

CHAPTER 2

Turning Comments into Code

Example

This is a conference management application. In the conference, every participant will wear a badge. On the badge there is some information of the participant (e.g., name and etc.). In the application the Badge class below is used to store this information. Please read the code and comments below:

```
//It stores the information of a participant to be printed on his badge.
public class Badge {
    String pid; //participant ID
    String engName; //participant's full name in English
    String chiName; //participant's full name in Chinese
    String engOrgName; //name of the participant's organization in English
    String chiOrgName; //name of the participant's organization in Chinese
    String engCountry; //the organization's country in English
    String chiCountry; //the organization's country in Chinese

    //*****
    //constructor.
    //The participant ID is provided. It then loads all the info from the DB.
    //*****
    Badge(String pid) {
        this.pid = pid;
        //*****
        //get the participant's full names.
        //*****
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(pid);
        if (part != null) {
            //get the participant's full name in English.
            engName = part.getELastName() + ", " + part.getEFirstName();
            //get the participant's full name in Chinese.
            chiName = part.getCLastName()+part.getCFirstName();
            //*****
            //get the organization's name and country.
            //*****
            OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
            //find the ID of the organization employing this participant.
            String oid = orgsInDB.getOrganization(pid);
            if (oid != null) {
                Organization org = orgsInDB.locateOrganization(oid);
                engOrgName = org.getEName();
                chiOrgName = org.getCName();
                engCountry = org.getEAddress().getCountry();
                chiCountry = org.getCAddress().getCountry();
            }
        }
    }
}
```

```
    }  
    ...  
}
```

Turn comments into code, making the code as clear as the comments

Consider the first comment:

```
//It stores the information of a participant to be printed on his badge.  
public class Badge {  
    ...  
}
```

Why do we need this comment? Because the programmer thinks the name "Badge" is not clear enough, so he writes this comment to complement this insufficiency. However, if we can use this comment directly as the name of the class, the name will be almost as clear as the comment, then we will not need this separate comment anymore, e.g.:

```
public class ParticipantInfoOnBadge {  
    ...  
}
```

Why do that? Isn't writing comments a good programming style? Before the explanation, let's see how to turn the other comments in the above example into code.

Turn comments into variable names

Consider:

```
public class ParticipantInfoOnBadge {  
    String pid; //participant ID  
    String engName; //participant's full name in English  
    String chiName; //participant's full name in Chinese  
    String engOrgName; //name of the participant's organization in English  
    String chiOrgName; //name of the participant's organization in Chinese  
    String engCountry; //the organization's country in English  
    String chiCountry; //the organization's country in Chinese  
    ...  
}
```

We can turn the comments into variables, then delete the separate comments, e.g.:

```
public class ParticipantInfoOnBadge {  
    String participantId;  
    String participantEngFullName;
```

```

String participantChiFullName;
String engOrgName;
String chiOrgName;
String engOrgCountry;
String chiOrgCountry;
...
}

```

Turn comments into parameter names

Consider:

```

public class ParticipantInfoOnBadge {
    ...
    //*****
    //constructor.
    //The participant ID is provided. It then loads all the info from the DB.
    //*****
    ParticipantInfoOnBadge(String pid) {
        this.pid = pid;
        ...
    }
}

```

We can turn the comments into parameter names, then delete the separate comments, e.g.:

```

public class ParticipantInfoOnBadge {
    ...
    //*****
    //constructor.
    //It loads all the info from the DB.
    //*****
    ParticipantInfoOnBadge(String participantId) {
        this.participantId = participantId;
        ...
    }
}

```

Turn comments into a part of a method body

How to get rid of the comment "It loads all the info from the DB" in the above example? It describes how the constructor of ParticipantInfoOnBadge is implemented (load the information from the database), therefore, we can turn it into a part of the body of the constructor, then delete it:

```

public class ParticipantInfoOnBadge {
    ...
    //*****
    //constructor.

