Software Engineering III

Demeter (continued)

Consider the following

- An object of class Person may talk to (i.e. send messages to) an object of class Oracle
- An object of class Oracle may talk to an object of class Knowledge.
- As there is no direct line of communication between Person and Knowledge, an object of class Person may not communicate directly with an object of class Knowledge.
- If they need to communicate directly, then there is a problem with the diagram.

Knowledge & *Oracle* Classes

```
public class Knowledge
{
    private String answerToLife ;

    public Knowledge()  {
            answerToLife = "The answer is...Nobody Knows!" ;
    }

    public String getAnswerToLife()
    {
            return answerToLife ;
    }
}


public class Oracle {

    private Knowledge knowledge ;

    public Oracle()  {
            knowledge = new Knowledge() ;
    }

    public Knowledge getKnowledge() {
            return knowledge ;
    }
}
```

Note this getter method

Person Class

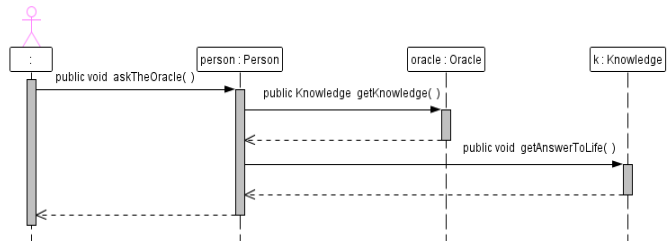
```
public class Person {
    private Oracle oracle ;

    public Person()  {
            oracle = new Oracle() ;
    }

    String askTheOracle()      {
            Knowledge k = oracle.getKnowledge() ;
            return k.getAnswerToLife() ;
    }

    public static void main(String[] args)              {
            Person person = new Person() ;
            String answer = person.askTheOracle() ;
            System.out.println(answer ) ;
    }
}
```

Can you see the problem here?

The Person object is "reaching inside" the Oracle object to get the Knowledge object. The Person object then 'talks' to the Knowledge object.

Law of Demeter cont'd

- A sequence diagram to illustrate the above implementation diagram is shown below.



- The diagram clearly shows the Person object sending a message to the Knowledge object. This is inconsistent with the class diagram and is a breach of the Law of Demeter (why?).

Law of Demeter cont'd

Better implementation

```
public class Oracle {

    private Knowledge knowledge ;

    public Oracle()  {
            knowledge = new Knowledge() ;
    }

    /*      public Knowledge getKnowledge()        {
            return knowledge ;
    }
    */

    public String answerToLife()  {
            return knowledge.getAnswerToLife() ;
    }
}
```

A new "wrapper" method is added to the Oracle class. The Oracle object '**delegates**' to the Knowledge object.

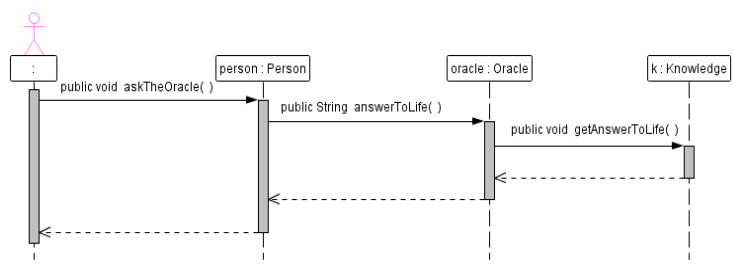
Law of Demeter cont'd

```
public class Person
{
    private Oracle oracle ;

    public Person()  {

            oracle = new Oracle() ;
    }

    String askTheOracle()           {
            return oracle.answerToLife() ;
    }

    public static void main(String[] args)        {
            Person person = new Person() ;
            String answer = person.askTheOracle() ;
            System.out.println(answer ) ;
    }
}
```

Person object now calls the new "wrapper" method, i.e. it no longer reaches inside the Oracle object to get the Knowledge object.

Law of Demeter cont'd

- The sequence diagram to describe the communication between the objects is now shown below.
- This diagram is now consistent with the class diagram. The communication between the objects now adheres to the Law of Demeter (explain).
- Any changes made to the Knowledge class will have no effect on the Person object.

