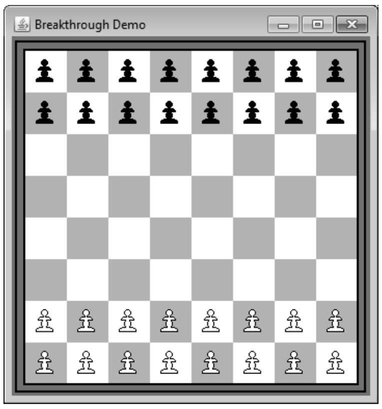
Exam Question Examples 2023

Henrik Bærbak Christensen

January 11, 2024

Contents

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 execute a test-driven development effort to develop the above specification. You should use terminology, techniques, and tools from the course to:

- Cover steps and TDD principles in two or more initial interations and give examples of JUnit test case code for them, as well as sketch initial production code.
- Broaden the discussion to include basic definitions, terminology, and techniques in TDD.
- 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:

```
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 execute a test-driven development effort to develop the above specification. You should use terminology, techniques, and tools from the course to:

- Cover steps and TDD principles in two or more initial interations and give examples of JUnit test case code for them, as well as sketch initial production code.
- Broaden the discussion to include basic definitions, terminology, and techniques in TDD.
- 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:

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

You are asked to execute a test-driven development effort to develop the above specification. You should use terminology, techniques, and tools from the course to:

- Cover steps and TDD principles in two or more initial interations and give examples of JUnit test case code for them, as well as sketch initial production code.
- Broaden the discussion to include basic definitions, terminology, and techniques in TDD.
- 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 Test-driven development.

Danish CPR numbers use a format consisting of two groups of digits separated by a "–". The first group of 6 digits encodes the birthday, whereas the second group of 4 digits encodes sex, century, and a validation, similar to a checksum. In this exercise, we look at validating if a string matches a subset of the requirements of a valid CPR number.

Consider the following specification:

```java
public interface CPRTest {
  /** Do a preliminary test of a Danish CPR string.
      A CPR has 6 digits stating the birthday, and 4 digits which
      are control digits, separated by a dash '-'.

      This method returns true iff:
       - the length of the string is exactly 11 characters
       - the form is "nnnnnn-nnnn" where n is a number between 0-9
         and there is exactly a '-' at position 6.
       - the first two digits represent a date in interval 01-31
       - the second two digits represent a month in interval 01-12
       - the third two digits represent the (last part of) year (00-99)

       - the first three control digits (just after the dash) are NOT in range
         - [537; 557] or
         - [637; 657] or
         - [737; 757] or
         - [837; 857]
   */
  public boolean preTestCPR(String cpr);
}
```

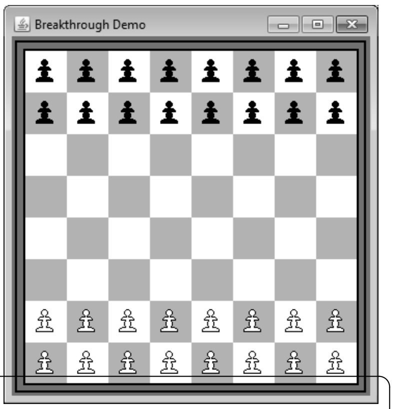
The first three control digits determine the century after a complex and irrelevant algorithm which disallow values in the shown ranges. That is, a string like "040119-**535**1" is valid (first three control digits are NOT in range [537; 557]), whereas a string "010119-**542**1" is not (first three control digits ARE in range [537; 557]).

You are asked to execute a test-driven development effort to develop the above specification. You should use terminology, techniques, and tools from the course to:

- Cover steps and TDD principles in two or more initial interations and give examples of JUnit test case code for them, as well as sketch initial production code.
- Broaden the discussion to include basic definitions, terminology, and techniques in TDD.
- 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.



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:

- Identify conditions in the specification.
- Use the heuristics to generate a equivalence class table, and argue for their representation and coverage properties.
- Generate a (partial) extended test case table using the heuristics of Myers.
- 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 Systematic black-box testing.

Consider the following specification:

```
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:

- Identify conditions in the specification.
- Use the heuristics to generate a equivalence class table, and argue for their representation and coverage properties.
- Generate a (partial) extended test case table using the heuristics of Myers.
- 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.

