Chapter 4: Implementation

The UML artifacts created during the design work the interaction diagrams and DCDs will be used as input to the code generation process.

4.1 Programming and Iterative, Evolutionary Development

The creation of code in an OO language such as Java or C# is not part of OOA/D it's an end goal. The artifacts created in the Design Model provide some of the information necessary to generate the code.

A strength of use cases plus OOA/D plus OO programming is that they provide an end-to-end roadmap from requirements through to code. The various artifacts feed into later artifacts in a traceable and useful manner, ultimately culminating in a running application. This is not to suggest that the road will be smooth, or can simply be mechanically followed there are many variables. But having a roadmap provides a starting point for experimentation and discussion.

A strength of an iterative and incremental development process is that the results of a prior iteration can feed into the beginning of the next iteration (see Figure). Thus, subsequent analysis and design results are continually being refined and enhanced from prior implementation work. For example, when the code in iteration N deviates from the design of iteration N (which it inevitably will), the final design based on the implementation can be input to the analysis and design models of iteration N+l.

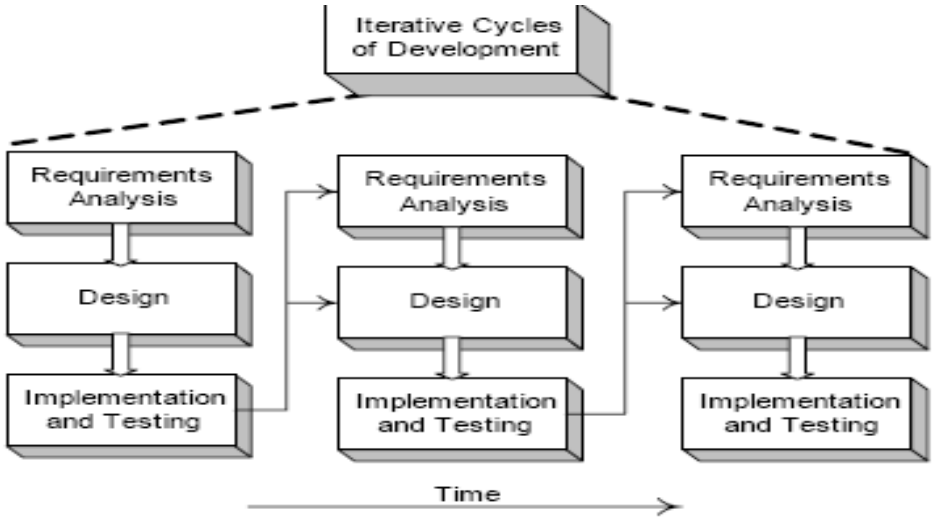


Fig: Implementation in an iteration influences later design.
An early activity within an iteration is to synchronize the design diagrams; the earlier diagrams of iteration N will not match the final code of iteration N, and they need to be synchronized before being extended with new design results.

4.1.2 Creativity and Change During Implementation

Some decision-making and creative work was accomplished during design work. It will be seen during the following discussion that the generation of the code in these examples a relatively mechanical translation process.

However, in general, the programming work is not a trivial code generation step quite the opposite! Realistically, the results generated during design modeling are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved.

Done well, the *ideas* and *understanding* (not the diagrams or documents!) generated during OO design modeling will provide a great base that scales up with elegance and robustness to meet the new problems encountered during programming. But, expect and plan for lots of change and deviation from the design during programming. That's a key and pragmatic attitude in iterative and evolutionary methods.

4.2 Mapping Designs to Code

In UP terms, there exists an Implementation Model. This is all the implementation artifacts, such as the source code, database definitions, JSP/XML/HTML pages, and so forth. Thus, the code being created in this chapter can be considered part of the UP Implementation Model.

4.2.1 Language Samples

Java is used for the examples because of its widespread use and familiarity. However, this is not meant to imply a special endorsement of Java; C#, Visual Basic, C++, Smalltalk, Python, and many more languages are amenable to the object design principles and mapping to code presented in this case study.

4..2.2 Mapping Designs to Code

Implementation in an object-oriented language requires writing source code for:

- Class and interface definitions
- Method definitions

4.2.3 Creating Class Definitions from DCDs (Design Class Diagrams)

At the very least, DCDs depict the class or interface name, super classes, operation signatures, and attributes of a class. This is sufficient to create a basic class definition in an OO language. If the DCD was drawn in a UML tool, it can generate the basic class definition from the diagrams.

4.2.3.1 Defining a Class with Method Signatures and Attributes

From the DCD, a mapping to the attribute definitions (Java *fields*) and method signatures for the Java definitionof *SalesLineItem* is straightforward, as shown in Figure 4.1

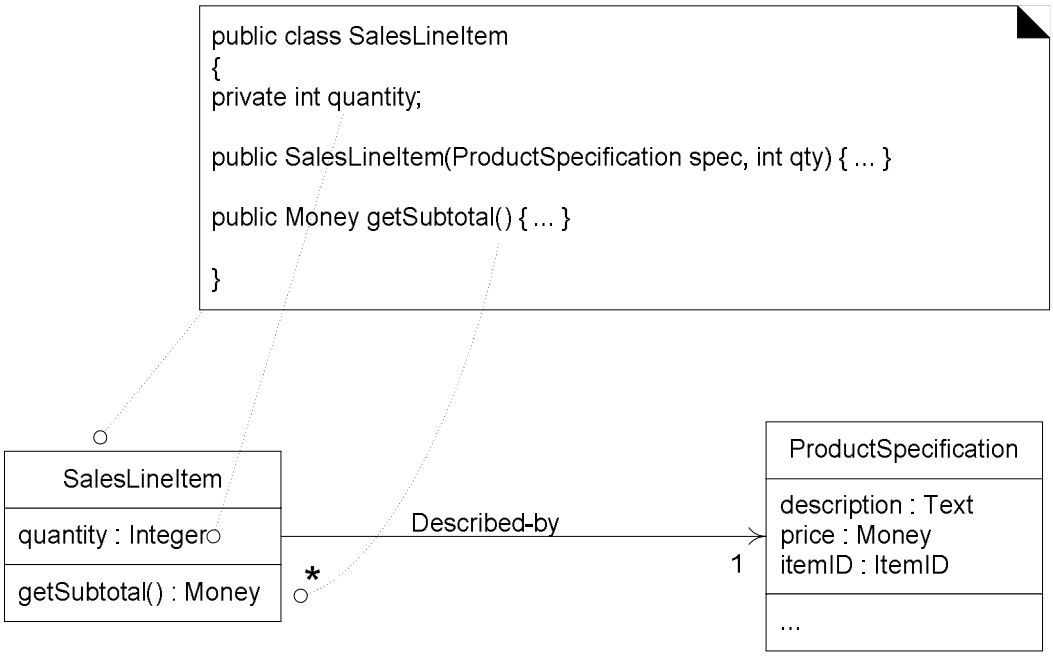


Fig 4.1: **SalesLineItem in Java.**

Note the addition in the source code of the Java constructor *SalesLineItem(…)*. It is derived from the *create(desc, qty)* message sent to a *SalesLineItem* in the *enterItem* interaction diagram. This indicates, in Java, that a constructor supporting these parameters is required. The *create* method is often excluded from the class diagram because of its commonality and multiple interpretations, depending on the target language.

