
16    abstract void displayFrequency( double f );
17    abstract double readTemperature();
18
19    double controlAlgorithm( double T ) {
20        /* [calculate frequency based on T] */
21        return 250.0; // Fake 'it
22    }
23 }
24
25 class CoolingFanControl_LED_Phillips extends CoolingFanControlB {
26     void displayFrequency( double f ) {
27         /* [Turn off all LEDs] */
28         if ( f < 100 ) { /* [LowSpeedLED.turnOn()] */ }
29         if ( f >= 100 && f < 500 ) { /* [MediumSpeedLED.turnOn()] */ }
30         if ( f >= 500 ) { /* [HighSpeedLED.turnOn()] */ }
31     }
32     double readTemperature() {
33         double reading; // the temperature measured, assigned in code below
34         /* [measure temperature using PHILIPS sensor] */;
35         return reading;
36     }
37 }
```

(Only the subclass for the LED plus Philips combination is shown. The [...] parts in comments are code to be filled in.)

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

- Sketch a refactored design using composition instead using UML and Java.
- Discuss the original and your refactored design using the concepts of separation and unification of the TEMPLATE METHOD pattern.
- Discuss concepts introduced from a theoretical viewpoint.
- Relate to other topics.

## 0.14 Clean Code and Refactoring

Consider the following (partial) method implementation of our HotCiv's moveUnit():

```
1  public boolean moveUnit( Position from, Position to ) {
2      boolean v;
3      if (playerInTurn == Player.RED) {
4          v = moveRedUnit(from, to);
5      } else
6          v = moveUnitForBluePlayer(to, from);
7      return v;
8  }
9
10 private boolean moveRedUnit( Position from, Position to ) {
11     //if the unit at 'from' isn't null &&
12     //if the tiletype on 'to' isn't ocean &&
13     //if the tiletype on 'to' isn't mountain &&
14     //if it's the turn of the owner of the unit 'from'. &&
15     //if the unit's movement count isn't zero &&
16     //if the distance is less or equal to one &&
17     //if not to and from are the same tile
18     if (unitMap.get(from) != null &&
19         !world.get(to).getTypeString().equals(GameConstants.OCEANS) &&
20         unitMap.get(from).getOwner() == Player.RED &&
21         !getTileAt(to).getTypeString().equals(GameConstants.MOUNTAINS) &&
22         getUnitAt(from).getMoveCount() >= 1 &&
23         Math.abs(from.getColumn() - to.getColumn()) <= 1 &&
24         Math.abs(from.getRow() - to.getRow()) <= 1 &&
25         !to.equals(from) ) {
26         [...]
```

You are asked to use terminology and techniques from Clean Code [Martin, 2009] to:

- Identify the Clean Code properties that are not obeyed in the code fragment.
- Sketch a refactoring of the code fragment to clean (important parts of) it.
- Discuss the ISO 9126 Maintainability quality and its sub-qualities and discuss it with respect to the code.
- Relate to other topics.

## 0.15 Distribution and Broker

A smartphone app “SnappyTalk” allows users to take a picture, and share it with friends on a set of user defined friend-lists, named like “family”, “school-class”, or “grandparents”. For example, I can take a picture and ask SnappyTalk to send it to all users listed in the “family” list. In this exercise the focus will be on *handling the friend-lists*. The SnappyTalk developers have designed the following interface:

```
1 public interface FriendListHandler {
2     // Create a new friend list with the given name
3     public void createList(String listName);
4
5     // Get a list of friends in a given friend list
6     public List<String> getList(String listName);
7
8     // Add a friend with given name to a list; return false if
9     // friendName or listName is undefined on server
10    public boolean addFriendToList(String friendName, String listName);
11
12    // Remove a friend with given name from a list; return false if
13    // friendName or listName is undefined on server
14    public boolean deleteFriendFromList(String friendName, String listName);
15
16 }
```

That is, when the app code invokes one of these methods, then it will execute the associated behavior on the central SnappyTalk server that handles all friend lists and friend names of all users.

We assume that friend names are already present and unique on the SnappyTalk server. That is

```
1 handler.addFriendToList(“henrikbaerbak”, “swea-team”);
```

will succeed iff 1) the user has created a friend list “swea-team”, and 2) “henrikbaerbak” is a known name on the server (typically defined when “henrikbaerbak” installed the SnappyTalk app).

The developers have furthermore decided to code their client-side and server-side implementation using the BROKER pattern using Google Protocol Buffers binary format as marshalling format, and URI Tunneling as the InterProcessCommunication layer.

You are asked to use terminology and techniques from the BROKER pattern to:

- Outline the BROKER pattern’s structure, roles, and responsibilities.
- Sketch Java code for the central broker roles (proxies, invoker) that need to be implemented for this exercise.
- Relate to other topics, notably compositional design.



## 0.16 Distribution and Broker

A SWEA student group has started a company that develops on-line **two player card games** for mobile phones, and has adopted a design similar to the *GameLobby* system, so both players have a GameProxy on their clients that are associated with the same GameServant object on the server, representing the card game.

In the games, a player's cards can attack an opponent's card, thereby reducing its "life" and ultimately destroy it, once its "life" count reaches zero. One of the card games has a magic theme, in which players can transform ("use magic on") a card in his/her hand or on the table, making it into a more powerful card. A typical transformation of a card will double its attack strength, double its life points, make it into a completely different card, etc. A typical, client-side, invocation would look like

```
1 Card oldCard = [...]  
2 Card newCard = game.transform(oldCard.getId(), CONVERT_TO_DRAGON_CARD);
```

using interfaces for the card game domain roles as outlined here:

```
1 public interface Game {  
2     // Transform card with 'cardId' into a NEW card, using the  
3     // transformation 't'  
4     public Card transform(String cardId, Transformation t);  
5     [...]  
6 }  
7 public interface Card {  
8     // Get the unique, remote, objectId  
9     public String getId();  
10    public int getAttackPoints();  
11    public int getLifePointsLeft();  
12 }  
13 public enum Transformation {  
14     CONVERT_TO_DRAGON_CARD,  
15     DOUBLE_THE_ATTACK,  
16     DOUBLE_THE_LIFE,  
17     [...]  
18 }
```

In this exercise, you should focus on **implementing the Card transform(...) method**. Functionally, the server-side GameServant object implements card transformations by first deleting the transformed card from the game, and then create a new card with the new characteristics (become a 'dragon card', double the life points, etc.).

You are asked to use terminology and techniques from the BROKER pattern to:

- Outline the BROKER pattern's structure, roles, and responsibilities.
- Sketch Java code for the central broker roles (proxies, invoker) that need to be implemented for this exercise.
- Sketch Java code for how server created objects are made available for interaction by clients.
- Relate to other topics, notably compositional design.