0.7 Systematic Black-box Testing

Danish CPR numbers use a format consisting of two groups of digits separated by a "–". The first group of 6 digits encodes the birthday, whereas the second group of 4 digits encodes sex, century, and a validation, similar to a checksum. In this exercise, we look at validating if a string matches a subset of the requirements of a valid CPR number.

Consider the following specification:

```java
public interface CPRTest {
  /** Do a preliminary test of a Danish CPR string.
      A CPR has 6 digits stating the birthday, and 4 digits which
      are control digits, separated by a dash '-'.

      This method returns true iff:
       - the length of the string is exactly 11 characters
       - the form is "nnnnnn-nnnn" where n is a number between 0-9
         and there is exactly a '-' at position 6.
       - the first two digits represent a date in interval 01-31
       - the second two digits represent a month in interval 01-12
       - the third two digits represent the (last part of) year (00-99)

       - the first three control digits (just after the dash) are NOT in range
         - [537; 557] or
         - [637; 657] or
         - [737; 757] or
         - [837; 857]
   */
  public boolean preTestCPR(String cpr);
}
```

The first three control digits determine the century after a complex and irrelevant algorithm which disallow values in the shown ranges. That is, a string like "040119-**535**1" is valid (first three control digits are NOT in range [537; 557]), whereas a string "010119-**542**1" is not (first three control digits ARE in range [537; 557]).

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

- Identify conditions in the specification.
- Use the heuristics to generate a equivalence class table, and argue for their representation and coverage properties.
- Generate a (partial) extended test case table using the heuristics of Myers.
- 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.

0.8 Variability Management

A *door alarm system* controls entry to a building by having a numeric key panel at each door (similar to that used by Computer Science). The door's lock is controlled by the software in the panel: the door is only unlocked if a personal and unique 4-digit key code is entered. For instance, Arne has code "1122" and Birte has code "4321". If a proper key code is entered, the person's access to the building is logged in a database, storing time and user name.

You have developed a reliable implementation of the door alarm that uses an internal hashmap (java.util.Map) to lookup user identity for a given keycode, stores the access log as a textfile on a removable flash drive (java.io.PrintStream), and operates the door lock using the DoorLock class. The Java code looks like this:

```java
public class DoorAlarmImpl implements DoorAlarm {
  private Map<String, String> userIndex; // Maps keycodes to username
  private DoorLock doorlock;
  private PrintStream logfile;

  /** Called whenever a 4-digit keycode has been entered; if valid, open door
   * and log username and timestamp in database */
  @Override
  public void handleKeycodeEntered(String keycode) {
    String username = userIndex.get(keycode);
    if (username != null) {
      doorlock.open();
      ZonedDateTime time = ZonedDateTime.now(); // Get Current time and date
      logfile.println("Access granted to "+username+" at: "+time);
    }
  }

  public DoorAlarmImpl() {
    userIndex = initializeUserIndex();
    doorlock = initializeDoorLockDriver();
    logfile = initializeLogFile();
  }

  // initialize... methods omitted
}
```

('ZonedDateTime' represents time in Java)

However, now a new customer wants to use our door alarm system but they already have (key-code, user-identity) pairs stored as a **table in an external SQL database**, and wants the access log to be stored in the **same SQL database**.

You are asked to describe both a **compositional** as well as a **parametric** solution to handle this requirement.

- For each sketch Java code that shows how the variability is handled—you may invent interfaces and method signatures that suits the problem.
- Discuss benefits and liabilities of both the parametric and compositional proposals.
- Discuss concepts introduced at a theoretical level.
- Relate to other topics

If pressed for time, focus on the compositional solution.

0.9 Variability Management

A *door alarm system* controls entry to a building by having a numeric key panel at each door (similar to that used by Computer Science). The door's lock is controlled by the software in the panel: the door is only unlocked if a personal and unique 4-digit key code is entered. For instance, Arne has code "1122" and Birte has code "4321". If a proper key code is entered, the person's access to the building is logged in a database, storing time and user name.