```

```

//*****
ParticipantInfoOnBadge(String participantId) {
    loadInfoFromDB(participantId);
}
void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    ...
}
}

```

Delete useless comments

Sometimes we may come across some comments that are obviously useless, e.g.:

```

public class ParticipantInfoOnBadge {
    ...
    //*****
    //constructor.
    //*****
    ParticipantInfoOnBadge(String participantId) {
        ...
    }
}

```

This comment is useless because even without it anyone can tell that this is a constructor. It not only is useless, but also takes up the precious visual space: A screen can display at most 20 and odds lines, but this useless comment already takes up 3 lines, squeezing out the useful information (e.g., code), making this code fragment hard to understand. Therefore, we should delete it as quickly as possible:

```

public class ParticipantInfoOnBadge {
    ...
    ParticipantInfoOnBadge(String participantId) {
        ...
    }
}

```

Extract some code to form a method and use the comment to name the method

Consider the first comment below:

```

void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    //*****
    //get the participant's full names.
    //*****
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
}

```

```

Participant part = partsInDB.locateParticipant(participantId);
if (part != null) {
    //get the participant's full name in English.
    engFullName = part.getELastName() + ", " + part.getEFirstName();
    //get the participant's full name in Chinese.
    chiFullName = part.getCLastName()+part.getCFirstName();
    //*****
    //get the organization's name and country.
    //*****
    OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
    //find the ID of the organization employing this participant.
    String oid = orgsInDB.getOrganization(participantId);
    if (oid != null) {
        Organization org = orgsInDB.locateOrganization(oid);
        engOrgName = org.getEName();
        chiOrgName = org.getCName();
        engOrgCountry = org.getEAddress().getCountry();
        chiOrgCountry = org.getCAddress().getCountry();
    }
}
}

```

This comment says that the code fragment following it will get the full name of the participant. In order to make the code fragment as clear as this comment, we can extract the code fragment into a method and use this comment to name the method, making this separate comment no longer necessary:

```

void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    getParticipantFullNames();
    //*****
    //get the organization's name and country.
    //*****
    //find the ID of the organization employing this participant.
    OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
    String oid = orgsInDB.getOrganization(participantId);
    if (oid != null) {
        Organization org = orgsInDB.locateOrganization(oid);
        engOrgName = org.getEName();
        chiOrgName = org.getCName();
        engOrgCountry = org.getEAddress().getCountry();
        chiOrgCountry = org.getCAddress().getCountry();
    }
}
void getParticipantFullNames() {
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
    Participant part = partsInDB.locateParticipant(participantId);
    if (part != null) {
        //get the participant's full name in English.
        engFullName = part.getELastName() + ", " + part.getEFirstName();
        //get the participant's full name in Chinese.
        chiFullName = part.getCLastName()+part.getCFirstName();
    }
}
}

```

Likewise, the code fragment to get the information of the organization of the participant can be extracted into a method and be named by the comment, making the comment unnecessary:

```

void loadInfoFromDB(String participantId) {
    this.participantId = participantId;
    getParticipantFullNames();
    getOrgNameAndCountry();
}
void getParticipantFullNames() {
    ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
    Participant part = partsInDB.locateParticipant(participantId);
    if (part != null) {
        //get the participant's full name in English.
        engFullName = part.getELastName() + ", " + part.getEFirstName();
        //get the participant's full name in Chinese.
        chiFullName = part.getCLastName()+part.getCFirstName();
    }
}
void getOrgNameAndCountry() {
    OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
    //find the ID of the organization employing this participant.
    String oid = orgsInDB.getOrganization(participantId);
    if (oid != null) {
        Organization org = orgsInDB.locateOrganization(oid);
        engOrgName = org.getEName();
        chiOrgName = org.getCName();
        engOrgCountry = org.getEAddress().getCountry();
        chiOrgCountry = org.getCAddress().getCountry();
    }
}
}

```

An extracted method can be put into another class

Consider the two comments below:

```

public class ParticipantInfoOnBadge {
    ...
    void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
            //get the participant's full name in English.
            engFullName = part.getELastName() + ", " + part.getEFirstName();
            //get the participant's full name in Chinese.
            chiFullName = part.getCLastName()+part.getCFirstName();
        }
    }
}

```

As the programmer thinks these code fragments not clear enough, then he should extract them and use the comments to name them. But this time the extracted methods should be put into the Participant class instead of the ParticipantInfoOnBadge class:

```

public class ParticipantInfoOnBadge {
    ...
    void getParticipantFullNames() {

```

```

ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
Participant part = partsInDB.locateParticipant(participantId);
if (part != null) {
    engFullName = part.getEFullName();
    chiFullName = part.getCFullName();
}
}
}
public class Participant {
    String getEFullName() {
        return getELastName() + ", " + getEFirstName();
    }
    String getCFullName() {
        return getCLastName() + getCFirstName();
    }
}
}

```

Use a comment to name an existing method

Consider the comment below:

```

public class ParticipantInfoOnBadge {
    ...
    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        //find the ID of the organization employing this participant.
        String oid = orgsInDB.getOrganization(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}
}

