Coverage of Lectures

1. **Design of Classes for Test and Validation**
    - **Preconditions**
    - **Assert**
    - **Exceptions and Errors**
2. **Design of Test Scenarios**
    - **Test Coverage**
    - **Test Strategy**
3. **Design and Implementation of Main Programs**
    - **Test/Validation mains**
    - **Production mains**
4. **Minimizing the Effort of Test and Validation**
    - **Using files**

What is a correct program?

1. A program which returns “correct” outputs for all valid inputs.

2. A program which, when invoked with invalid inputs terminates gracefully and returns diagnostic information.

3. A program which completes in a “timely” manner.

How do we determine if a program is correct?

1. Correctness by testing - Execute the program for each valid set of inputs and determine that it obtains the correct answer.

Why is this difficult?

2. Correctness by formal proof. - Determine by analysis that the program implements its specifications.

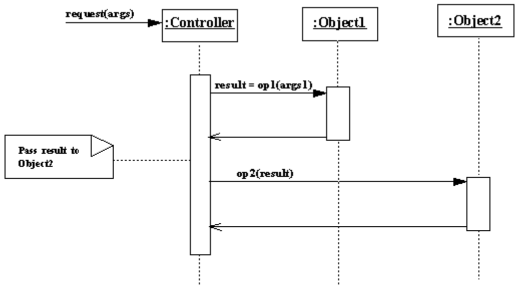
Why is this difficult?

Program Structure

An object-oriented program consists of a main program and a set of interacting instances of a set of classes. The main program instantiates an initial set of class instances and initiates execution of the interactions.

Execution Model

Execution of an object-oriented program is a sequence of interactions among some set of object instances.



UML Sequence Diagrams diagrammatically show these interactions.

Designing of Classes for Test and Validation

1. **Construct state models for classes with complex behavior. (For control flow driven systems.)**
    - **State models assist in defining preconditions and post conditions.**
    - **State models identify the methods to be implemented.**
2. **Construct state models for classes which must process externally generated events.**
    - **State models determine whether the sequence of invocations of methods are correct.**
    - **Apply state table matching for validation of event occurrences.**
3. **Specify value ranges for instance variables where possible. Enforce these ranges in the implementation of the class constructors.**
4. **Use the narrowest possible access control. Provide accessor and mutator methods as needed.**

Design and Implementation of Methods for Test and Validation

1. Write pre- and post- conditions for each method as comments as a part of the method header.

2. Use The Assert class to validate preconditions when they specifiable as conditional expressions.

3. Use exceptions to monitor specific conditions such as range errors, etc.

The Coin Puzzle Problem

Specification
Object Model
State Model
Interface Definition
Implementation Design
Verification and Validation Design
Implementation and Test

This lecture is a case study for the project in that it illustrates most of the phases of the project with an emphasis on verification and validation.

Verification and Validation begin with object model definition and have some activity at each stage of development.

The Coin Puzzle Problem - A puzzle board has five slots in a line. The slots are numbered from left to right with the numbering beginning with zero. Four coins, a dime, a penny, a nickel and a quarter, are placed on four of the slots.

0	1	2	3	4
25	5	1	10	

0	1	2	3	4
25	1	5	10	

Possible initial states of the puzzle.

A quarter is physically bigger than a nickel which is physically bigger than a penny which is physically bigger than a dime. The goal of the puzzle is to rearrange the coins so that the quarter is in the rightmost or number 4 slot, the nickel is in the number 3 slot, the penny is in the number 2 slot and the dime is in the number 1 slot using only legal moves.

Legal moves are:
a) A coin can be moved into any open slot.
b) A coin can be placed on top of any physically larger coin.
c) Only the coin on the top of a stack of coins can be moved.

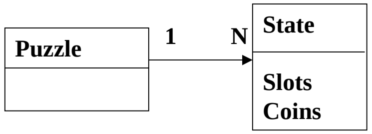
There may be multiple legal moves from a given initial and/or intermediate state. Therefore there may be several different sequences of moves which can put the puzzle in the solution state from any given initial state.

The goals for this project are to write a program which solves the coin puzzle for any initial configuration, records the sequence of moves taken to obtain the solution and which can be used to determine minimum number of moves which can lead to a solution from a given initial state.