4.3.2.2 Adding Reference Attributes

A reference attribute is an attribute that refers to another complex object, not to a primitive type such as a String, Number, and so on. The reference attributes of a class are suggested by the associations and navigability in a class diagram. For example, a SalesLineItem has an association to a ProductSpecification, with navigability to it. It is common to interpret this as a reference attribute in class SalesLineItem that refers to a ProductSpecification instance (see Figure below).

In Java, this means that an instance field referring to a ProductSpecification instance is suggested.

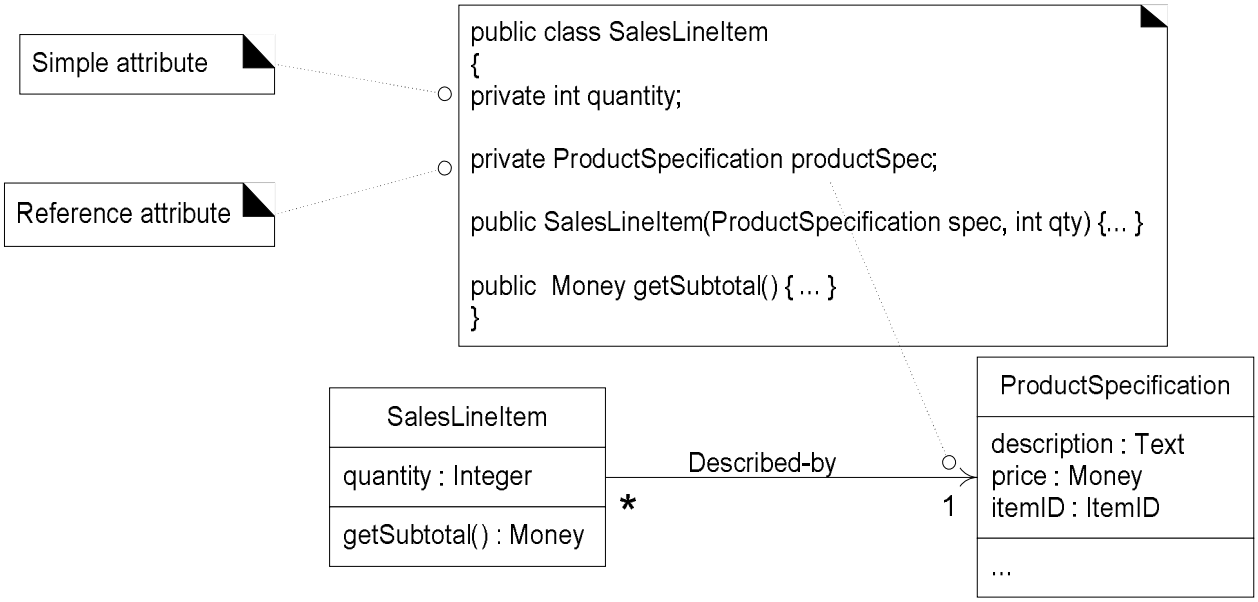


Fig: Adding Reference Attributes

Note that reference attributes of a class are often implied, rather than explicit, in a DCD. For example, although we have added an instance field to the Java definition of SalesLineItem to point to a ProductSpecification, it is not explicitly declared as an attribute in the attribute section of the class box. There is a suggested attribute visibility—indicated by the association and navigability—which is explicitly defined as an attribute during the code generation phase.

4.2.3.3 Reference Attributes and Role Names

The next iteration will explore the concept of role names in static structure diagrams.

Each end of an association is called a role. Briefly, a role name is a name that identifies the role and often provides some semantic context as to the nature of the role. If a role name is present in a class diagram, use it as the basis for the name of the reference attribute during code generation, as shown in Figure below.

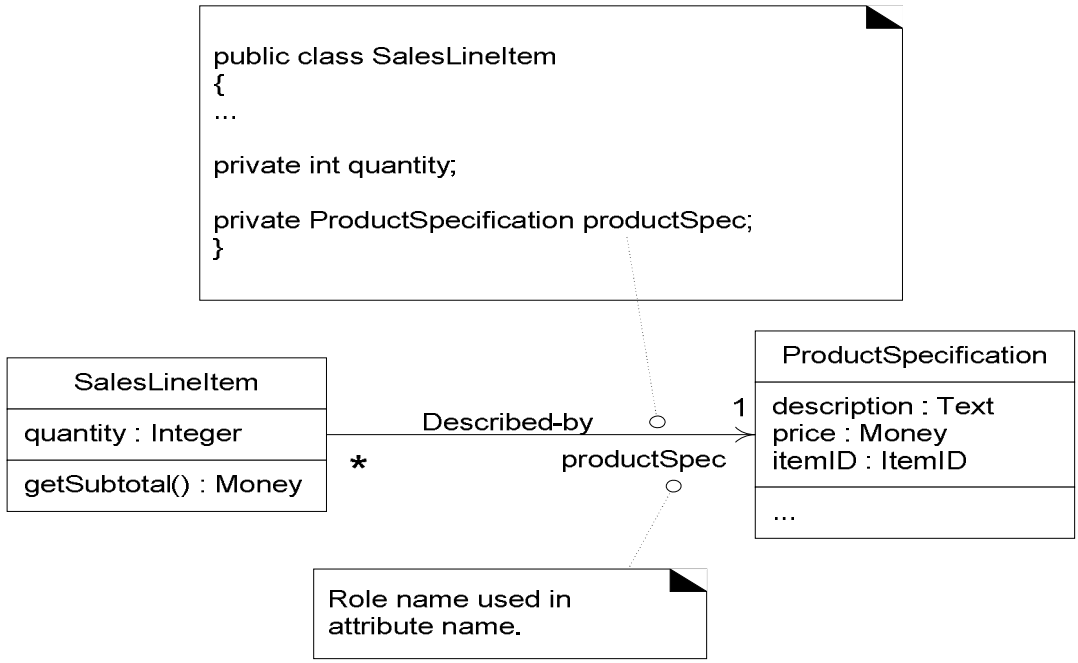


Fig: Role names may be used to generate instance variable names.

4.3.2.4 Mapping Attributes

The *Sale* class illustrates that in some cases one must consider the mapping of attributes from the design to the code in different languages. Figure fig illustrates the problem and its resolution.

