American J. of Engineering and Applied Sciences 3 (1): 121-127, 2010
ISSN 1941-7020


# Electronic Design Automation Using Object Oriented Electronics

[1]Walid M. Aly and [2]Mohamed Said Abuelnasr
[1]Technical and Vocational Institute,
[2]Department of Computer Engineering, Faculty of Engineering,
Arab Academy for Science, Technology and Maritime Transport, Alexandria, Egypt

**Abstract: Problem statement:** Electronic design automation is the usage of computer technology and software tools for designing integrated electronic system and creating electrical schematics. **Approach:** An approach is presented for modeling of various electronic and electric devices using object oriented design, aiming on building a library of devices (classes) which can be used for electronic design automation. **Results:** The presented library was implemented using Java programming language to form an Electronic Application Programmer Interface (EAPI) that can be easily utilized for electronic design automation. **Conclusion:** The proposed EAPI that implemented these models in JAVA language can be used for simulation of real electronic circuits and for educational purposes, as the proposed API was designed using object oriented design, adding more new classes, attributes and behaviors to current classes can be done easily.

**Key words:** Modeling, electronics design automation, object oriented, java

## INTRODUCTION

Electronic Design Automation (EDA) (Rosenthal and Damore, 1999) is the usage of computer technology and software tools for designing integrated electronic system and creating electrical schematics, with the continuous growth of semiconductor technology, EDA has become indispensable for modern circuit design.

One of the well known EDA tools is SPICE (Simulation Program with Integrated Circuits Emphasis) (Taubin *et al*., 2007), which is a general-purpose analog electronic circuit simulator program. SPICE is a powerful program which is used in integrated circuits and board-level design to check the integrity of circuit designs and to predict circuit behavior.

For usage simplicity, SPICE is invoked using ASCII text files containing lines of text, each of these lines states a circuit component and how it is connected. Many programs are based on different versions of SPICE and are offering a convenient Graphical User Interface (GUI).

Specialized computer languages were developed to create different software programs for designing and simulating of electronic circuits, the programming language VHDL (Very High Speed Integrated Circuit Hardware Description Language ) -defined in the mid 1980's-is a well known programming language that can be used to write programs that model and simulate electronic circuits (Perry, 2002). VHDL is commonly used as a design language for field-programmable gate arrays and application-specific IC's in electronic design automation of digital circuits.

Object-Oriented Design (OOD) is the design of a system as a group of interacting software objects. In OOD, every entity in the system under consideration is an object, these software objects mimics the real life objects. Objects can be clients, bank accounts, data base connections, stocks……etc.

Each object is created from its class (Johnson and John, 1994), each class defines a certain concept by defining the state and behavior that the created objects can encounter. Each object has its unique state, this state can be changed within the rules set by the behavior. When programming, the state is coded into a number of variables and the behavior is translated into a number of methods, these variables and methods are known as class members (Kortright, 1997). Object oriented design is based upon a number of concrete principles, these principles include- among others- abstraction, encapsulation and inheritance, the following section highlights these principles:

**Abstraction:** Abstraction is keeping a separating distance between the idea and its details, the designer of a class should not be carried away by representing all

**Corresponding Author:** Walid M. Aly, Technical and Vocational Institute,
Arab Academy for Science, Technology and Maritime Transport, Alexandria, Egypt

the states and behavior details of the class, but rather defining only the relevant state and behavior for the concept under consideration. Furthermore a class should be cohesive, representing only one abstraction. We use abstraction every day when interacting with technological objects such as a shift gear. A vehicle driver simply understands its external behavior but have no idea of its inner implementation details. A more efficient design methodology is the one with increased abstraction level.

**Encapsulation:** Encapsulation is as a protective wrapper that prevents the code and data from being misused by other code defined outside the wrapper, encapsulating objects provides abstraction.

The class is the mechanism by which encapsulation is achieved, as you can use the class through its methods without having to worry about the details of its implementation. Access to the code and data inside the wrapper is tightly controlled through well-defined methods.

Correct encapsulation enables the inner workings of objects to be changed as needed as long as the interface to the object is left unaltered.

**Inheritance:** Inheritance in object oriented is a relation between two classes that represents the relation "is a", for example a graduate student is a student. This means that class graduate student inherits from class student, inheritance concept embeds the concept of specialization and generalization, the concept graduate student is a specialization of concept student, and concept student is a generalization of concept graduate student.