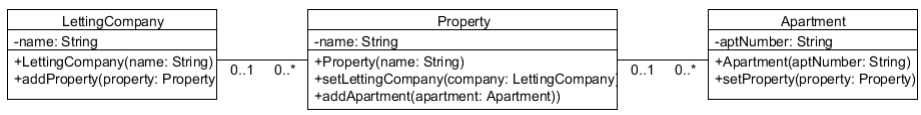


Letting Company Example

- A *letting company* operates a service whereby they provide *apartment* block *properties* for rent. Each apartment would have a specific rent associated with it.
- We want to model a system which can realise the use case:

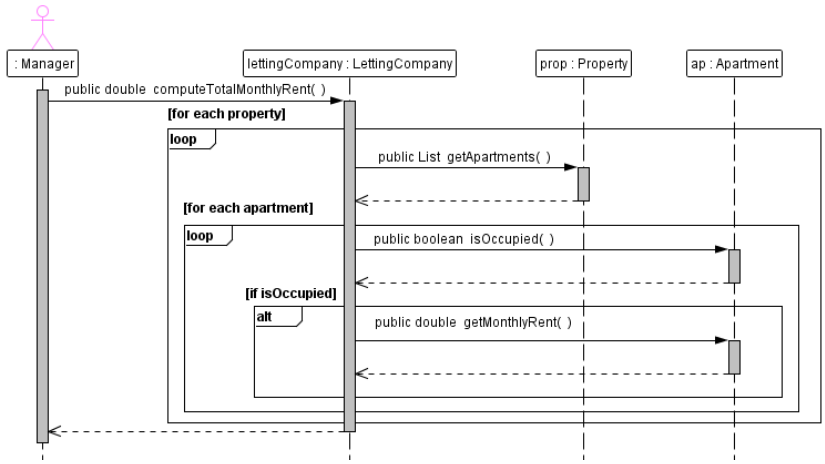
  The manager computes the total monthly rent for the company

Letting Company Class Diagram



Letting Company Sequence Diagram

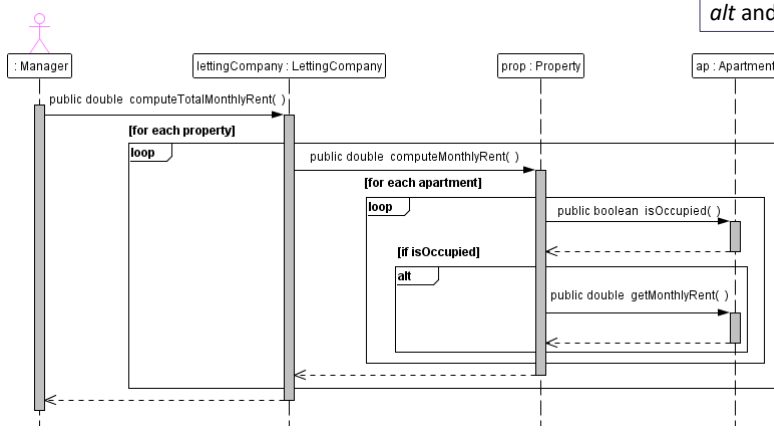
Junior developer's attempt



Letting Company Sequence Diagram

2nd attempt

Note the use of the *alt* and *loop* **frames**



The first attempt did not comply with the association model

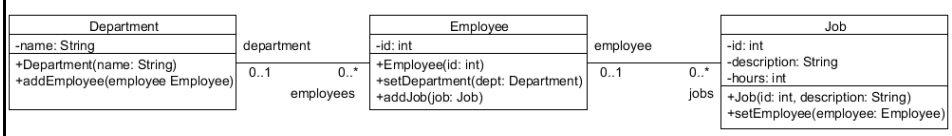
Note no messages sent between the *lettingCompany* and *ap* objects

Department Example

- A (company) *department* has many *employee*s, each of which may have a number of *job*s assigned to them. Each job would require a set number of hours to be associated with it.
- We want to model a system that will realise the following use case:

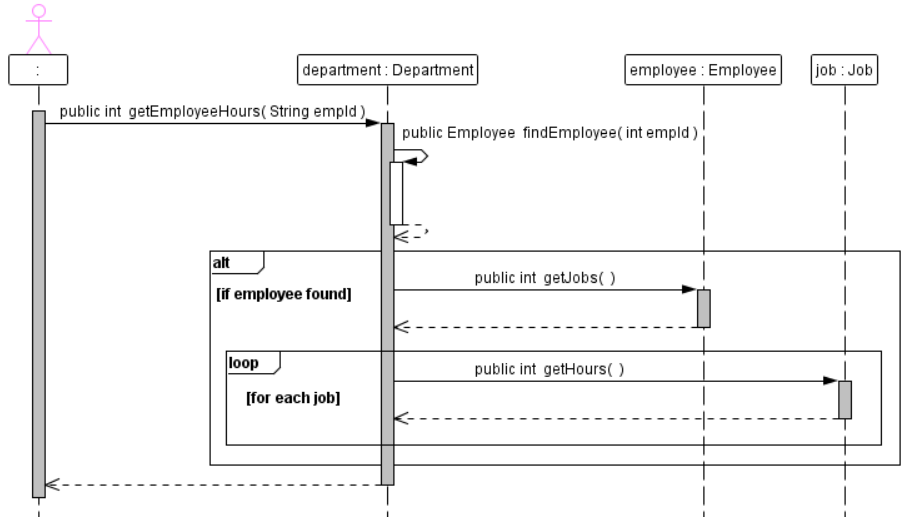
  A manager wants to retrieve the number of hours worked for a given employee

Department: Class Diagram



Department: Sequence Diagram

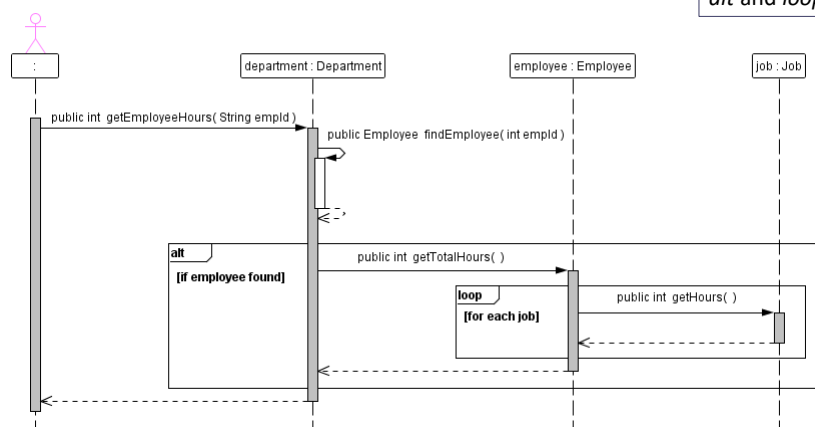
Junior developer's attempt



Department: Sequence Diagram

2nd attempt

Note the use of the *alt* and *loop* **frames**



Note no messages sent between *department* and *job*

Interface Classes

Interface Class

- Definition: An Interface:
  - selection of externally visible public operations
- Two modelling requirements arise:
  - A class may be required to present more than one external interface to other collaborating classes
  - Several classes may be required to present the same interface.
- An '**Interface Class'** provides for this design
  - specifies the externally-visible (public) operations of another class.

19

Interface Class

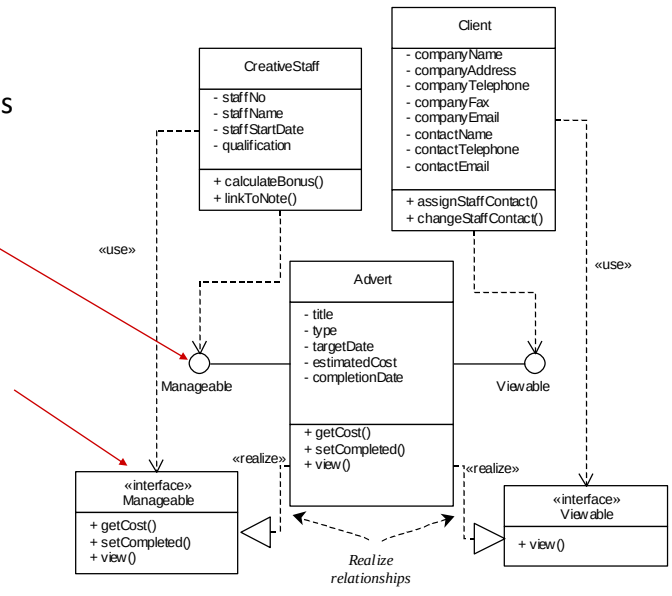
- It provides a mechanism for a class to communicate with a *restricted* view of another class
- An interface class has
  - no internal structure,
  - no attributes,
  - no associations
  - no implementation of its own
    - It compares to an **abstract** class (no instances).
- An interface class typically specifies only a limited part of the behaviour of another class.

