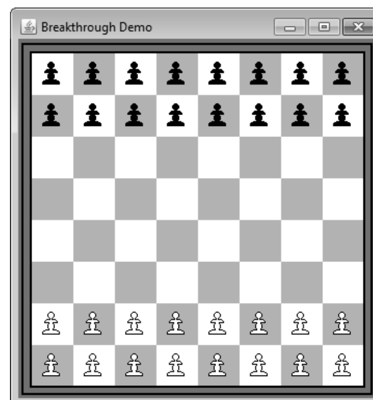
Exam Question Examples 2022

Henrik Bærbak Christensen

November 28, 2022

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:

```
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 posititions, 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 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:

```
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 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 Test-driven Development

We have been asked to develop a **tax calculator** which can compute (simplified) Danish tax for a person. The tax consists of two parts: *bottom-bracket tax and labor market contribution* (Da: bundskat og arbejdsmarkedbidrag) which is calculated based upon income, both salary income and capital income (Da: lønindkomst og kapitalindkomst (dvs. renteindtægter og -udgifter)).

The bottom-bracket (BB) tax is a 15% tax of salary income *above* a 45.000 Dkr. deduction (Da: bundfradrag) (i.e., you pay no tax of the 'first' 45.000 you earn.) You also pay 15% tax of any positive capital income (i.e. no tax if capital income is negative, but tax on any capital income above 0).

Example: Hans has salary income = 55.000 and capital income = 10.000, then the BB tax is 15% of (55.000 - 45.000 + 10.000).

The labor market (LM) tax is an 10% tax of the salary income. However, if the income is from public benefits (Da: overførselsindkomst, eg. SU, pension, or kontanthjælp), you do not pay this tax. You do not pay this tax on capital income either.

Example: Bente has a salary income (from a job) = 55.000 and capital income = 10.000, then the LM tax is 10% of 55.000.

Example: Carl has a salary income (from public benefits) = 55.000 and capital income = 10.000, then the LM tax is 0.

Consider the Java method, which can compute the total of the two taxes (BB+LM) for a person:

```
1 public interface TaxCalculator {
2   /** Calculate combined bottom-braket and labor market tax.
3   @throws IllegalArgumentException if the salary income is negative
4   */
5   public int calculateTax(int salaryIncome, boolean isPublicBenefit,
6                           int capitalIncome)
7           throws IllegalArgumentException {
8 }
```

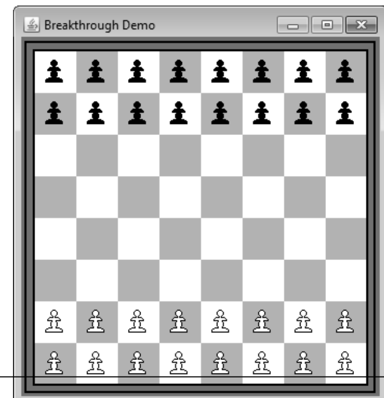
You are asked to describe a preliminary plan for a test-driven development effort. You should use terminology, techniques, and tools from the course to:

- 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.

0.4 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 posititions, 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.

0.5 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.

0.6 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 frequence based on T] */
21     return 250.0; // Fake'it
22   }
23 }
24
25 class CoolingFanControl_LED_Philips 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.

0.7 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.

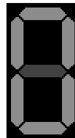
0.8 Test Doubles and unit/integration testing

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:

```
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 doubles 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 doubles(s) according to the classification of Meszaros (Section 12.6 in FRS 2nd Edition).
- Relate to other topics.

0.9 Test Doubles 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 frequence based on T] */
21     return 250.0; // Fake'it
22   }
23 }
24
25 class CoolingFanControl_LED_Philips 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 }
```

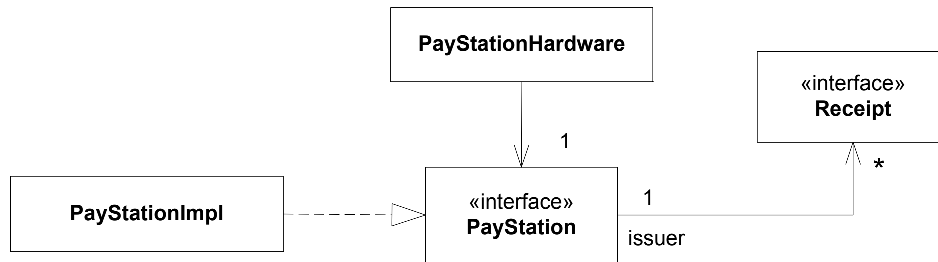
(Only the subclass for the LED plus Philips combination is shown. The [...] parts in comments are code to be filled in.)

You are asked to use terminology, techniques, and tools for test doubles 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 doubles and unit/integration tests in relation to the outlined case.
- Relate to other topics.

0.10 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.

0.11 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.)