```

We need the comment of "find the ID of the organization employing..." only because the name "getOrganization" is not clear enough, so, we should directly use the comment as the name:

```

public class ParticipantInfoOnBadge {
    ...
    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        String oid = orgsInDB.findOrganizationEmploying(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}
}

```

```

public class OrganizationsInDB {
    ...
    void findOrganizationEmploying(String participantId) {
        ...
    }
}

```

The improved code

The improved code is shown below (all the comments have been turned into code and no longer exist separately):

```

public class ParticipantInfoOnBadge {
    String participantId;
    String participantEngFullName;
    String participantChiFullName;
    String engOrgName;
    String chiOrgName;
    String engOrgCountry;
    String chiOrgCountry;

    ParticipantInfoOnBadge(String participantId) {
        loadInfoFromDB(participantId);
    }
    void loadInfoFromDB(String participantId) {
        this.participantId = participantId;
        getParticipantFullNames();
        getOrgNameAndCountry();
    }
    void getParticipantFullNames() {
        ParticipantsInDB partsInDB = ParticipantsInDB.getInstance();
        Participant part = partsInDB.locateParticipant(participantId);
        if (part != null) {
            participantEngFullName = part.getEFullName();
            participantChiFullName = part.getCFullName();
        }
    }
    void getOrgNameAndCountry() {
        OrganizationsInDB orgsInDB = OrganizationsInDB.getInstance();
        String oid = orgsInDB.findOrganizationEmploying(participantId);
        if (oid != null) {
            Organization org = orgsInDB.locateOrganization(oid);
            engOrgName = org.getEName();
            chiOrgName = org.getCName();
            engOrgCountry = org.getEAddress().getCountry();
            chiOrgCountry = org.getCAddress().getCountry();
        }
    }
}

```

Why delete the separate comments?

Why delete the separate comments? In fact, comments by themselves are not bad. The problem is that we often do not write clear code (because it is hard), so we take a shortcut (use comments) to hide the problem. The result is, nobody will try to make the code clearer. Later, as the code is updated, commonly nobody updates the comments accordingly. In time, opposed to making the code easier to read, these outdated comments will actually mislead the readers. At the end of the day, what we have is: Some code that is unclear by itself, mixed with some incorrect comments.

Therefore, whenever we see a comment or would like to write one, we should think twice: Can the comment be turned into code, making the code as clear as the comment? You will find that in most of the time the answer is yes. That is, every comment in the code is a good opportunity for us to improve our code. To say in another way, if the code includes a lot of comments, it probably means that the code quality is not that high (however, including a few or no comments does not necessarily mean the code quality is high).

Method name is too long

Consider the example below:

```
class StockItemsInDB {
    //find all the stock items from overseas whose inventory is smaller than 10.
    StockItem[] findStockItems() {
        ...
    }
}
```

In order to turn this comment into code, in principle we should change the code like this:

```
class StockItemsInDB {
    StockItem[] findStockItemsFromOverseasWithInventoryLessThan10() {
        ...
    }
}
```

However, this method name is too long, warning us that the code has problems. What should we do? We should determine: Is the customer of this system really only interested in those stock items from overseas and whose inventory are less than 10? Would he be interested in those stock items from overseas and whose inventory are less than 20? Would he be interested in those stock items from local and whose inventory are greater than 25? If yes, we can turn the comment into parameters:

```
class StockItemsInDB {
    StockItem[] findStockItemsWithFeatures(
        boolean isFromOverseas,
        InventoryRange inventoryRange) {
```

```
    ...  
    }  
}  
class InventoryRange {  
    int minimumInventory;  
    int maximumInventory;  
}
```

If the customer is really only interested in those stock items from overseas and whose inventory are less than 10, he must have some particular reason (why only those but not the others?). After further conversation he may say it is because he needs to replenish the stock, because the shipping from overseas takes longer. Therefore, we find out that what he is really interested is the stock items that need replenishing, instead of those from overseas and whose inventory are less than 10. Therefore, we can turn the real purpose into the method name and turn the comment into the method body:

```
class StockItemsInDB {  
    StockItem[] findStockItemsToReplenish() {  
        StockItem stockItems[];  
        stockItems = findStockItemsFromOverseas();  
        stockItems = findStockItemsInventoryLessThan10(stockItems);  
        return stockItems;  
    }  
}
```

References

- <http://c2.com/cgi/wiki?TreatCommentsWithSuspicion>.