Using Java terminology (Herbert, 2006), the inheriting class is called a subclass. The class from which it inherits is called the superclass. When a class inherits from another class, it means that all the non private class members are inherited and can be used directly from the sub class without needing to redefine them again, thus emphasizing the concept of code reuse.

Inheritance also plays an important role in polymorphism, where an object reference variable can point to objects from the actual class of the reference or any of its subclasses which makes the door wide open for methods that is defined to receive one type of object reference to handle different types of objects as long as their classes are sub classes of the class of reference defined as a method argument.

This study aims on investigating the idea of using OOD (Bahrami, 1998) in electronic circuit design as a tool in electronic design automation.

## MATERIALS AND METHODS

The proposed electronic application programmer interface (EAPI) defines the set of classes that model the behavior of each electronic device, these classes are to be used by a programmer for creating the objects of various electronic circuits for simulation and testing. The related classes are grouped together in a package, EAPI defines three different main packages:

- Package electricDevices with two sub packages digitalElectronicDevices and analogElectronicDevices.
- Package circuits.
- Package powerSource.

The following section highlights some of the important classes in these packages:

**Main classes in EAPI:**
**Class ElectricDevice:**
**Concept:** Abstract class ElectricDevice is designed to be the super class of all the devices and encapsulates the concept of electric device.

**State:** Each ElectricDevice object has a symbol, a variable representing the number of ports (terminals) and an array of object from class Port that represents the terminal of the device, and an object of reference type class DeviceModel.

**Behavior:** the class provides accessor methods for the member variables, another abstract method getNetList() also is defined which should be implemented by the subclasses to return netlist representation of the device.

**Constructor:** Objects from this class are created according to a certain model which contains all the data about the device, objects from the appropriate model is created, then passed to the constructor of the required device together with other required parameters, class ElectronicDevice has only this constructor for creating objects, this forces the subclasses to form a model containing all the relevant information before creation of objects of the actual device, the signature of the constructor is: public ElectricDevice (int numOfPorts, String symbol, DeviceModel model).

**Class DeviceModel: Concept:** Abstract class device model represents an abstraction of a model for the electric device, the model will store the important information about the device, this information is normally mentioned in the device data sheet.

This class is abstract, sub classes from this class are classes like class BJTModel and class ResistanceModel.

**Class BJT model:**
**Concept:** This class defines the important information about devices created from class BJT.

**State:** Information stored is the on characteristics ($h_{FE}$,$V_{CE(sat)}$….), off characteristics and maximum ratings. Well known BJTs devices like 2N3903 are stored as final static BJT objects in this class and created with the actual data from the data sheet for immediate use.

**Behavior:** Accessor and mutator methods for all the member variables.

**Class port:**
**Concept:** Class Port encapsulated the terminal of a device.

**State:** Each port object has a title and the integer number of the circuit node to which it is connected.

**Behavior:** Includes a number of useful methods, an interesting one of them is method setNode which receives an integer number representing the node number to connect the terminal to it.

Figure 1 shows class diagram for class ElectronicDevice with its dependencies.

**Class ElectronicCircuit:**
**Concept:** The electronic circuit will be encapsulated using this class.

**State:** the internal state of the class will be stored using hash map data structure, the hash map stores objects from class ElectronicDevice, a map cannot contain duplicate keys; each key can map to at most one value, the key used for each object will be its symbol, each object in the map is a subclass of class ElectronicDevice thus encapsulating all its information including its ports and how they are connected.

**Behavior:** Includes a number of useful methods, an interesting one of them is method addComponent (ElectricDevice component) which add devices to the circuit. This method demonstrates direct usage of polymorphism as the mentioned method can accept objects from any the subclasses of class ElectronicDevice. Each Object is added with a certain key, Objects from class ElectronicCircuit can call method getNetList ( ) to return the netlist description of the circuit for further processing by any spice environment program.