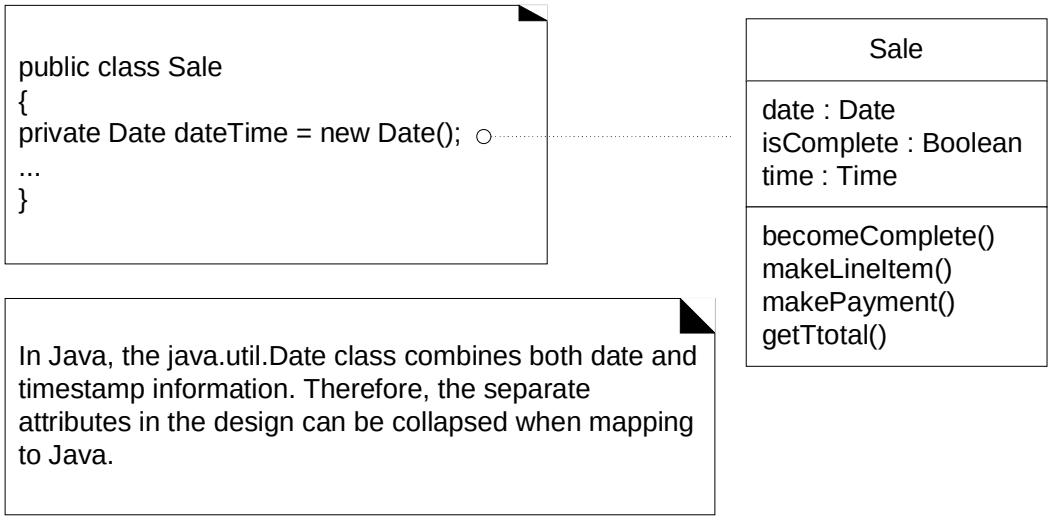


Fig: ***Mapping date and time to Java.***

4.4 Creating Methods from Interaction Diagrams

The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions. The *enterItem* interaction diagram in Figure 4.2 illustrates the Java definition

of the *enterItem* method. For this example, we will explore the implementation of the *Register* and its *enterItem* method. A Java definition of the *Register* class is shown in Figure

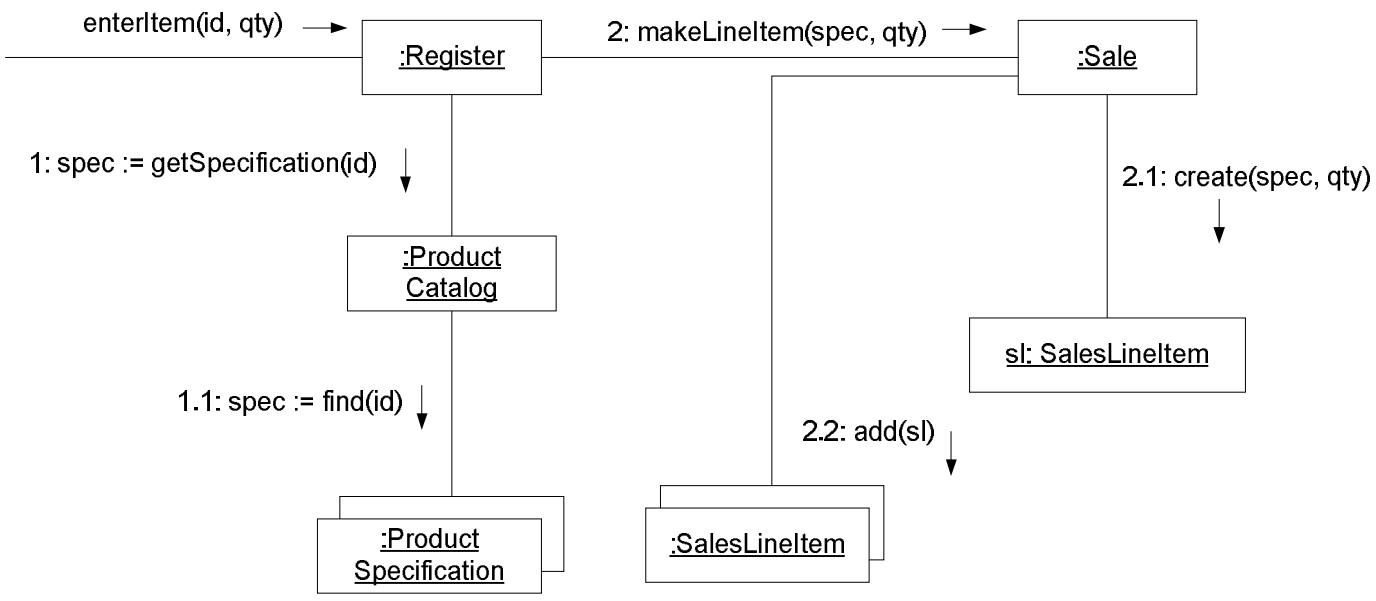


Fig 4.2 : **The enterItem interaction diagram.**

The *enterItem* message is sent to a *Register* instance; therefore, the *enterItem* method is defined in class *Register*.

public void enterItem(ItemID itemID, int qty)

Message 1: A *getProductDescription* message is sent to the *ProductCatalog* to retrieve a *ProductDescription*.

ProductDescription desc = catalog.getProductDescription(itemID);

Message 2: The *makeLineItem* message is sent to the *Sale*.

currentSale.makeLineItem(desc, qty);

In summary, each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method. The complete *enterItem* method and its relationship to the interaction diagram is shown in Figure 4.3