You have developed a reliable implementation of the door alarm that uses an internal hashmap (java.util.Map) to lookup user identity for a given keycode, stores the access log as a textfile on a removable flash drive (java.io.PrintStream), and operates the door lock using the DoorLock class. The Java code looks like this:

```java
public class DoorAlarmImpl implements DoorAlarm {
  private Map<String, String> userIndex; // Maps keycodes to username
  private DoorLock doorlock;
  private PrintStream logfile;

  /** Called whenever a 4-digit keycode has been entered; if valid, open door
   * and log username and timestamp in database */
  @Override
  public void handleKeycodeEntered(String keycode) {
    String username = userIndex.get(keycode);
    if (username != null) {
      doorlock.open();
      ZonedDateTime time = ZonedDateTime.now(); // Get Current time and date
      logfile.println("Access granted to "+username+" at: "+time);
    }
  }

  public DoorAlarmImpl() {
    userIndex = initializeUserIndex();
    doorlock = initializeDoorLockDriver();
    logfile = initializeLogFile();
  }

  // initialize... methods omitted
}
```

('ZonedDateTime' represents time in Java)

However, now a new customer wants to use our door alarm system but they already have (key-code, user-identity) pairs stored as a **table in an external SQL database**, and wants the access log to be stored in the **same SQL database**.

You are asked to describe both a **compositional** as well as a **polymorphic** solution to handle this requirement.

– For each sketch Java code that shows how the variability is handled—you may invent interfaces and method signatures that suits the problem.
– Discuss benefits and liabilities of both the polymorphic and compositional proposals.
– Discuss concepts introduced at a theoretical level.
– Relate to other topics

If pressed for time, focus on the compositional solution.

0.10 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:

```java
public abstract class CoolingFanControlB {
  public static void main(String args[]) {
    CoolingFanControlB ctrl = new CoolingFanControl_LED_Phillips();
    ctrl.controlTemperature();
  }
  /** The main controller loop: read sensor and control fan.
   */
  public void controlTemperature() {
    while (true) {
      double reading = readTemperature();
      double fanFrequency = controlAlgorithm( reading );
      displayFrequency( fanFrequency );
      // [control the cooling fan]
    }
  }
  abstract void displayFrequency( double f );
  abstract double readTemperature();

  double controlAlgorithm( double T ) {
    /* [calculate frequence based on T] */
    return 250.0; // Fake'it
  }
}

class CoolingFanControl_LED_Philips extends CoolingFanControlB {
  void displayFrequency( double f ) {
    /* [Turn off all LEDs] */
    if ( f < 100 )              { /* [LowSpeedLED.turnOn()] */ }
    if ( f >= 100 && f < 500 )  { /* [MediumSpeedLED.turnOn()] */ }
    if ( f >= 500 )             { /* [HighSpeedLED.turnOn()] */ }
  }
  double readTemperature() {
    double reading; // the temperature measured, assigned in code below
    /* [measure temperature using PHILIPS sensor] */;
    return reading;
  }
}
```

(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.

0.11 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.

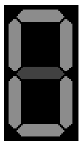
0.12 Test Doubles and unit/integration testing

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:

```java
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:

```java
 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:

```java
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 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.

0.13 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:

```java
public abstract class CoolingFanControlB {
  public static void main(String args[]) {
    CoolingFanControlB ctrl = new CoolingFanControl_LED_Phillips();
    ctrl.controlTemperature();
  }
  /** The main controller loop: read sensor and control fan.
   */
  public void controlTemperature() {
    while (true) {
      double reading = readTemperature();
      double fanFrequency = controlAlgorithm( reading );
      displayFrequency( fanFrequency );
      // [control the cooling fan]
    }
  }
  abstract void displayFrequency( double f );
  abstract double readTemperature();

  double controlAlgorithm( double T ) {
    /* [calculate frequence based on T] */
    return 250.0; // Fake'it
  }
}