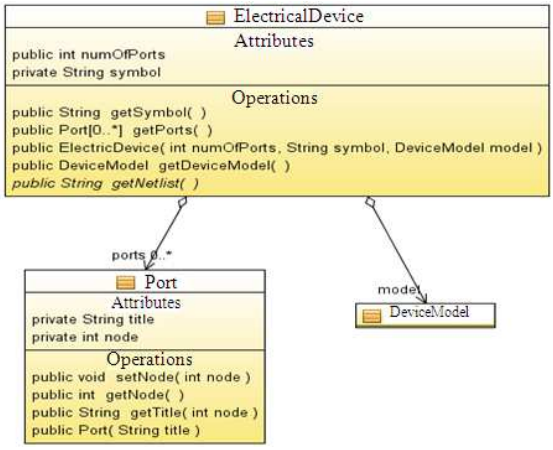


Fig. 1: Class ElectricDevice

One of the subclass of class ElectronicCircuit is abstract class BiasCircuit, subclasses of class BiasCircuit represent bias circuits like voltage divider bias circuit, creating a biased transistor means creating an instance of the required bias circuit and passing it as a reference while calling the appropriate constructor from class Transistor. Class ElectronicCircuit implements the interface CircuitSolver which defines the method required to acquire a complete numeric solution for the circuit. Figure 2 shows the class diagram of class ElectronicCircuit with two of its subclasses.

**Class transistor:**
**Concept:** This abstract class encapsulates the concept of transistor, as this class serves as a super class for all types of transistors; transistor class hierarchy is shown in Fig. 3.

**State:** The internal state of the class will be stored in variables representing the values for voltage gain, current gain, input resistance and output resistance.

**Behavior:** The class defines abstract methods for calculating the variables defined in the state, as these methods are abstract, sub classes from this class are forced to implement them otherwise they would have to be declared themselves abstract too.

**Class BJT:**
**Concept:** This abstract class encapsulates the concept of bipolar junction transistor; this class also serves as a super class for concepts common base, common emitter and common collector transistors.