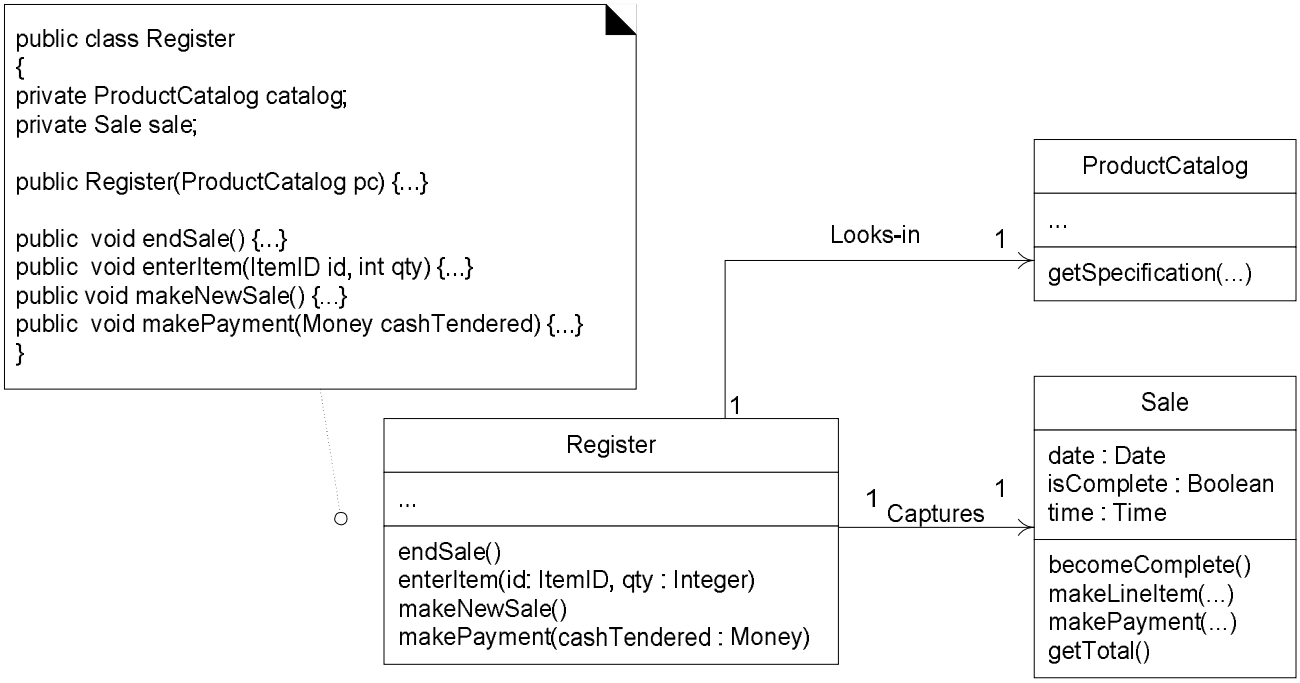


Fig: **The Register class.**

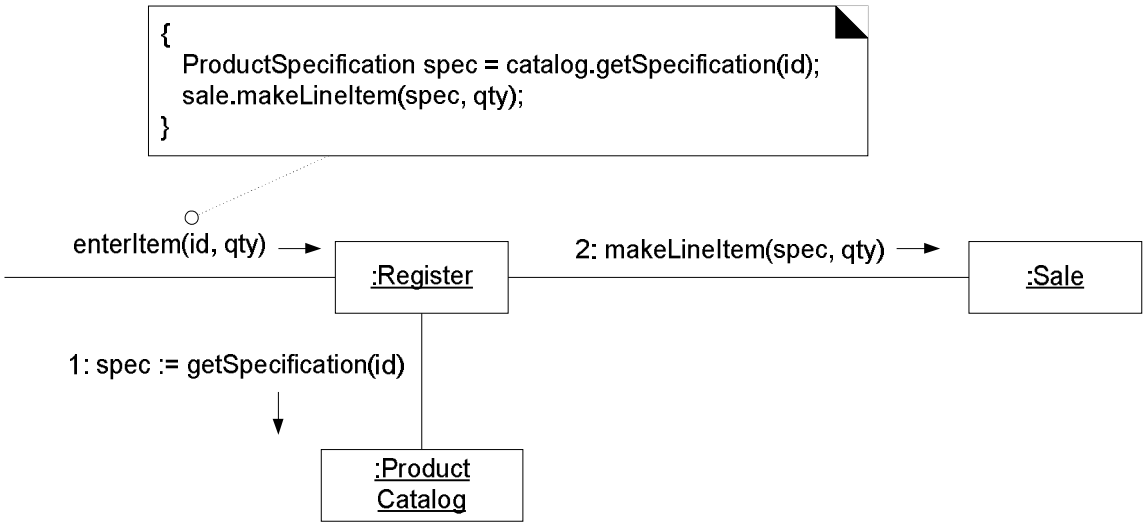


Fig: **The enterItem method.**

4.5 Collection Classes in Code

A collection is a group of data manipulate as a single object. Corresponds to a bag.

- Collection classes insulate client programs from the implementation. Eg. array, linked list, hash table, balanced binary tree
- Insulate client programs from the implementation.n array, linked list, hash table, balanced binary tree

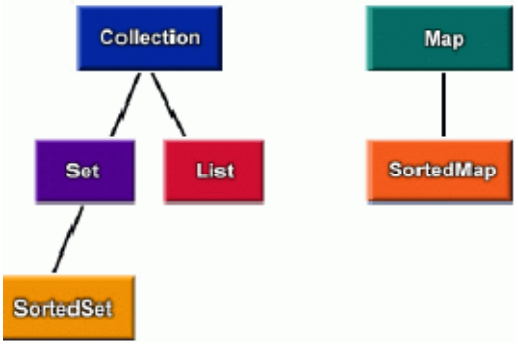
- Like C++'s Standard Template Library (STL)
- Can grow as necessary.
- Contain only Objects (reference types).
- Heterogeneous.
- Can be made thread safe (concurrent access).
- Can be made not-modifiable.

In real worlds ,One-to-many relationships are common. For example, a *Sale* must maintain visibility to a group of many *SalesLineItem* instances, as shown in Figure 4.5. In OO programming languages, these relationships are usually implemented with the introduction of a **collection** object, such as a *List* or *Map*, or even a simple array.

Collection Interfaces

Collections are primarily defined through a set of interfaces. Supported by a set of classes that implement the interfaces. Interfaces are used of flexibility reasons:

- Programs that uses an interface is not tightened to a specific implementation of a collection.
- It is easy to change or replace the underlying collection class with another (more efficient) class that implements the same interface.



Collection Advantages and Disadvantages

Advantages

- Can hold different types of objects.
- Resizable

Disadvantages

- Must cast to correct type
- Cannot do compile-time type checking.

Example :

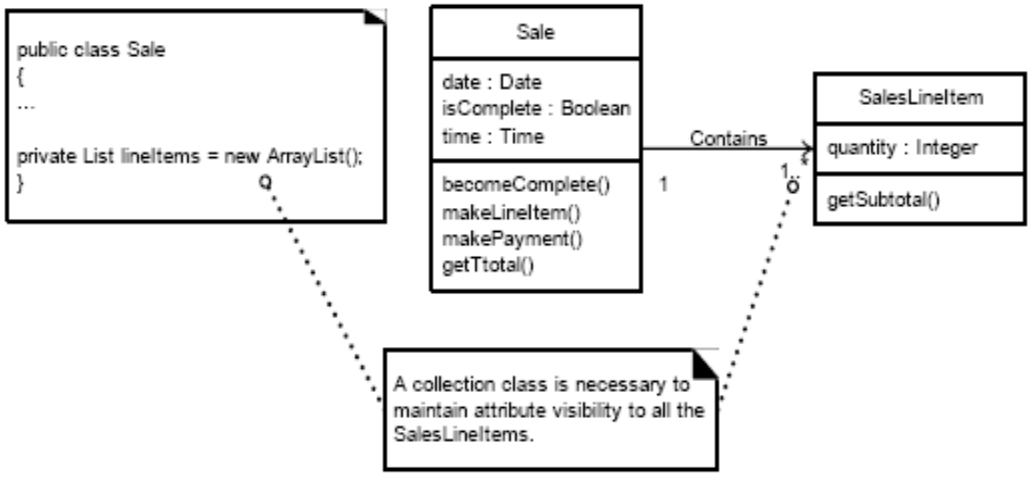


Fig: Adding Collection

For example, the Java libraries contain collection classes such as *ArrayList* and *HashMap*, which implement the *List* and *Map* interfaces, respectively. Using *ArrayList*, the *Sale* class can define an attribute that maintains an ordered list of *SalesLineItem* instances.

The choice of collection class is of course influenced by the requirements; key-based lookup requires the use of a *Map*, a growing ordered list requires a *List*, and so on.

As a small point, note that the *lineItems* attribute is declared in terms of its interface.

Guideline: If an object implements an interface, declare the variable in terms of the interface, not the concrete class.

For example, in Figure the definition for the *lineItems* attribute demonstrates this guideline:

private List lineItems = new ArrayList();

Collections in Java

- Arrays
  - Has special language support
- Iterators
  - `Iterator` (i)
- Collections (also called containers
  - `Collection` (i)
  - `Set` (i),
    - `HashSet` (c), `TreeSet` (c)
  - `List` (i),
    - `ArrayList` (c), `LinkedList` (c)
  - `Map` (i),
    - `HashMap` (c), `TreeMap` (c)

ArrayList :

- is an array based implementation where elements can be accessed directly via the **get** and **set** methods.
- Default choice for simple sequence.

LinkedList

- is based on a double linked list n Gives better performance on **add** and **remove** compared to **ArrayList**.
- Gives poorer performance on **get** and **set** methods compared to **ArrayList**.

Note: The classes ArrayList and LinkedList implement the List interface.

Difference between Array and Collection

Array

- Holds objects of known type.
- Fixed size.

Collections

- Generalization of the array concept.
- Set of interfaces defined in Java for storing object.
- Multiple types of objects.
- Resizable.