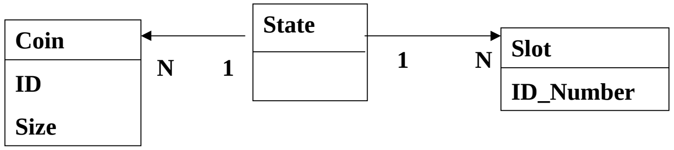
0	1	2	3	4
	10	1	5	25

The solution state for the puzzle.

Object Model for the Coin Puzzle.

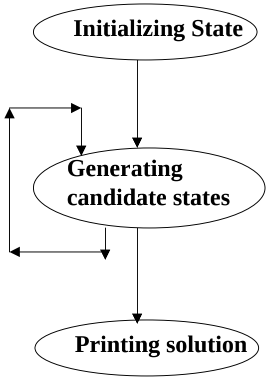


Both of these models are OK. But since Coin and Slot are both simple objects which interact together we will use the model on the left for State.

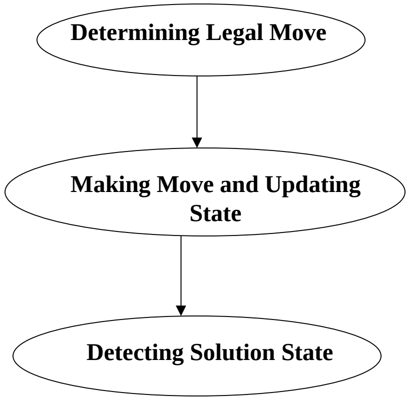


For the sake of convenience let us note that Slot.ID_Number and Coin.Size will be small ints. Dime = 0, Penny = 1, etc.

State Model for Coin Puzzle Object



State Model for State Object



Actions and Methods

<u>CoinPuzzle</u>

a) Initialize

b) Request move

c) Select move

d) Record State

e) Print solution

<u>State</u>

a) Find valid move

Is a coin free to move?

Is there a neighboring slot into which a “free” coin “fits.”

b) Make move - update state of puzzle

Create a new instance of State and update this instance with coin move.

c) Determine if puzzle is in solution state.

d) Print move sequence leading from initial to solution state.

<u>State</u>

a) Find valid move

Is a coin free to move?

Is there a neighboring slot into which a “free” coin “fits.”

```
public boolean validMove(int coin, int direction)     /*
       pre: State.DIME <= coin <= State.QUARTER */
       /* direction = State.left or State.right */
       /* post: returns true if coin can be moved in the
              desired direction */
```

b) Make move - update state of puzzle

Create a new instance of State and update this instance with coin move.

```
 public State move(int coin, int direction)
       /* pre: coin and direction describe valid move.*/
       /* post: coin is moved in that direction. */
```

c) Determine if puzzle is in solution state.

```
public boolean done()
       /* Invoking instance of State is a valid state. */
       /* post: returns true if state is the finish
              state.*/
```

d) Print move sequence leading from initial to solution state.

```
public int printModes()
```

```
       /* States have been added to FIFO queue with this
              state at tail of queue. */
       /* post: print moves up to and including this
              point. */
```

Puzzle

Puzzle needs only a main program which sequences invocations of methods which create and modify instances of State and which invokes methods defined by State.

Specifications for a Cheap Microwave Oven

This microwave oven has a powertube, a light, a door, a timer, and just one button. If the door is closed and the oven is not cooking then pushing the button causes the light and the powertube to be turned on for a period of one minute. If the button is pushed while the oven is cooking then the cooking period is extended for one more minute. If the door is opened while the oven is cooking then the cooking period is suspended. If the door is closed while cooking is suspend then cooking is resumed with the duration being the time left on the timer when the door was opened plus one minute. If the button is pushed while the door is open any time remaining on the cooking period is terminated. If a cooking period terminates by the expiration of the timer then the oven becomes idle.

What are the objects?
- microwave oven - tangible
- powertube - tangible
- light - tangible
- timer -tangible
- button - tangible
- door - tangible

The powertube, the light, the timer, the button and the door are all components of the microwave oven.

The powertube and light objects have two events and two states, on and off and thus will need only on and off methods so we need not consider them any further.

The timer will be a system primitive so we need not consider it and further. We will use whatever behavior and interface the system timer provides.

The door and the button, however, generate <u>events</u> or actions which will be processed by the oven.

So only the oven has complex behavior and thus significant software to be implemented.