Chapter exercises

Problems

1. Turn the comments into code:

```
class InchToPointConverter {
    //convert the quantity in inches to points.
    static float parseInch(float inch) {
        return inch * 72; //one inch contains 72 points.
    }
}
```

2. Turn the comments into code:

```
class Restaurant extends Account {
    //the string "Rest" is concated with the restaurant ID to
    //form the key.
    final static String RestaurantIDText = "Rest";
    String website;
    //address in Chinese.
    String addr_cn;
    //address in English.
    String addr_en;
    //the restaurant would like to update its fax # with this. After it is
    //confirmed, it will be stored in Account. Before that, it is stored
    //here.
    String numb_newfax;
    boolean has_NewFax = false;
    //a list of holidays.
    Vector Holiday; // a holiday
    //id of the category this restaurant belongs to.
    String catId;
    //a list of business session. Each business session is an array
    //of two times. The first time is the start time. The second time
    //is the end time. The restaurant is open for business in each
    //session.
    Vector BsHour; //Business hour
    ...
    //y: year.
    //m: month.
    //d: date.
    void addHoliday(int y,int m,int d){
        if(y<1900) y+=1900;
        Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
        Holiday.add(aHoliday);
    }
    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
        int fMin = fromHr*60 + fromMin; //start time in minutes.
        int tMin = toHr*60 + toMin; //end time in minutes.
        //make sure both times are valid and the start time is earlier
        //than the end time.
        if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
            GregorianCalendar bs[] = {
                new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
```

```

        new GregorianCalendar(1900,1,1, toHr, toMin,0)
    };
    BsHour.add(bs);
    return true;
} else {
    return false;
}
}
}

```

3. Turn the comments into code:

```

class Account {
    ...
    //check if the password is complex enough, i.e.,
    //contains letter and digit/symbol.
    boolean isComplexPassword(String password){
        //found a digit or symbol?
        boolean dg_sym_found=false;
        //found a letter?
        boolean letter_found=false;
        for(int i=0; i<password.length(); i++){
            char c=password.charAt(i);
            if(Character.isLowerCase(c)||Character.isUpperCase(c))
                letter_found=true;
            else dg_sym_found=true;
        }
        return (letter_found) && (dg_sym_found);
    }
}

```

4. Turn the comments into code:

```

class TokenStream {
    Vector v; //a list of tokens parsed from br.
    int index; //index of the current token in v.
    BufferedReader br; //read the chars from here to parse the tokens.
    int currentChar; //previous char read from br.

    //read the chars from the reader and parse the tokens.
    TokenStream(Reader read) {
        br = new BufferedReader(read);
        takeChar();
        v = parseFile();
        index=0;
    }
    //read the chars from br, parse the tokens and store them into a vector.
    Vector parseFile() {
        Vector v; //accumulate the tokens that have been parsed.
        ...
        return v;
    }
    ...
}

```

5. Turn the comments into code:

```

class FoodDB {

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
//find all foods whose names contain both the keywords. returns
//an iterator on these foods.
public Iterator srchFood(String cName, String eName){
    //it contains all the foods to be returned.
    TreeMap foodTree = new TreeMap();
    Iterator foodList;
    Food food;
    foodList = getAllRecords();
    while (foodList.hasNext()){
        food = (Food) foodList.next();
        //do its names contain both keywords?
        if ((cName==null || //null or "" means no key is specified
            cName.equals("") ||
            food.getCName().indexOf(cName)!=-1)
            &&
            (eName==null || //null or "" means no key is specified
            eName.equals("") ||
            food.getEName().indexOf(eName)!=-1)){
            foodTree.put(food.getFoodKey(), food);
        }
    }
    return foodTree.values().iterator();
}
}
```

6. Turn the comments into code:

```
//an order.
class Order {
    String orderId; //order ID.
    Restaurant r1; //the order is placed for this restaurant.
    Customer c1; //the order is created by this customer.
    //"H": deliver to home address of the customer.
    //"W": deliver to work address of the customer.
    //"O": deliver to the address specified here.
    String addressType;
    String otherAddress; //address if addressType is "O".
    HashMap orderItems; //order items.

    //get the total amount of this order.
    public double getTotalAmt() {
        //total amount.
        BigDecimal amt= new BigDecimal("0.00");
        //for each order item do...
        Iterator iter=orderItems.values().iterator();
        while(iter.hasNext()){
            //add the amount of the next order item.
            OrderItem oi=(OrderItem)iter.next();
            amt = amt.add(new BigDecimal(String.valueOf(oi.getAmount())));
        }
        return amt.doubleValue();
    }
}
}
```