Example : Use of LinkedList Collection type

`LinkedList`, Example

```
import java.util.*;
public class MyStack {
    private LinkedList list = new LinkedList();
    public void push(Object o){
        list.addFirst(o);
    }
    public Object top(){
        return list.getFirst();
    }
    public Object pop(){
        return list.removeFirst();
    }

    public static void main(String args[]) {
        Car myCar;
        MyStack s = new MyStack();
        s.push (new Car());
        myCar = (Car)s.pop();
    }
}
```

4.6 Exceptions and Error Handling

Exception handling has been ignored so far in the development of a solution. This was intentional to focus on the basic questions of responsibility assignment and object design. However, in application development, it's wise to consider the large-scale exception handling strategies during design modeling (as they have a large-scale architectural impact), and certainly during implementation. Briefly, in terms of the UML, exceptions can be indicated in the property strings of messages and operation declarations.

An exception is a condition that is caused by a runtime error in the program. Provide a mechanism to signal errors directly without using flags. Allow errors to be handled in one central part of the code without cluttering code. An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.

- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works, you need to understand the three categories of exceptions:

Checked exceptions: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

Runtime exceptions: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

When the JVM(Java Virtual Machine) encounters an error such as divide by zero, it creates an exception object and throws it – as a notification that an error has occurred. I f the exception object is not caught and handled properly; the interpreter will display an error and terminate the program. If we want the program to continue with execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then take appropriate corrective actions. This task is known as exception handling.

Some Common Example of Exceptions:

- ArithmeticException
- ArrayIndexOutOfBoundException
- ArrayStoreException
- FileNotFoundException
- IOException – general I/O failure
- NullPointerException – referencing a null object
- OutOfMemoryException
- SecurityException – when applet tries to perform an action not allowed by the browser's security setting.
- StackOverflowException
- StringIndexOutOfBoundException

Exception Handling Process

- A method can signal an error condition by throwing an exception **– throws**
- The calling method can transfer control to a exception handler by catching an exception - **try, catch**
- Clean up can be done by **– finally**

- Try block, code that could have exceptions errors
- Catch block(s), specify code to handle various types of exceptions. First block to have appropriate type of exception is invoked.
- If no 'local' catch found, exception propagates up the method call stack, all the way to main()
- Any execution of try, normal completion, or catch then transfers control on to finally block

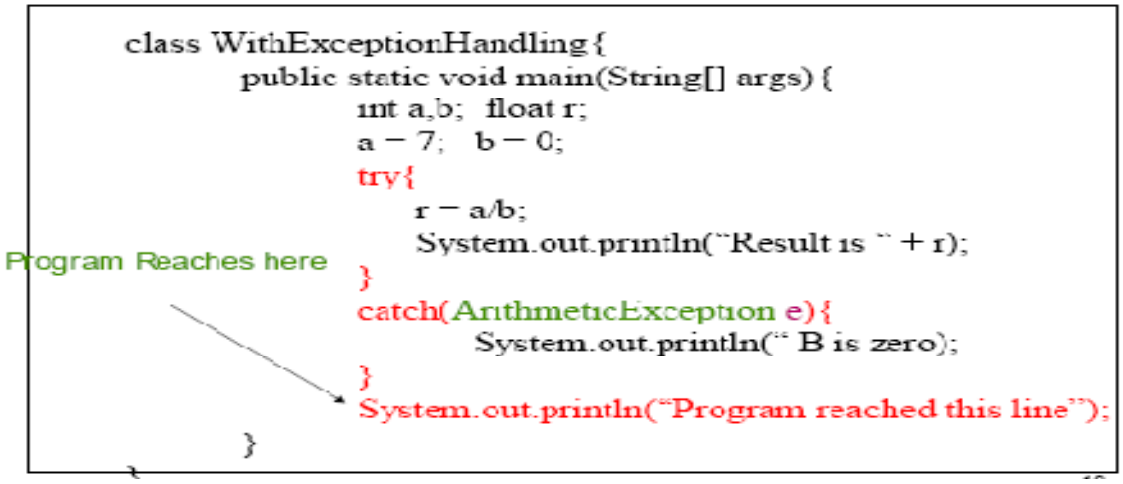


Fig: Example of Exception Handling

```
class WithExceptionCatchThrowFinally{
        public static void main(String[] args){
                int a,b;  float r;  a = 7;   b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
Program reaches here  catch(ArithmeticException e){
                        System.out.println(" B is zero);
                        throw e;
                }
                finally{
                 System.out.println("Program is complete");
                }
        }
}
```