Attributes for Object MicrowaveOven

1. Components as Attribute.
Each of the components, powertube, light, beeper, timer, button and door will be attributes of the oven and instance variables of the resulting class.
2. Status Attribute - We need an attribute to record the state of the oven. Let us call this attribute "status" or "current state." In fact all objects with more than one or two actions will need a status variable.

Attributes for Other Objects:
1. Status - on or off
2. Unique ID

Timer

We need to know only the interface of the system supplied timer object.

Event/action - An event is a result of some action defined at the object level which results in a interaction among objects or within an object. An event always causes some other action to occur.

1. **Pushing the button creates an event.**
2. **Opening the door creates an event.**
3. **Closing the door creates an event.**
4. **The timer expiring creates an event.**

Every externally generated event that is directed to an object requires a public method in the implementing class to execute the action required by the event. We know that we will have at least 4 public methods in the class MicrowaveOven.

There is a Java class Event which has highly similar semantics to the definition for an event used here. It is a part of the java.awt package and is used for buttons, etc. in GUIs.

Example:

Pushing the button (when the door is closed and the oven is idle) causes:

a. The light to be turned on.
b. The powertube to be turned on
c. The timer to be set to expire in one minute.

This means that we need a method in class MicrowaveOven to execute this action.

Pushing the button when the oven is cooking causes:

a. One minute to be added to the duration before the timer expires.

Therefore we need a method to execute this action.

The action to be taken on generation of an event depends on the state of the object which is the target of the event. Which is why we need a status variable for each object with multiple possible states. We must select an appropriate action on the basis of the state of the object.

What should happen when the button is pushed when the door is open?

To realize the functionality of an object it is often necessary that events be generated in a specific order. This implies that the object will pass through a specific sequence of states while serving its purpose.

Opening the door and then pushing the button
is not a useful sequence of events.

Furthermore some events should be ignored or taken as errors if they occur when an object is in some specified state.

A state machine is used to specify the valid sequence of states. A state machine for an object is simply a sequence of states with the transitions between states being invoked by occurrence of events targeting the object. An action is associated with each state.

Scenario Analysis for the Microwave Oven

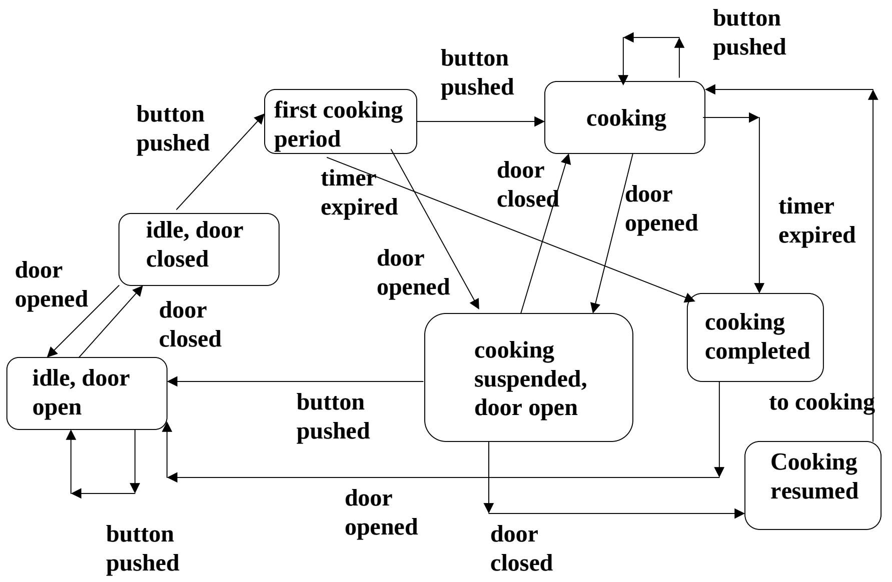
Precondition: Oven idle with door closed.

Initiator	Action	Part.	Service	Prior State	New State
user	push button	oven	turn on light and powertube, set timer for one minute, change oven state	idle, door closed	first cooking period
user	push button	oven	add one minute to timer	first cooking period	cooking
timer	expires	oven	turn off oven	cooking	cooking completed
user	open oven door	oven	change oven state	cooking completed	idle. door open

Scenario Analysis for Microwave Oven

Precondition: Oven idle with door closed.