20

Interfaces for the class Advert

UML has 2 interfaces notations

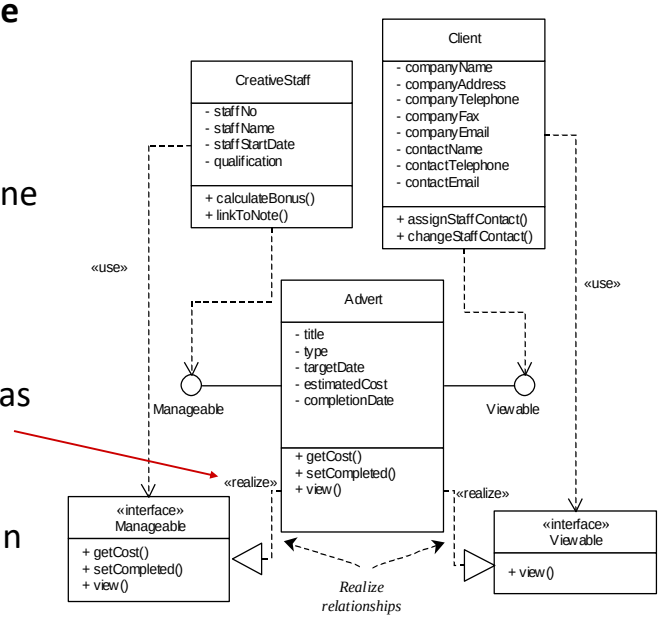
- The small circle icon showing no detail
- A stereotyped class icon with a list of the operations supported



21

Interfaces: Realise Relationship

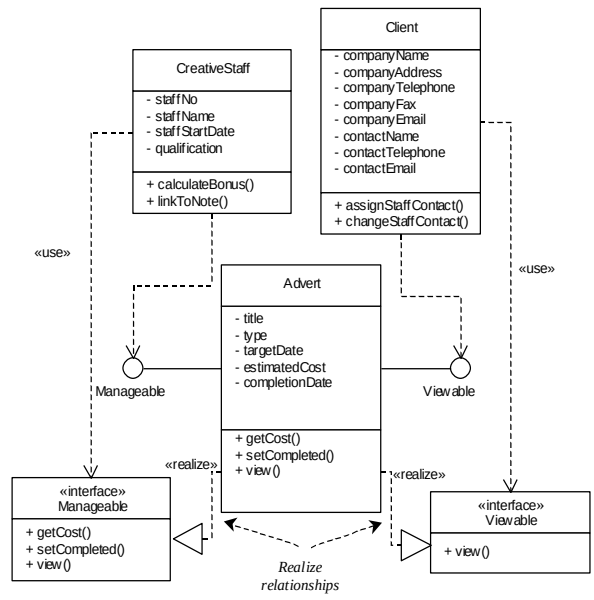
- a relationship between two classes where one class offers the interface of the other, but does not have the same structure as the other.
- Used to show a class supports an interface



22

Interfaces: Realise Relationship

- The realize notation is similar to generalisation and reflects that class Advert inherits the Manageable interface.
- Advert supports at least the operations listed in the interface Manageable

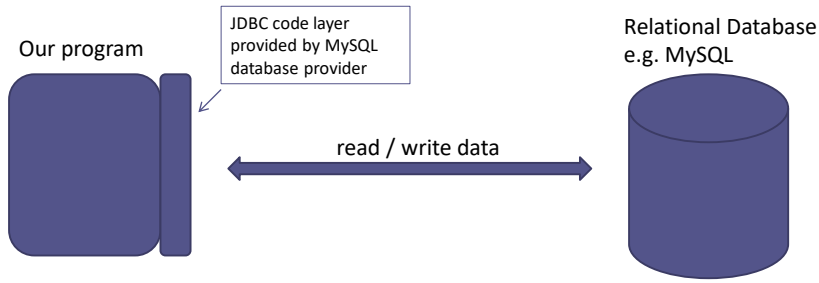


Advert, Manageable and Viewable in in Java

```
public interface Manageable {
   public getCost();
   public setCompleted();
   public view();
}

public interface Viewable {
   public view();
}

public class Advert implements Manageable, Viewable{
   public string title; public int type;..
   public getCost(){…}
   public setCompleted(){…}
   public view(){…}
}
```