Fig: Example of Exception Handling with finely

Example of Exception Hierarchy in Java

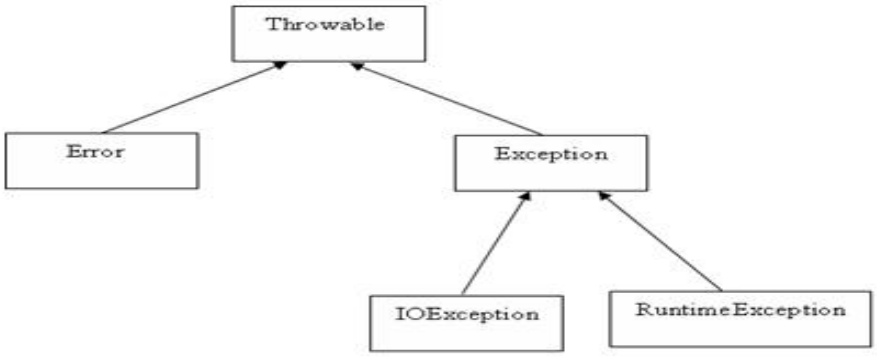


Fig: Exception class hierarchy in java

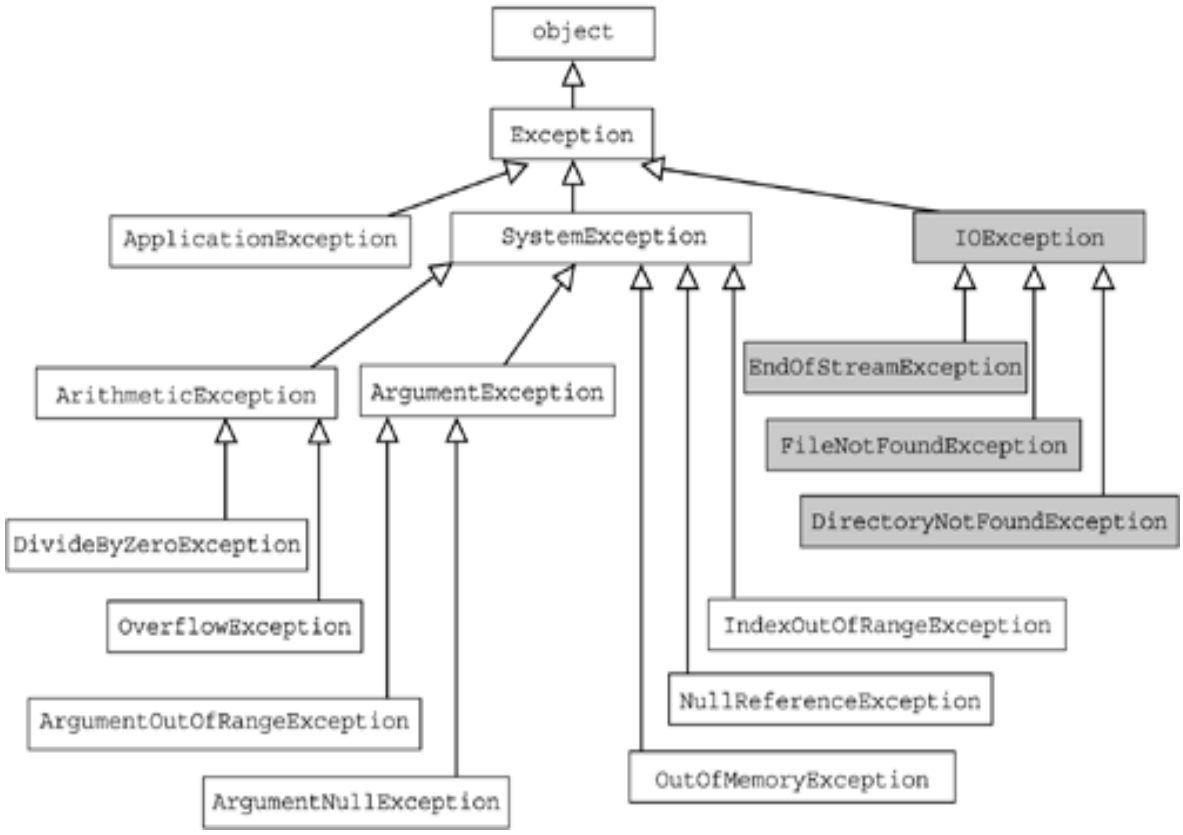


Fig: Exception class hierarchy in C#

Declaring you own Exception(User defined Exceptions):

You can create your own exceptions. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throw able.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

You just need to extend(inherit) the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

```
Example of user defined exception in java:
class MyException extends Exception
{
}
```

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

The fundamentals of exception handling

- The "normal" code is put in try block. It means that we "try to execute code" in the try block.
- If the system succeeds to run the code, everything is fine (execution goes in order from top to down, catch blocks are skipped).
- If something goes wrong when code of try block is executed, this code throws an exception object and stops executing the code of try block further.
- Another part of the code (the error handling part) catches the exception (object) and make necessary actions needed in that error situation. Execution continues with the next statement following the catch blocks
- The exception object can contain information about the exception, so that the error handling part of the program can examine the reason and make appropriate actions.

Summary of Exception Handling

- A good programs does not produce unexpected results.
- I t is always a good practice to check for potential problem spots in programs and guard against program failures.
- Exceptions are mainly used to deal with runtime errors.
- Exceptions also aid in debugging programs.

- Exception handling mechanisms can effectively used to locate the type and place of errors.

Order of Implementation

Classes need to be implemented (and ideally, fully unit tested) from least-coupled to most-coupled (see Figure). For example, possible first classes to implement are either Payment or ProductDescription; next are classes only dependent on the prior implementations—ProductCatalog or SalesLineItem.

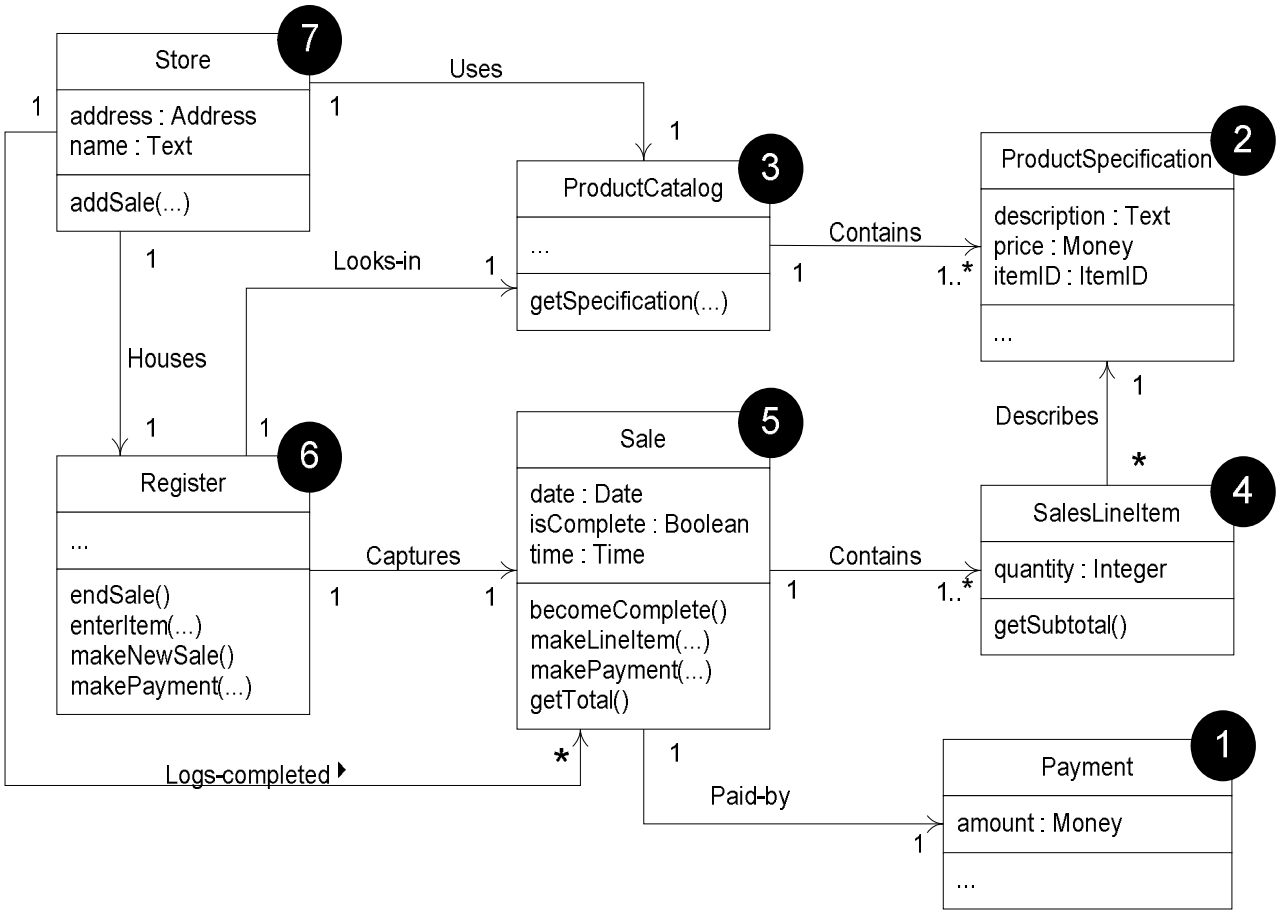


Fig: Possible order of class implementation and testing.

Implementation Code For Case Study

This section presents a sample domain object layer program solution in Java for this iteration. The code generation is largely derived from the design class diagrams and interaction diagrams defined in the design work, based on the principles of mapping designs to code as previously explored.