0.12 Design Patterns

The startup company *BigCloud* provides a cloud based SQL database service free of charge. Developers can utilize a database using the BigBase Java interface that has methods for updating tables as well as make queries using standard SQL statements:

```
1 import java.sql.*;
2
3 public interface BigBase {
4   // update tables (CREATE and UPDATE statements)
5   public void executeUpdate(String sqlUpdate) throws SQLException;
6   // query (SELECT FROM WHERE statement)
7   public ResultSet executeQuery(String sqlQuery) throws SQLException;
8 }
```

You shall assume that the actual execution of SQL on the BigBase server is handled by a class implementing the BigBase interface. A tentative implementation on the server side is shown below, where Request objects are received on the server for every internet call from a client. The server logs the client in, using his/her req.username, and stores/uses the reference to invoke methods on his/her BigBase implementation:

```
1 private Map<String,BigBase> databaseMap; // Map username to database reference
2 public void handleRemoteClientLogin(LoginRequest req) {
3   // [log the user with username 'req.username' in]
4   BigBase database = new BigBaseImpl();
5   databaseMap.put(req.username, database);
6 }
7 public void handleRemoteClientRequest(Request req) throws SQLException {
8   BigBase database = databaseMap.get(req.username); // get database for user
9   if ( req.type == Request.UPDATE ) {
10    database.executeUpdate(req.statement);
11  } else if ( req.type == Request.QUERY ) {
12    ResultSet rs = database.executeQuery(req.statement);
13  }
14  // [send the answer back to client]
15 }
```

Now, one year after launch, BigCloud is highly successfull and wants to start generating profits from its success. Therefore it is decided on a *pay-per-use* model such that **the first 1000 updates per month are free while all subsequent updates cost 1 cent pr call to the executeUpdate method**. It is considered highly likely that the limit of free updates and the cost of each update will change in the future.

It has also been decided that **statistics data on all queries should be collected**, i.e., all executeQuery method calls for all users should be counted on the servers.

You are asked to identify design pattern(s) that will allow BigCloud to implement this behaviour.

- Identify suitable design pattern(s) to implement a flexible way to fulfil the requirements.
- Sketch Java code and UML diagrams for the solution.
- Discuss design patterns from a theoretical viewpoint.
- Relate to other topics.

Hint: The set of patterns to consider are ADAPTER, COMMAND, DECORATOR, and PROXY.

0.13 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 ③-①-②process 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.

0.14 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 cental aspects in the refactored design using UML and Java.
- Discuss concepts introduced from a theoretical viewpoint.
- Relate to other topics.

0.15 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 frequence based on T] */
21     return 250.0; // Fake'it
22   }
23 }
24
25 class CoolingFanControl_LED_Philips 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.

0.16 Clean Code and Refactoring

A HotStone Game implementation uses an array to implement the battlefield of the two players, index 0 is Findus, while index 1 is Peddersen:

```
private ArrayList<Card>[] field;
```

Consider the following (partial) method implementation of our HotStone's attackCard():

```
@Override
public Status attackCard(Player playerAttacking,
                         Card attackingCard, Card defendingCard) {
  Status status = null;
  if (playerAttacking == Player.FINDUS)
    status = attackCardByFindus(attackingCard, defendingCard);
  else
    status = peddersensAttackCard(defendingCard, attackingCard);

  return status;
}

private Status peddersensAttackCard(Card defendingCard, Card attackingCard) {
  Status status;
  if (attackingCard.getOwner() != Player.PEDDERSEN) {
    status = Status.NOT_OWNER;
  } else {
    if (defendingCard.getOwner() == Player.PEDDERSEN) {
      status = Status.ATTACK_NOT_ALLOWED_ON_OWN_MINION;
    } else {
      if (Player.PEDDERSEN != playerInTurn) {
        status = Status.NOT_PLAYER_IN_TURN;
      } else {
        if (!attackingCard.isActive()) {
          status = Status.ATTACK_NOT_ALLOWED_FOR_NON_ACTIVE_MINION;
        } else {
          StandardCard atC = (StandardCard) attackingCard;
          StandardCard defender = (StandardCard) defendingCard;
          // Findus attacks the card
          atC.lowerHealthBy(defender.getAttack());
          defender.lowerHealthBy(atC.getAttack());

          // remove defeated minions
          if (atC.getHealth() <= 0)
            field[1].remove(atC);
          if (defender.getHealth() <= 0)
            field[0].remove(defender);

          // toggle the active flag of attacker
          atC.setActive(false);

          status = Status.OK;
        }
      }
    }
  }
  return status;
}
[Similar code for 'attackCardByFindus']
```

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.

0.17 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”, 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.*

A test case for creating a friend-list, adding some friends, and reviewing it is:

```
1   @Test
2   public void shouldHandleFriendList() {
3     // Given a list of school friends
4     FriendList schoolList = snappy.createFriendList("school");
5     schoolList.addFriend("Bjarne");
6     schoolList.addFriend("Carla");
7
8     // When I try to retrieve a list again
9     FriendList theList = snappy.getFriendList("school");
10    // Then contents is correct
11    assertThat(theList.size(), is(2));
12    assertThat(theList.get(0), is("Bjarne"));
13    assertThat(theList.get(1), is("Carla"));
14  }
```

Given the interfaces:

```
1  public interface SnappyTalk {
2    // Create and return a new FriendList with the given name
3    FriendList createFriendList(String listName);
4    // Return FriendList for the given name
5    FriendList getFriendList(String listName);
6  }
7  public interface FriendList {
8    // Add a friend to my friend list, with the given name
9    void addFriend(String friendName);
10   // Get name of friend at the given index
11   String get(int index);
12   // Return size of the friend list
13   int size();
14 }
```

In this exercise, you should focus on the SnappyTalk.createFriendList() method. Be aware that the FriendList role must be a remote object.

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.

0.18 Distribution and Broker

A SWEA student group has started a company that develops on-line **two player card games** for mobile phones, inspired by their work on the SWEA mandatory HotStone project.

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, similar to HotStone. 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 field, 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, CONVERT_TO_DRAGON_CARD);
```

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

```
1  public interface Game {
2    // Transform given card into a NEW card, using the
3    // transformation 't'
4    public Card transform(Card card, 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**, that is, the client proxy code and the invoker code for this 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.), and return that using our Broker's *pass-by-reference* technique.

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.