Initiator	Action	Part.	Service	Prior State	NewState
user	push button	oven	turn on light and powertube, set timer for one minute, change oven state	idle, door closed	first cooking period
user	push button	oven	add one minute to timer	first cooking period	cooking
user	open door	oven	turn off light and powertube	cooking	idle, cooking suspended
user	close door	oven	turn on light, powertube	idle, cooking suspended	resume cooking
oven	return to cooking	oven	add minute to timer	resume cooking	cooking



Definition of Moore State Machines
(State Model)

1. Let us associate an action with each state. The action associated with the “first cooking period” state is to turn on the powertube and the light and to set the timer for one minute.

2. Let us stipulate that execution of the action will be initiated when the state is entered.

3. The first thing the action should do is to determine whether or not it should execute its functionality at all (evaluate the precondition), should ignore the event that caused the transition or report an exception or error.

4. The second thing the action associated with a given state does (assuming the precondition is satisfied) is to change the object instance state to the new state.

5. The action is then executed and the state machine halts and waits for the next action. (The implementing method returns to the invocation site and the class instance is left in the new state.)

	idle door closed	idle door open	first cooking period	cooking	cooking suspended, door open	cooking completed	Resume cooking
button pushed	First cooking period	ignore	cooking	cooking	idle door open	ignore	can’t happen
door opened	idle door open	can’t happen	Cooking suspended, door open	Cooking suspended, door open	can’t happen	idle door open	can’t happen
door closed	can’t happen	idle door closed	can’t happen	can’t happen	cooking	can’t happen	can’t happen
timer expires	can’t happen	can’t happen	idle door closed	idle door closed	can’t happen	can’t happen	can’t happen

Why bother with all of this rigmarole?

1. It is wonderful documentation.
    Pictures are much better than words.

2. It leads to a uniform implementation structure.
    All classes will have the same structure and anyone can read them.
    Furthermore you will be able to read them yourselves.

3. It is the basis for an algorithm for designing classes which will always work.

4. It eliminates wasted thought.
    If you don’t have to think about structure you can think about content.

5. It facilitates debugging.
    Preconditions are naturally defined.

6. The code will be robust to external errors.
    Errors are detected before they corrupt the program state.

How does this model of execution map to an implementation in Java?

1. Define a public method for each event. If the event has only one possible action then implement the action in this method.

2. If the action to be executed upon the occurrence of an event is state dependent then define a private method for each action and invoke the appropriate method for the current state of the object from the public method.

```
Public class MicrowaveOven
{           private Powertube powertube;
            private Light light;
            private Timer timer;
            private String currentState;
            public MicrowaveOven(String initialState)
            {           currentState = initialState;  }
/* The method signatures and implementations follow. */
}
```

```
public void buttonPushed() {
            if (currentState = “idle door closed”)
            then {this.firstCookingPeriod();
            return;}
            if (currentState =“ idle door open”)
            then {return;}
            if (currentState= “firstCookingPeriod”)
            then {this.cooking();
            return;}
            if (currentState = “cooking”)
            then {this.cooking();
            return;}
            if (currentState = “idle cooking suspended”)
            then {return;}
            /* Throw appropriate Exception. */
            return; }
```

```
public void doorOpened()
{           if (currentState = “idle door closed”)
            then {this.idleDoorOpened();
            return;}
            if (currentState = “first cooking period”)
            then {this.idleCookingSuspended();
            return; }
            if (currentState = “cooking”)
            then {this.idleCookingSuspended();
            return;}
            if (currentState= “idle door open”)
            then { /* Throw appropriate Exception.*/;
            return;}
}
```

```
public void doorClosed()
{           if (currentState = “idleDoorOpen”)
            then {this. IdleDoorClosed();
            return;}
            if (currentState = “cookingSuspended”)
            then {this.resumeCooking();
            return;}
            / Throw appropriate Exception.*/
            return;
}
```

```
            Public void TimerExpires()

            {           if (currentState = “first cooking period”)

                        then {this.cookingCompleted();

                        return;}

                        if (currentState = “cooking”)

                        then {this.cookingCompleted();

                        return;}

            }
}
```