Introduction to the NextGen POS Program Solution

Class Payment

```
// all classes are probably in a package named
// something like:
package com.foo.nextgen.domain;
public class Payment
{
private Money amount;
public Payment( Money cashTendered ){ amount = cashTendered; }
public Money getAmount() { return amount; }
}
```

Class ProductCatalog

```
public class ProductCatalog
{
private Map<ItemID, ProductDescription>
descriptions = new HashMap()<ItemID, ProductDescription>;
public ProductCatalog()
{
// sample data
ItemID id1 = new ItemID( 100 );
ItemID id2 = new ItemID( 200 );
Money price = new Money( 3 );
ProductDescription desc;
desc = new ProductDescription( id1, price, "product 1" );
descriptions.put( id1, desc );
desc = new ProductDescription( id2, price, "product 2" );
descriptions.put( id2, desc );
}
public ProductDescription getProductDescription( ItemID id )
{
return descriptions.get( id );
}
}
```

Class Register

```
public class Register
{
private ProductCatalog catalog;
private Sale currentSale;
public Register( ProductCatalog catalog )
{
this.catalog = catalog;
}
public void endSale()
```

```
{
currentSale.becomeComplete();
}
public void enterItem( ItemID id, int quantity )
{
ProductDescription desc = catalog.getProductDescription( id );
currentSale.makeLineItem( desc, quantity );
}
public void makeNewSale()
{
currentSale = new Sale();
}
public void makePayment( Money cashTendered )
{
currentSale.makePayment( cashTendered );
}
}
```

Class ProductDescription

```
public class ProductDescription
{
private ItemID id;
private Money price;
private String description;
public ProductDescription( ItemID id, Money price, String description )
{
this.id = id;
this.price = price;
this.description = description;
}
public ItemID getItemID() { return id; }
public Money getPrice() { return price; }
public String getDescription() { return description; }
}
```

Class Sale

```
public class Sale
{
private List<SalesLineItem> lineItems =
new ArrayList()<SalesLineItem>;
private Date date = new Date();
private boolean isComplete = false;
private Payment payment;
public Money getBalance()
{
return payment.getAmount().minus( getTotal() );
}
public void becomeComplete() { isComplete = true; }
```

```
public boolean isComplete() { return isComplete; }
public void makeLineItem
( ProductDescription desc, int quantity )
{
lineItems.add( new SalesLineItem( desc, quantity ) );
}
public Money getTotal()
{
Money total = new Money();
Money subtotal = null;
for ( SalesLineItem lineItem : lineItems )
{
subtotal = lineItem.getSubtotal();
total.add( subtotal );
}
return total;
}
public void makePayment( Money cashTendered )
{
payment = new Payment( cashTendered );
}
}
```

Class SalesLineItem

```
public class SalesLineItem
{
private int quantity;
private ProductDescription description;
public SalesLineItem (ProductDescription desc, int quantity )
{
this.description = desc;
this.quantity = quantity;
}
public Money getSubtotal()
{
return description.getPrice().times( quantity );
}
}
```

Class Store

```
public class Store
{

private ProductCatalog catalog = new ProductCatalog();

private Register register = new Register( catalog );

public Register getRegister() { return register; }

}
```

Introduction to the Monopoly Program Solution

This section presents a sample domain layer of classes in Java for this iteration. Iteration-2 will lead to refinements and improvements in this code and design. Comments excluded on purpose, in the interest of brevity, as the code is simple.

Class Square

```
// all classes are probably in a package named
// something like:
package com.foo.monopoly.domain;
public class Square
{
private String name;
private Square nextSquare;
private int index;
public Square( String name, int index )
{
this.name = name;
this.index = index;
}
public void setNextSquare( Square s )
{
nextSquare = s;
}
public Square getNextSquare( )
{
return nextSquare;
}
public String getName( )
{
return name;
}
public int getIndex()
{
return index;
}
}
```

Class Piece

```
public class Piece
{
private Square location;
public Piece(Square location)
{
this.location = location;
}
```

```
public Square getLocation()
{
return location;
}
public void setLocation(Square location)
{
this.location = location;
}
}
```

Class Die

```
public class Die
{
public static final int MAX = 6;
private int faceValue;
public Die( )
{
roll( );
}
public void roll( )
{
faceValue = (int) ( ( Math.random( ) * MAX ) + 1 );
}
public int getFaceValue( )
{
return faceValue;
}
}
```

Class Board

```
public class Board
{
private static final int SIZE = 40;
private List squares = new ArrayList(SIZE);
public Board()
{
buildSquares();
linkSquares();
}
public Square getSquare(Square start, int distance)
{
int endIndex = (start.getIndex() + distance) % SIZE;
return (Square) squares.get(endIndex);
}
public Square getStartSquare()
{
return (Square) squares.get(0);
}
```

```
private void buildSquares()
{
for (int i = 1; i <= SIZE; i++)
{
build(i);
}
}
private void build(int i)
{
Square s = new Square("Square " + i, i - 1);
squares.add(s);
}
private void linkSquares()
{
for (int i = 0; i < (SIZE - 1); i++)
{
link(i);
}
Square first = (Square) squares.get(0);
Square last = (Square) squares.get(SIZE - 1);
last.setNextSquare(first);
}
private void link(int i)
{
Square current = (Square) squares.get(i);
Square next = (Square) squares.get(i + 1);
current.setNextSquare(next);
}
}
```

Class Player

```
public class Player
{
private String name;
private Piece piece;
private Board board;
private Die[] dice;
public Player(String name, Die[] dice, Board board)
{
this.name = name;
this.dice = dice;
this.board = board;
piece = new Piece(board.getStartSquare());
}
public void takeTurn()
{
// roll dice
```

```
int rollTotal = 0;
for (int i = 0; i < dice.length; i++)
{
dice[i].roll();
rollTotal += dice[i].getFaceValue();
}
Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
piece.setLocation(newLoc);
}
public Square getLocation()
{
return piece.getLocation();
}
public String getName()
{
return name;
}
}
```

Class MonopolyGame

```
public class MonopolyGame
{
private static final int ROUNDS_TOTAL = 20;
private static final int PLAYERS_TOTAL = 2;
private List players = new ArrayList( PLAYERS_TOTAL );
private Board board = new Board( );
private Die[] dice = { new Die(), new Die() };
public MonopolyGame( )
{
Player p;
p = new Player( "Horse", dice, board );
players.add( p );
p = new Player( "Car", dice, board );
players.add( p );
}
public void playGame( )
{
for ( int i = 0; i < ROUNDS_TOTAL; i++ )
{
playRound();
}
}
public List getPlayers( )
{
return players;
```

```
}
private void playRound( )
{
for ( Iterator iter = players.iterator( ); iter.hasNext( ); )
{
Player player = (Player) iter.next();
player.takeTurn();
}
}
}
```

Software Crisis

It was in late 1960's

- Many software projects failed.
- Many software projects late, over budget, providing unreliable software that is expensive to maintain.
- Many software projects produced software which did not satisfy the requirements of the customer.
- Complexities of software projects increased as hardware capability increased.
- Larger software system is more difficult and expensive to maintain.
- Demand of new software increased faster than ability to generate new software.

All the above attributes of what was called a 'Software Crisis'. So the term 'Software Engineering' first introduced at a conference in late 1960's to discuss the software crisis.

Software Quality Metrics for Object-Oriented Environments

Object-oriented software development requires a different approach from traditional development methods, including the metrics used to evaluate the software. With object-oriented analysis and design methods gaining popularity, it is time to investigate object-oriented quality metrics. Since metrics should never be developed in a void, this article first looks at criteria for the metrics, then discusses specific metrics for object-oriented development, including traditional metrics and metrics developed to measure specific object-oriented structures.