7. Come up with some code that contains comments and then turn the comments into code.

Hints

1. Why not just call the method "convertToPoints"? Also, introduce a variable with a good name to hold the value of 72.

Sample solutions

1. Turn the comments into code:

```
class InchToPointConvertor {
    //convert the quantity in inches to points.
    static float parseInch(float inch) {
        return inch * 72; //one inch contains 72 points.
    }
}
```

Turn the first comment into the method name and the second into a variable name:

```
class InchToPointConvertor {
    final static int POINTS_PER_INCH=72;
    static float convertToPoints(float inch) {
        return inch * POINTS_PER_INCH;
    }
}
```

2. Turn the comments into code:

```
class Restaurant extends Account {
    //the string "Rest" is concated with the restaurant ID to
    //form the key.
    final static String RestaurantIDText = "Rest";
    String website;
    //address in Chinese.
    String addr_cn;
    //address in English.
    String addr_en;
    //the restaurant would like to update its fax # with this. After it is
    //confirmed, it will be stored in Account. Before that, it is stored
    //here.
    String numb_newfax;
    boolean has_NewFax = false;
    //a list of holidays.
    Vector Holiday; // a holiday
    //id of the category this restaurant belongs to.
    String catId;
    //a list of business session. Each business session is an array
    //of two times. The first time is the start time. The second time
    //is the end time. The restaurant is open for business in each
    //session.
    Vector BsHour; //Business hour
    ...
    //y: year.
    //m: month.
    //d: date.
    void addHoliday(int y,int m,int d){
        if(y<1900) y+=1900;
        Calendar aHoliday = (new GregorianCalendar(y,m,d,0,0,0));
        Holiday.add(aHoliday);
    }
    public boolean addBsHour(int fromHr, int fromMin, int toHr, int toMin){
        int fMin = fromHr*60 + fromMin; //start time in minutes.
        int tMin = toHr*60 + toMin; //end time in minutes.
    }
}
```

```

//make sure both times are valid and the start time is earlier
//than the end time.
if(fMin > 0 && fMin <= 1440 && tMin > 0 && tMin <=1440 && fMin < tMin){
    GregorianCalendar bs[] = {
        new GregorianCalendar(1900,1,1, fromHr, fromMin,0),
        new GregorianCalendar(1900,1,1, toHr, toMin,0)
    };
    BsHour.add(bs);
    return true;
} else {
    return false;
}
}
}

```

The most severe problem here is the long comment explaining the Vector named "BsHour". It is explaining how a business session is represented by an array of two times. We should turn this comment into a class definition for business session:

```

class BusinessSession {
    int minStart;
    int minEnd;
    BusinessSession(int fromHour, int fromMinute, int toHour, int toMinute) {
        minStart=getMinutesFromMidNight (fromHour, fromMinute);
        minEnd=getMinutesFromMidNight (toHour, toMinute);
    }
    int getMinutesFromMidNight(int hours, int minutes) {
        return hours*60+minutes;
    }
    boolean isMinutesWithinOneDay(int minutes) {
        return minutes>0 && minutes<=1440;
    }
    boolean isValid() {
        return isMinutesWithinOneDay(minStart) &&
            isMinutesWithinOneDay(minEnd) &&
            minStart<minEnd;
    }
}

class Restaurant extends Account {
    final static String keyPrefix="Rest";
    String website;
    String chineseAddr;
    String englishAddr;
    String faxNoUnderConfirmation;
    boolean hasFaxNoUnderConfirmation = false;
    Vector holidayList;
    String catIdForThisRestaurant;
    Vector businessSessionList;
    ...
    void addHoliday(int year, int month, int day){
        if(year<1900) year+=1900;
        Calendar aHoliday = (new GregorianCalendar(year,month,day,0,0,0));
        holidayList.add(aHoliday);
    }
    public boolean addBusinessSession(int fromHr, int fromMin, int toHr, int
toMin){

```

```

    BusinessSession bs=new BusinessSession(fromHr, fromMin, toHr, toMin);
    if(bs.isValid()){
        businessSessionList.add(bs);
        return true;
    } else {
        return false;
    }
}
}

```

3. Turn the comments into code:

```

class Account {
    ...
    //check if the password is complex enough, i.e.,
    //contains letter and digit/symbol.
    boolean isComplexPassword(String password){
        //found a digit or symbol?
        boolean dg_sym_found=false;
        //found a letter?
        boolean letter_found=false;
        for(int i=0; i<password.length(); i++){
            char c=password.charAt(i);
            if(Character.isLowerCase(c)||Character.isUpperCase(c))
                letter_found=true;
            else dg_sym_found=true;
        }
        return (letter_found) && (dg_sym_found);
    }
}