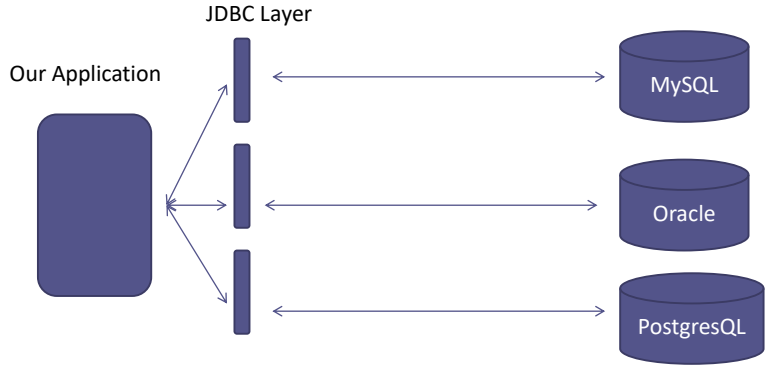
Interfaces & Database Connectivity

Java Database Connectivity (JDBC)



Most major RDBMS providers supply JDBC code libraries to allow java developers to easily connect to their database.

Database Connectivity
Java Database Connectivity (JDBC)



- Each RDBMS has its own implementation of the JDBC Java Interfaces
- **E.g.** Each JDBC library must have a class that implements the `Statement` interface
- `Statement` - The object used for executing a static SQL statement and returning the results it produces

Database Connectivity
Java Database Connectivity (JDBC)

- JDBC code libraries referred to as *JDBC Drivers*
- Essentially consist of a single jar file containing all the necessary java classes
  - E.g. mysql-connector-java-x.x.xx.jar
- Once the jar file is included in the classpath, the specific JDBC classes can be used by your own code

E.g. JDBC Connection Interface

Method Summary

Methods

Modifier and Type	Method and Description
void	abort(Executor executor) Terminates an open connection.
void	clearWarnings() Clears all warnings reported for this Connection obje
void	close() Releases this Connection object's database and JDBC resources immediately instead of
void	commit() Makes all changes made since the previous commit/rollback permanent and releases any
Array	createArrayOf(String typeName, Object[] elements) Factory method for creating Array objects.
Blob	createBlob() Constructs an object that implements the Blob interface.
Clob	createClob() Constructs an object that implements the Clob interface.
NClob	createNClob() Constructs an object that implements the NClob interface.
SQLXML	createSQLXML() Constructs an object that implements the SQLXML interface.
Statement	createStatement() Creates a Statement object for sending SQL statements to the database.
Statement	createStatement(int resultSetType, int resultSetConcurrency) Creates a Statement object that will generate ResultSet objects with the given type and c

Specific JDBC libraries (Oracle, MySQL, PostgreSQL, etc.) each have their own implementation of this interface (among others…)

Database Connectivity

Java Database Connectivity (JDBC)

The java class from MySQL that implements the JDBC interfaces. This registers the JDBC driver with the java DriverManager

The url giving the location of the database

The SQL query to execute against the database

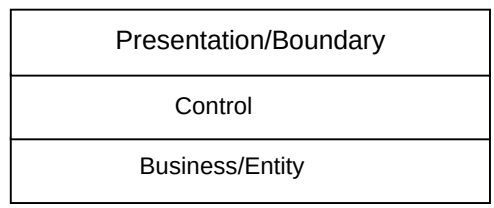
The ResultSet object contains the data returned from the database

```
// Load the database driver
Class.forName( "com.mysql.jdbc.Driver" ) ;

// Get a connection to the database
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test" ) ;

// Get a statement from the connection
Statement stmt = conn.createStatement() ;

// Execute the query
ResultSet rs = stmt.executeQuery( "SELECT * FROM customer" ) ;

// Loop through the result set
while( rs.next() ){
    System.out.println( rs.getString(1)+ " " + rs.getString(2) + " " + rs.getString(3));
}

// Close the result set, statement and the connection
rs.close() ;
stmt.close() ;
conn.close() ;
```

Boundary & Controller Classes

Architecture of the Presentation Layer

- The aim is to separate the classes that have the responsibility for the interface with the user, or with other systems, (boundary classes) from the business classes (entity classes) and the classes that handle the application logic (control classes)
- This is the Three-Tier Architecture