Object-oriented design and development are popular concepts in today's software development environment--some even herald them as the "silver bullet" for solving software problems. Although there is no silver bullet, object-oriented (OO) development has proved its value for

systems that must be maintained and modified. OO software development requires a different approach from more traditional functional decomposition and data flow development methods, including the metrics used to evaluate OO software.

The concepts of software metrics are well established, and many metrics relating to product quality have been developed and used. With OO analysis and design methods gaining popularity, it is time to start investigating OO metrics with respect to software quality. In this article, we answer the following questions:

What concepts and structures in OO design affect the quality of the software?

Can traditional metrics measure the critical OO structures?

If so, are the threshold values for the metrics the same for OO designs as for functional or data designs?

Which of the many new metrics found in the literature are useful to measure the critical concepts of OO structures?

Metric Evaluation Criteria

Traditional functional decomposition metrics and data analysis design metrics measure the design structure or data structure independently. However, OO metrics must be able to treat function and data as a combined, integrated object . To evaluate a metric's usefulness as a quantitative measure of software quality, it must be based on the measurement of a software quality attribute. The metrics selected, however, are useful in a wide range of models. The OO metric criteria, therefore, are to be used to evaluate the following attributes:

Efficiency -- are the constructs efficiently designed?

Complexity -- could the constructs be used more effectively to decrease the architectural complexity?

Understandability -- does the design increase the psychological complexity?

Reusability -- does the design quality support possible reuse?

Testability and maintainability -- does the structure support ease of testing and changes?

Whether a metric is traditional or new, it must effectively measure one or more of these attributes. As each metric is presented, we will briefly discuss its applicability.

The Software Assurance Technology Center's (SATC) approach was to select OO metrics that apply to the primary, critical constructs of OO design. The suggested metrics are supported by most literature and are now found in some OO tools. The metrics evaluate the OO concepts: methods, classes, cohesion, coupling, and inheritance. The metrics focus on internal object

structure, external measures of the interactions among entities, measures of the efficiency of an algorithm and the use of machine resources, and the psychological measures that affect a programmer's ability to create, comprehend, modify, and maintain software.

We support the use of three traditional metrics and present six additional metrics specifically for OO systems. The SATC has found that there is considerable disagreement in the field about software quality metrics for OO systems , with some who contend that traditional metrics are inappropriate for OO systems. However, there are valid reasons to apply traditional metrics when it can be done. The traditional metrics have been widely used, are well understood by researchers and practitioners, and their relationships to software quality attributes have been validated. Preceding each metric, a brief description of the OO structure is given. Each metric is then described, interpretation guidelines given, and the applicable quality attributes listed.

Traditional Metrics

Methods

In an OO system, traditional metrics are generally applied to the methods that comprise the operations of a class. A method is a component of an object that operates on data in response to a message and is defined as part of the declaration of a class. It is an operation upon an object and is defined in the class declaration. Methods reflect how a problem is broken into segments and the capabilities other classes expect of a given class.

Metric 1: Cyclomatic Complexity. Cyclomatic complexity is used to evaluate the complexity of an algorithm in a method. Low cyclomatic complexity methods are generally better, although some are low because decisions are deferred through message passing, not because the method is not complex. Because of inheritance, cyclomatic complexity cannot be used to measure the complexity of a class, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class.

Generally, the cyclomatic complexity for a method should be below 10, which indicates that decisions are deferred through message passing. Although this metric is specifically applicable to evaluation of the complexity quality attribute, it also is related to all the other attributes

Metric 2: Size. Size of a method is used to evaluate the ease of understandability of the code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, and the number of blank lines. Thresholds for evaluating the size measures vary, depending on the coding language used and the complexity of the method. However, because size affects ease of understanding, large-size

routines always pose a higher risk in the attributes of understandability, reusability, and maintainability

Metric 3: Comment Percentage. The line counts done to compute size metrics can be expanded to include a count of the number of comments, both on line (with code) and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. The SATC found that a comment percentage of about 30 percent is most effective. Because comments help developers and maintainers, this metric is used to evaluate the attributes of understandability, reusability, and maintainability

OO-Specific Metrics

As discussed, many different metrics have been proposed for OO systems. The OO metrics chosen by the SATC measure principle structures that, if improperly designed, negatively affect the design and code quality attributes.

The selected OO metrics are primarily applied to the concepts of classes, coupling, and inheritance. Multiple definitions are given for some of the OO metrics discussed here, because researchers and practitioners have not reached a common definition or counting method. In some cases, the counting method for a metric is determined by the software analysis package used to collect the metrics.

Classes

A class is a template from which objects can be created. This set of objects shares a common structure and a common behavior manifested by the set of methods. Three class metrics described here measure the complexity of a class using the class's methods, messages, and cohesion.

Metric 4: Weighted Methods per Class (WMC). The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement because not all methods are accessible within the class hierarchy because of inheritance.

The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children, since children inherit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This metric measures understandability, reusability, and maintainability

Message

A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. The next metric looks at methods and messages within a class.

Metric 5: Response for a Class (RFC). The RFC is the cardinality of the set of all methods that can be invoked in response to a message sent to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric uses a number of methods to review a combination of a class's complexity and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class.

If a large number of methods can be invoked in response to a message, testing and debugging the class requires a greater understanding on the part of the tester. A worst-case value for possible responses assists in the appropriate allocation of testing time. This metric evaluates understandability, maintainability, and testability

Cohesion

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective OO designs maximize cohesion because they promote encapsulation.

Metric 6: Lack of Cohesion of Methods (LCOM). LCOM uses data input variables or attributes (structural properties of classes) to measure the degree of similarity between methods. Any measure of method separateness helps identify flaws in the design of classes. There are at least two ways to measure cohesion:

For each data field in a class, calculate the percentage of methods that use that data field. Average the percentages, then subtract from 100 percent. Lower percentages indicate greater data and method cohesion within the class.

Methods are more similar if they operate on the same attributes. Count the disjoint sets produced from the intersection of the sets of attributes used by the methods.

High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during development. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates efficiency and reusability

Coupling

Coupling is a measure of the strength of association established by a connection from one entity to another. Classes (objects) are coupled three ways:

When a message is passed between objects, the objects are said to be coupled.

Classes are coupled when methods declared in one class use methods or attributes from the other classes.

Inheritance introduces significant tight coupling between superclasses and their subclasses.

Since good OO design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling. The next OO metric measures coupling strength.

Metric 7: Coupling Between Object Classes (CBO). CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct noninheritance-related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class, the easier it is to reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design; maintenance is therefore more difficult. Strong coupling complicates a system, since a module is harder to understand, change, or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates efficiency and reusability.

Inheritance

Another design abstraction in OO systems is the use of inheritance. Inheritance is a type of relationship among classes that enables programmers to reuse previously defined objects, including variables and operators. Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.

Metric 8: Depth of Inheritance Tree (DIT). The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods. A support metric for DIT is the number of methods inherited. This metric primarily evaluates efficiency and reuse but also relates to understandability and testability

Metric 9: Number of Children (NOC). The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper parent abstraction, and it may be an indication of subclassing misuse.

But the greater the number of children, the greater the reusability, since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increase the testing time. NOC, therefore, primarily evaluates efficiency, reusability, and testability