class CoolingFanControl_LED_Philips extends CoolingFanControlB {
  void displayFrequency( double f ) {
    /* [Turn off all LEDs] */
    if ( f < 100 )              { /* [LowSpeedLED.turnOn()] */ }
    if ( f >= 100 && f < 500 )  { /* [MediumSpeedLED.turnOn()] */ }
    if ( f >= 500 )             { /* [HighSpeedLED.turnOn()] */ }
  }
  double readTemperature() {
    double reading; // the temperature measured, assigned in code below
    /* [measure temperature using PHILIPS sensor] */;
    return reading;
  }
}
```

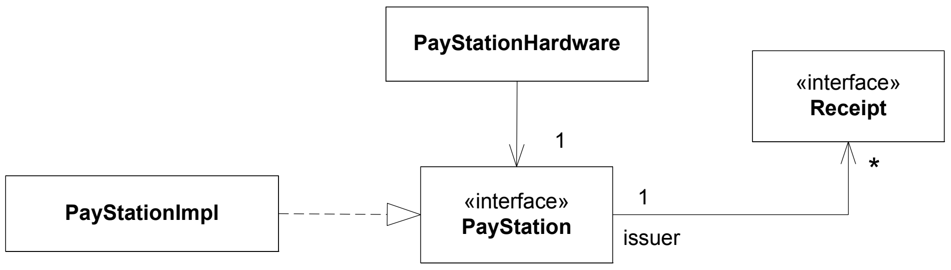
(Only the subclass for the LED plus Philips combination is shown. The [...] parts in comments are code to be filled in.)

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.

0.14 Design patterns

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



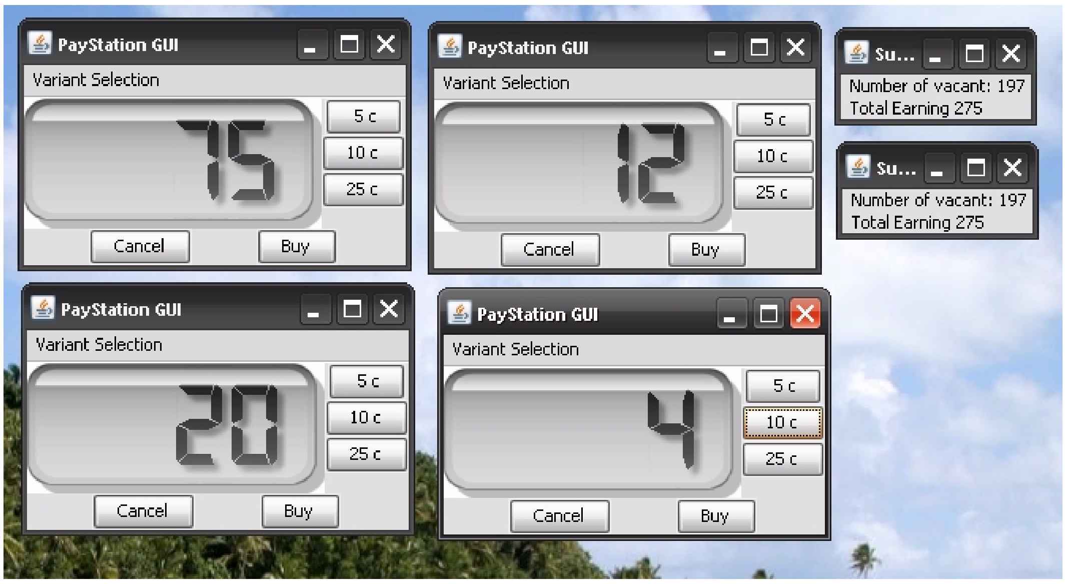
You are now faced with a new customer requirement: *In the four southern pay stations on our parking lot, the pay stations should not accept payment from 19:00 evening until 7:00 morning.* Rephrasing this, the addPayment method of interface PayStation of these pay stations should throw an IllegalConException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

0.15 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.



Number of vacant: 197
Total Earning 275

Number of vacant: 197
Total Earning 275

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.)

0.16 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:

```
import java.sql.*;

public interface BigBase {
  // update tables (CREATE and UPDATE statements)
  public void executeUpdate(String sqlUpdate) throws SQLException;
  // query (SELECT FROM WHERE statement)
  public ResultSet executeQuery(String sqlQuery) throws SQLException;
}
```

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:

```
private Map<String,BigBase> databaseMap; // Map username to database reference
public void handleRemoteClientLogin(LoginRequest req) {
  // [log the user with username 'req.username' in]
  BigBase database = new BigBaseImpl();
  databaseMap.put(req.username, database);
}
public void handleRemoteClientRequest(Request req) throws SQLException {
  BigBase database = databaseMap.get(req.username); // get database for user
  if ( req.type == Request.UPDATE ) {
    database.executeUpdate(req.statement);
  } else if ( req.type == Request.QUERY ) {
    ResultSet rs = database.executeQuery(req.statement);
  }
  // [send the answer back to client]
}
```

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.

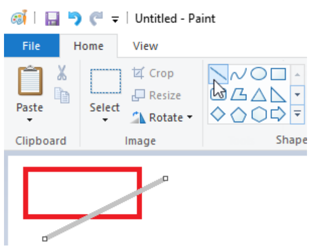
It has also been decided that **statistics data on all queries should be collected**, i.e., all executeQuery method calls for all users should be counted on the servers.

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.

0.17 Design Patterns



A simple drawing program, similar to "Paint", allows users to draw shapes like lines, curves, rectangles, etc., using the mouse. To choose a shape, the user clicks a tool button (like 'Draw Line' in the figure) and next the program interprets mouse down, mouse drag, and mouse up, to draw the shape. A first prototype of the design, having only a line drawing and rectangle drawing tool, and using print statements to simulate shape drawing, looks like this:

```java
public class DrawingProgram {
  public static void main(String[] args) {
    DrawingEditor editor = new DrawingEditor();
    editor.setTool(ToolType.DrawLineTool);
    // simulate mouse events, to demonstrate behavior
    editor.mousePressed();
    editor.mouseMoved();
    editor.mouseReleased();
  }
}
interface Controller {
  /** when mouse button is pressed */
  public void mousePressed();
  /** when mouse is moved across window while button down*/
  public void mouseMoved();
  /** when mouse button is released */
  public void mouseReleased();
}
enum ToolType { DrawLineTool, DrawRectangleTool };

class DrawingEditor implements Controller {
  ToolType currentToolType;
  public void setTool(ToolType newToolType) { currentToolType = newToolType; }

  public void mousePressed() {
    if ( currentToolType == ToolType.DrawRectangleTool ) {
      System.out.println("Rectangle Drawing Start");  }
    if ( currentToolType == ToolType.DrawLineTool ) {
      System.out.println("Line Drawing Start"); }
  }
  public void mouseMoved() {
    if ( currentToolType == ToolType.DrawRectangleTool ) {
      System.out.println("Rectangle Drawing Drag");  }
    if ( currentToolType == ToolType.DrawLineTool ) {
      System.out.println("Line Drawing Drag"); }
  }
  public void mouseReleased() {
    if ( currentToolType == ToolType.DrawRectangleTool ) {
      System.out.println("Rectangle Drawing Created"); }
    if ( currentToolType == ToolType.DrawLineTool ) {
      System.out.println("Line Drawing Created"); }
  }
}
```

However, we would like to be able to draw many more shapes, like circles, triangles, etc.

You are asked to use design pattern terminology, techniques, and tools to:

– Describe the current design and identify a design pattern that would improve flexibility and allow *change by addition*. You may invent and modify interfaces and method signatures for the case.
– Sketch Java code that implements your new design.
– Discuss benefits and liabilities, and relate to other topics.

0.18 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:

- Describe the compositional design principles and the ③-①-②process.
- Use them to refactor the above design problem to become more flexible. You should sketch concrete Java code and/or UML diagrams.
- Relate the refactored design to multi-dimensional variance, and discuss benefits and liabilities of the original and refactored design.
- Relate to other topics.

0.19 Compositional Design

The RunApp company is designing a framework for developing running apps for smartphones: the runner's GPS location is read from the phone's hardware device, next the location is adjusted to the nearest road or path, and the location is then shown overlaid on a graphical map; similar to the figure on the right.

The RunApp framework must be configurable to support: **GPS hardware variants:** GPS hardware from *Samsung*, *LG*, and *Apple*. **Map services:** Show location on map services provided by *Google* and *Apple*.

Consider the following Java code skeleton which configures the RunApp for a Samsung GPS and Google map service:

```java
public abstract class RunApp {
  public static void main(String[] args) {
    RunApp runapp = new RunApp_SamsungGPS_GoogleMap();
    // enter main loop, constantly reading GPS and updating map
    while (true) {
      runapp.updatePositionOnMap();
    }
  }
  /** read location from GPS and update the map */
  public void updatePositionOnMap() {
    Location l = pollGPSLocation();
    l = adjustPositionToBeOnARoad(l)
    showLocationOnMap(l);
  }
  abstract Location pollGPSLocation();
  abstract void showLocationOnMap(Location location);
  private Location adjustPositionToBeOnARoad(Location location) {
    /* [Correct location so it is on a road or path] */
    return adjustedLocation
  }
}

class RunApp_SamsungGPS_GoogleMap extends RunApp {
  public Location pollGPSLocation() {
    Location l = new Location();
    /* [interact with Samsung hardware to get GPS location] */
    return l;
  }
  public void showLocationOnMap(Location location) {
    /* [Use Google API to show 'l' on a visual map */
  }
}
/** Encapsulate (latitude,longitude) data. Here it
    is a fake-it implementation, always Aarhus */
class Location {
  public int getLatitude() { return 56; }
  public int getLongitude() { return 10; }
}
```

[Only the Samsung/Google variant is shown. The Location *class is a fake here to reduce code size.]*

You are asked to:

- Describe the compositional design principles and the ③-①-②process.
- Use them to refactor the above design problem to become more flexible. You should sketch concrete Java code and/or UML diagrams.
- Relate the refactored design to multi-dimensional variance, and discuss benefits and liabilities of the original and refactored design.
- Relate to other topics.

0.20 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.)

```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 use framework terminology, techniques, and tools to:

- Describe how the design could be refactored to become a framework.
- Sketch the framework design using Java code fragments and/or UML diagrams.
- Analyse your design in terms of the *unification* and *separation* variant of the TEMPLATE METHOD pattern.
- Discuss concepts introduced from a theoretical viewpoint.
- Relate to other topics.

0.21 Frameworks

The RunApp company is designing a framework for developing running apps for smartphones: the runner's GPS location is read from the phone's hardware device, next the location is adjusted to the nearest road or path, and the location is then shown overlaid on a graphical map; similar to the figure on the right.

The RunApp framework must be configurable to support: **GPS hardware variants:** GPS hardware from *Samsung*, *LG*, and *Apple*. **Map services:** Show location on map services provided by *Google* and *Apple*.

Consider the following Java code skeleton which configures the RunApp for a Samsung GPS and Google map service:

```java
public abstract class RunApp {
  public static void main(String[] args) {
    RunApp runapp = new RunApp_SamsungGPS_GoogleMap();
    // enter main loop, constantly reading GPS and updating map
    while (true) {
      runapp.updatePositionOnMap();
    }
  }
  /** read location from GPS and update the map */
  public void updatePositionOnMap() {
    Location l = pollGPSLocation();
    l = adjustPositionToBeOnARoad(l)
    showLocationOnMap(l);
  }
  abstract Location pollGPSLocation();
  abstract void showLocationOnMap(Location location);
  private Location adjustPositionToBeOnARoad(Location location) {
    /* [Correct location so it is on a road or path] */
    return adjustedLocation
  }
}

class RunApp_SamsungGPS_GoogleMap extends RunApp {
  public Location pollGPSLocation() {
    Location l = new Location();
    /* [interact with Samsung hardware to get GPS location] */
    return l;
  }
  public void showLocationOnMap(Location location) {
    /* [Use Google API to show 'l' on a visual map */
  }
}
/** Encapsulate (latitude,longitude) data. Here it
    is a fake-it implementation, always Aarhus */
class Location {
  public int getLatitude() { return 56; }
  public int getLongitude() { return 10; }
}
```

[Only the Samsung/Google variant is shown. The Location *class is a fake here to reduce code size.]*

You are asked to use framework terminology, techniques, and tools to refactor the current design:

– Propose a compositional framework design that will handle the suggested variants. Your proposal should include Java code sketches and UML diagrams. You may invent interfaces to suit the problem.