32

Reasons for Separating Classes

- Logical
  - Enables presentation designs that are independent of the physical hardware and the software technology
- Interface Independence
  - Object attributes may be input/displayed in several ways and this responsibility is assigned to the presentation classes
- Reuse
  - Of classes from all 3 layers

33

Presentation Layer

- Handles the interface with users and other connected systems
- Formats and presents data at the interface
- Provides a mechanism for data entry by the user, but the events are handled by control classes
- Presentation can be many forms:
  - for display as text or charts
  - printing on a printer
  - speech synthesis
  - formatting in XML to transfer to another system

34

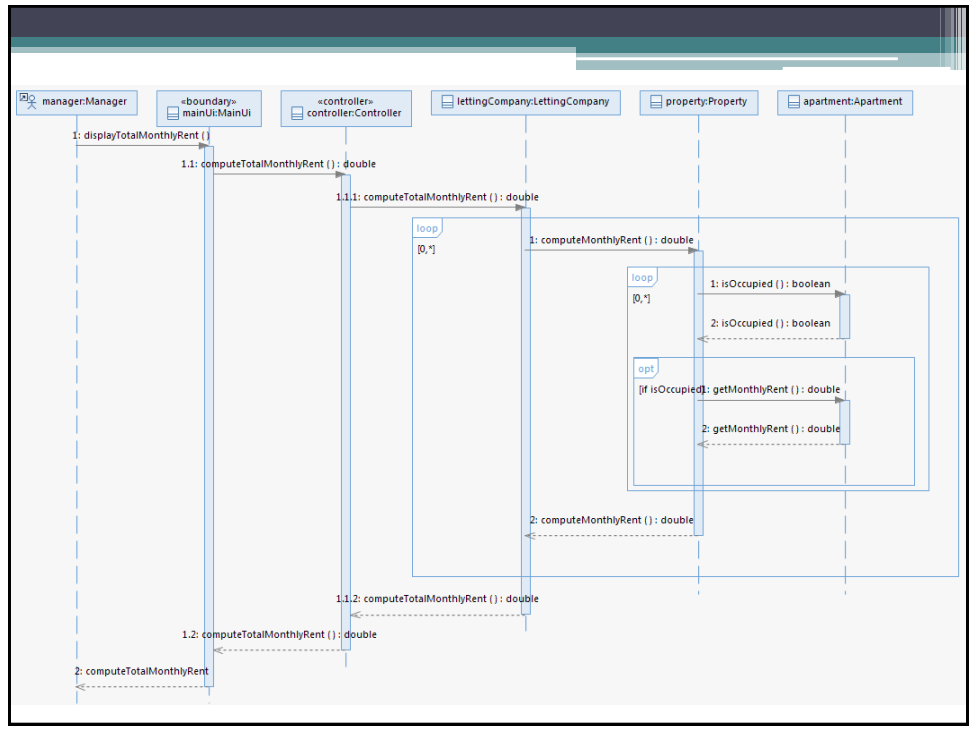
Presentation Layer

- Does not contain business classes—Clients, Campaigns, Adverts, Invoices, Staff etc.
- Does not contain the business logic—rules like 'A Campaign must have one and only one Campaign Manager'.
- Does not handle validation, beyond perhaps simple checks on formatting, value ranges and lookup list.

35

Steps in Developing Boundary Classes

- Prototype the user interface
- Design the classes
- Model the interaction involved in the interface (ISD)
- Model the control of the interface using statechart diagrams (if necessary)


manager:Manager
«boundary»
mainUi:MainUi
«controller»
controller:Controller
lettingCompany:LettingCompany
property:Property
apartment:Apartment
1: displayTotalMonthlyRent ( )
1.1: computeTotalMonthlyRent ( ) : double
1.1.1: computeTotalMonthlyRent ( ) : double
loop
[0,*]
1: computeMonthlyRent ( ) : double
loop
[0,*]
1: isOccupied ( ) : boolean
2: isOccupied ( ) : boolean
opt
[if isOccupied]
1: getMonthlyRent ( ) : double
2: getMonthlyRent ( ) : double
2: computeMonthlyRent ( ) : double
1.1.2: computeTotalMonthlyRent ( ) : double
1.2: computeTotalMonthlyRent ( ) : double
2: computeTotalMonthlyRent