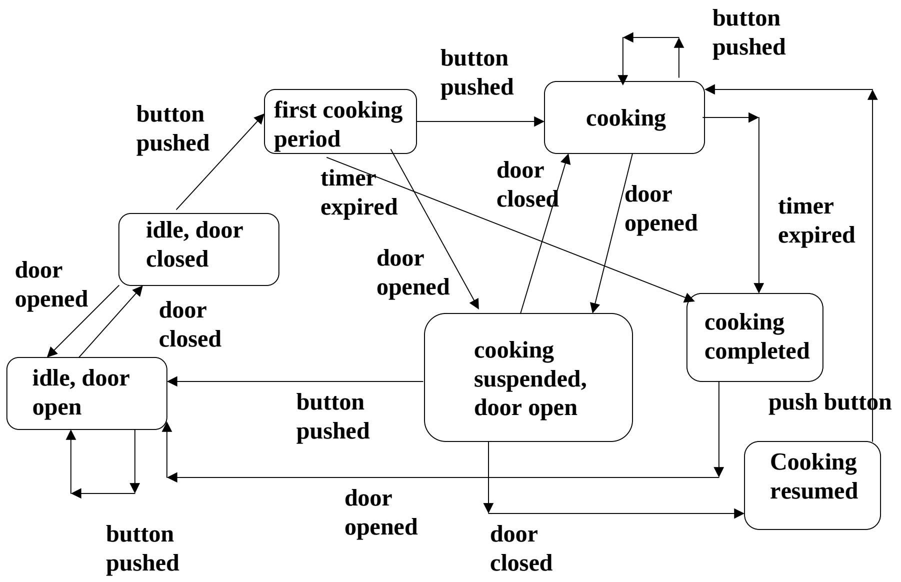
Private Methods - one for each state of the state model.

```
private void idleDoorClosed()
{ /* Set state to “idle door closed.” */ }
private void idleDoorOpen()
{/* Set state to “idle door open.” */
 /* Clear Timer.*/ }
private void firstCookingPeriod()
{ /* Set state to “first cooking period.” */
  /* Turn on powertube and light and set timer. */}
private void cooking()
{ /* Add one minute to timer. */ }
private void idleCookingSuspended()
{ /* Suspend powertube and light. }
private voidresumeCooking()
{ /* Turn on powertube and light.” }
private void cookingCompleted()
{/* Turn off powertube and light. */
```

Scenario Analysis for Microwave Oven

Precondition: Oven idle with door closed.

Initiator	Action	Part.	Service	Prior State	NewState
user	push button	oven	turn on light and powertube, set timer for one minute, change oven state	idle, door closed	first cooking period
user	push button	oven	add one minute to timer	first cooking period	cooking
user	open door	oven	turn off light and powertube	cooking	idle, cooking suspended
user	close door	oven	turn on light, powertube	idle, cooking suspended	resume cooking
user	push button	oven	add minute to timer	resume cooking	cooking



Use of the Assert Class to Validate Preconditions

Question: How does one detect the occurrence of a condition known to lead to an error.
Answer: Either explicitly check for the occurrence of the condition with an if statement or use the Assert class.

Question: Why use Assert instead of just adding if and print statements?
Answer: You want to obtain the context of the error. Use of Assert gives you a print out of the “stack trace.”

Question: What is the ‘stack trace?”
Answer: Follows in the next figures.

Invocation of a method requires recall of the origin of the call and space for storage of parameters and data local to the method.

Instances of execution of methods are stored on the system stack.

A stack is a LIFO sequenced set so that if one traverses the stack from top to bottom one gets a history of the execution of the program.

Use of the Assert Class to Validate Preconditions

```
public class Assert
/* Class used to evaluate preconditions and postconditions
       for methods. Throws an exception if pre-
       postcondition is invalid. */
{  private Assert()
   /* pre: An Assert cannot be constructed. */
       {
      Assert.fail("Attempt to construct an Assert!?");
       }
   static public void pre(boolean test, String message)
   /* pre: result of precondition test. */
   /* post: does nothing if test true, otherwise abort
             w/message. */
             {if (test == false) throw new
             FailedPrecondition(message);
             }
```

```
static public void post(boolean test, String
       message)
/* pre: result of postcondition test. * /
/* post: does nothing if test true, otherwise abort
             w/message. */
{  if (test == false)
             {throw new FailedPostcondition(message); }
       }
static public void condition(boolean test, String message)
/* pre: result of general condition test.  */
/* post: does nothing if test true, otherwise abort
       w/message. */
{ if (test == false) throw new
       FailedAssertion(message);
}
static public void fail(String message)
// post: throws error with message
{  throw new FailedAssertion(message);    }
}
```

```
class FailedAssertion extends Error
{    public FailedAssertion(String reason)
/*/ post: constructs a new failed assertion error.*/
       {super("\nAssertion that failed: " + reason);}
}
class FailedPrecondition extends FailedAssertion
{
       public FailedPrecondition(String reason)
/* post: constructs a new failed precondition.*/
       {super("\nA precondition: " + reason);}
}
class FailedPostcondition extends FailedAssertion{
       public FailedPostcondition(String reason)
/*/ post: constructs a new failed postcondition.*/
       { super("\nA postcondition: " + reason);}
}
```

```
Public static double sqrt(double x)
/* Pre: x is nonnegative. */
/*  post: returns the square root of x. */
{
Assert.pre(x>=0,”The value of x is negative”);
double guess = 1.0;
double guessSquared = guess*guess;

while (math.abs(x-guessSquared) >= 0.000000001)
{ guess = guess + ((x-guessSquared)/2.0)/guess
  guessSquared = guess*guess;
}
return guess
}
```