```

The first comment contains two sentences. The first sentence ("check if the password is complex enough") describes the purpose of the method, it should be the method name. As the method name (isComplexPassword) is exactly like that, so we can simply delete this sentence. The second sentence ("contains letter and digit/symbol") describes how the method works, so it should become the method body, not the method name.

```

class Account {
    ...
    boolean isComplexPassword(String password){
        return containsLetter(password) &&
            (containsDigit(password) || containsSymbol(password));
    }
    boolean containsLetter(String password) {
        ...
    }
    boolean containsDigit(String password) {
        ...
    }
    boolean containsSymbol(String password) {
        ...
    }
}

```

4. Turn the comments into code:

```

class TokenStream {
    Vector v; //a list of tokens parsed from br.
    int index; //index of the current token in v.
    BufferedReader br; //read the chars from here to parse the tokens.
    int currentChar; //previous char read from br.

    //read the chars from the reader and parse the tokens.
    TokenStream(Reader read) {
        br = new BufferedReader(read);
        takeChar();
        v = parseFile();
        index=0;
    }
    //read the chars from br, parse the tokens and store them into a vector.
    Vector parseFile() {
        Vector v; //accumulate the tokens that have been parsed.
        ...
        return v;
    }
    ...
}

```

The comments can be turned into names of variables, parameters and methods. The comment describing how the constructor works should be turned into the body of the constructor.

```

class TokenStream {
    Vector parsedTokenList;
    int currentTokenIdxInList;
    BufferedReader charInputSourceForParsing;
    int previousCharReadFromSource;

    TokenStream(Reader reader) {
        charInputSourceForParsing = new BufferedReader(reader);
        takeChar();
        parsedTokenList = parseTokensFromInputSource();
        currentTokenIdxInList = 0;
    }
    Vector parseTokensFromInputSource() {
        Vector tokensParsedSoFar;
        ...
        return tokensParsedSoFar;
    }
    ...
}

```

5. Turn the comments into code:

```

class FoodDB {
    //find all foods whose names contain both the keywords. returns
    //an iterator on these foods.
    public Iterator srchFood(String cName, String eName){
        //it contains all the foods to be returned.
        TreeMap foodTree = new TreeMap();
        Iterator foodList;
        Food food;
        foodList = getAllRecords();
    }
}

```

```

while (foodList.hasNext()){
    food = (Food) foodList.next();
    //do its names contain both keywords?
    if ((cName==null || //null or "" means no key is specified
        cName.equals("") ||
        food.getCName().indexOf(cName)!=-1)
        &&
        (eName==null || //null or "" means no key is specified
        eName.equals("") ||
        food.getEName().indexOf(eName)!=-1)){
        foodTree.put(food.getFoodKey(), food);
    }
}
return foodTree.values().iterator();
}
}

```

The comments can be turned into names of variables, parameters and methods.

```

class FoodDB {
    public Iterator findFoodsWithKeywordsInNames(String cKeyword, String
eKeyword){
        TreeMap foodsFound = new TreeMap();
        for (Iterator foodIter=getAllRecords(); foodIter.hasNext(); ){
            Food food = (Food) foodIter.next();
            if (foodContainsKeyInNames(food, cKeyword, eKeyword)) {
                foodsFound.put (food.getFoodKey(), food);
            }
        }
        return foodsFound.values().iterator();
    }
    boolean foodContainsKeyInNames(
        Food food,
        String cKeyword,
        String eKeyword) {
        return nameContainsKeyword(food.getCName(), cKeyword) &&
            nameContainsKeyword(food.getEName(), eKeyword);
    }
    boolean nameContainsKeyword(String name, String keyword) {
        return keywordIsNotSpecified(keyword) || name.indexOf(keyword)!=-1;
    }
    boolean keywordIsNotSpecified(String keyword) {
        return keyword==null || keyword.equals("");
    }
}

```

6. Turn the comments into code:

```

//an order.
class Order {
    String orderId; //order ID.
    Restaurant r1; //the order is placed for this restaurant.
    Customer c1; //the order is created by this customer.
    //"H": deliver to home address of the customer.
    //"W": deliver to work address of the customer.
    //"O": deliver to the address specified here.
    String addressType;
}

```