– Analyse your design in terms of the *unification* and *separation* variant of the TEMPLATE METHOD pattern.

– Relate to other topics.

0.22 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:

```java
public abstract class CoolingFanControlB {
  public static void main(String args[]) {
    CoolingFanControlB ctrl = new CoolingFanControl_LED_Phillips();
    ctrl.controlTemperature();
  }
  /** The main controller loop: read sensor and control fan.
   */
  public void controlTemperature() {
    while (true) {
      double reading = readTemperature();
      double fanFrequency = controlAlgorithm( reading );
      displayFrequency( fanFrequency );
      // [control the cooling fan]
    }
  }
  abstract void displayFrequency( double f );
  abstract double readTemperature();

  double controlAlgorithm( double T ) {
    /* [calculate frequence based on T] */
    return 250.0; // Fake'it
  }
}

class CoolingFanControl_LED_Philips extends CoolingFanControlB {
  void displayFrequency( double f ) {
    /* [Turn off all LEDs] */
    if ( f < 100 )                { /* [LowSpeedLED.turnOn()] */ }
    if ( f >= 100 && f < 500 )    { /* [MediumSpeedLED.turnOn()] */ }
    if ( f >= 500 )               { /* [HighSpeedLED.turnOn()] */ }
  }
  double readTemperature() {
    double reading; // the temperature measured, assigned in code below
    /* [measure temperature using PHILIPS sensor] */;
    return reading;
  }
}
```

(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:

- Describe how the design could be refactored to become a framework.
- Sketch the framework design using Java code fragments and/or UML diagrams.
- Analyse your design in terms of the *unification* and *separation* variant of the TEMPLATE METHOD pattern.
- Discuss concepts introduced from a theoretical viewpoint.
- Relate to other topics.

0.23 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:

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

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

```java
@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 (most important) Clean Code properties that are not obeyed in the code fragment. You may document it using the Clean Code Template.
- 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.

0.24 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:

```
  @Test
  public void shouldHandleFriendList() {
    // Given a list of school friends
    FriendList schoolList = snappy.createFriendList("school");
    schoolList.addFriend("Bjarne");
    schoolList.addFriend("Carla");

    // When I try to retrieve a list again
    FriendList theList = snappy.getFriendList("school");
    // Then contents is correct
    assertThat(theList.size(), is(2));
    assertThat(theList.get(0), is("Bjarne"));
    assertThat(theList.get(1), is("Carla"));
  }
```

Given the interfaces:

```
public interface SnappyTalk {
  // Create and return a new FriendList with the given name
  FriendList createFriendList(String listName);
  // Return FriendList for the given name
  FriendList getFriendList(String listName);
}
public interface FriendList {
  // Add a friend to my friend list, with the given name
  void addFriend(String friendName);
  // Get name of friend at the given index
  String get(int index);
  // Return size of the friend list
  int size();
}
```

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.
- Sketch Java code for how server created objects are made available for interaction by clients.
- Relate to other topics, notably compositional design.

0.25 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

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

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

```
public interface Game {
  // Transform given card into a NEW card, using the
  // transformation 't'
  public Card transform(Card card, Transformation t);
  [...]
}
public interface Card {
  // Get the unique (remote) objectId
  public String getId();
  public int getAttackPoints();
  public int getLifePointsLeft();
}
public enum Transformation {
  CONVERT_TO_DRAGON_CARD,
  DOUBLE_THE_ATTACK,
  DOUBLE__THE_LIFE,
  [...]
}
```

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.