Assume that the sqrt class in package examples and the Assert class is in package structure. Then if we call sqrt with x < 0 from a main program then the following will be printed out.

```
Structure.FailedPrecondition:
Assertion that failed: A precondition: The value of x is nonnegative.
        At Assert.pre(Assert.java:17)
        at sqrt(examples.java:24)
        at main(examples.java:15)
```

This is a stack trace. The Java RTS keeps a stack of calls to routines. The bottom entry of the stack is the top level of the invocation sequence and where you should start looking for an error. Specifically the error should be close to line 15 of the main program you are executing.

Assert is useful when a precondition can be expressed as a conditional expression.

---

java.lang Class Error

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
```

Direct Known Subclasses:

AWTError, LinkageError, ThreadDeath, VirtualMachineError

public class **Error** extends Throwable

An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The `ThreadDeath` error, though a "normal" condition, is also a subclass of `Error` because most applications should not try to catch it.

Constructor Summary

`Error()`
Constructs an Error with no specified detail message.

`Error(String s)`
Constructs an Error with the specified detail message.

Methods inherited from class java.lang.Throwable

`fillInStackTrace`, `getLocalizedMessage`, `getMessage`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `toString`

Methods inherited from class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

---

java.lang Class Throwable

```
java.lang.Object
  |
  +--java.lang.Throwable
```

Direct Known Subclasses:

Error, Exception

public class **Throwable** extends Object implements Serializable

The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or of one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause.

Instances of two subclasses, Error and Exception, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

By convention, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce an error message.

A Throwable class contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error.

Here is one example of catching an exception:

```
try {
    int a[] = new int[2];
    a[4];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("exception: " +
               e.getMessage());
    e.printStackTrace();
}
```

Constructor Summary

Throwable()
Constructs a new Throwable with null as its error message string.

Throwable(String message)
Constructs a new Throwable with the specified error message.

Method Summary

Throwable	**fillInStackTrace**() Fills in the execution stack trace.
String	**getLocalizedMessage**() Creates a localized description of this Throwable.
String	**getMessage**() Returns the errort message string of this throwable object.
void	**printStackTrace**() Prints this Throwable and its backtrace to the standard error stream.
void	**printStackTrace**(PrintStream s) Prints this Throwable and its backtrace to the specified print stream.
void	**printStackTrace**(PrintWriter s) Prints this Throwable and its backtrace to the specified print writer.
String	**toString**() Returns a short description of this throwable object.

Constructor with Assert ion Precondition Validation

```
public State(int dime,int penny, int nickel,
       int quarter)
/* Generates specific initial States. */
{/* pre: Each slot position must be between 0 and 4.
       0<=dime<=4, etc. */
       Assert.pre((0<=dime&&dime<=4)&&
       (0<=penny&&penny<=4)&&(0<=nickel&&nickel<=4)
       && (0<=quarter&&quarter<=4), "Precondition
       error in testEndCases");

       position = new int[4];
       position[DIME] = dime;
       position[PENNY] = penny;
       position[NICKEL] = nickel;
       position[QUARTER] = quarter;
       previous = null;
      }
```

```
public State(State prior)
    /* pre: prior is a non-null State
      post: constructs a copy of that State to be
                               successor State. */
    {
      position = new int[4];
      for (int coin = DIME; coin <= QUARTER; coin++)
      {
          this.position[coin] =
             prior.position[coin];
      }
      this.previous = prior;

    }
```