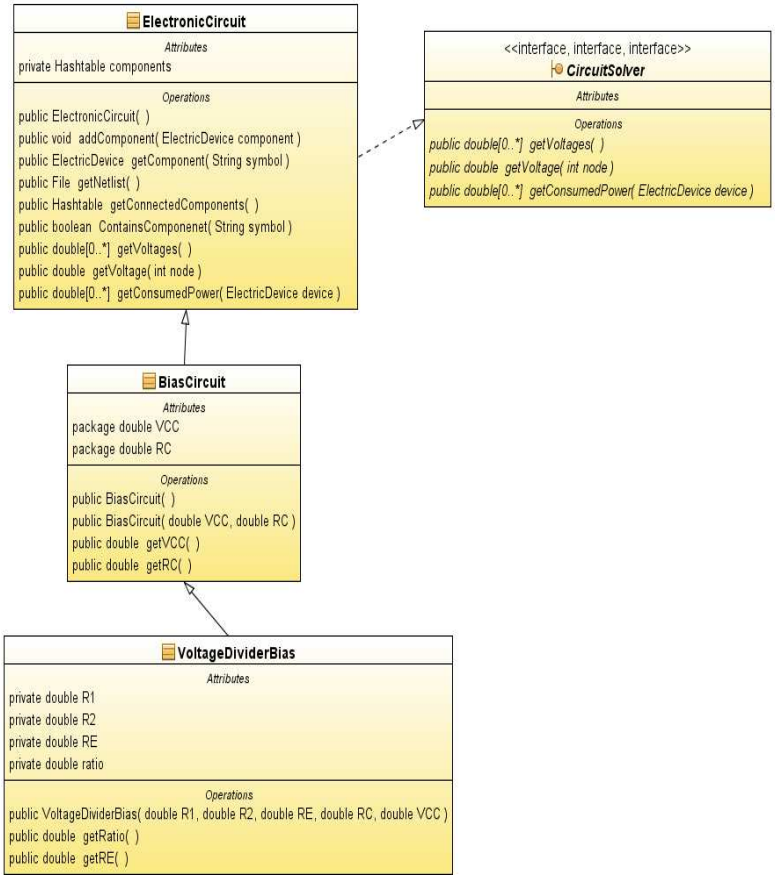


Fig. 2: Class ElectronicCircuit

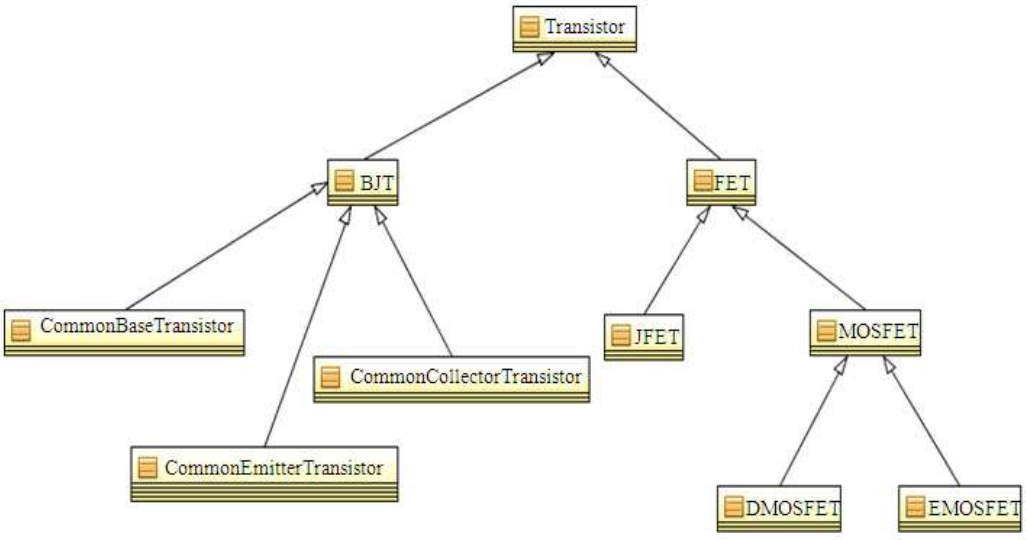


Fig. 3: Transistor hierarchy

**State:** The state of the class is determined using variables for all the currents and voltages of transistor (DC and AC quantities).

NPN and PNP are identified as types in class BJT (represented by a final integer constant), they cannot be defined as separate class because if so they will have to be sub classes of all of the configuration classes and multiple inheritance is not accepted in Java.

The class has a reference of type class BiasCircuit representing the circuit in which the BJT is connected.

**Behavior:** Accessor methods and mutator methods for member variables, other methods exist for checking the mode of the transistor, the class does not implement the inherited methods from its super class and leaves the implementation to the sub classes. Figure 4 shows the class diagram of class BJT.

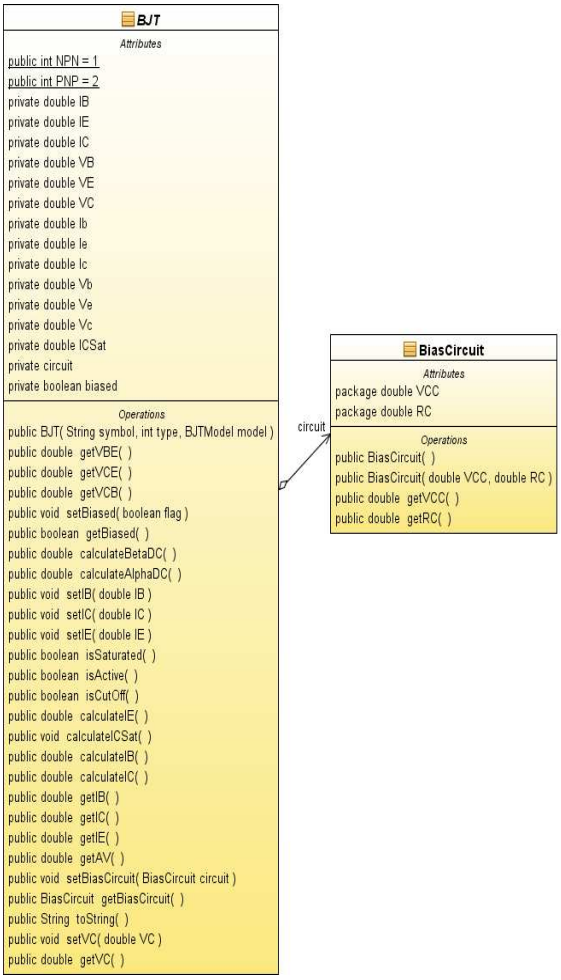


Fig. 4: Class BJT

## RESULTS

As the result of the current research, the electronic application programming interface is developed, this interface can be utilized for modeling different electronic devices and creating various electronic circuits. In the following demonstrations, the actual code for using the proposed interface for creating an electronic circuit and for checking the mode of a BJT transistor is presented, the code is included with comments for illustration and clarity.

**Example of EAPI usage:**

**Creating an electronic circuit:** The following java code represents how an electronic circuit of a common base bipolar junction transistor-shown in Fig. 5 can be coded using the proposed EAPI:

```
public static void create_CB_BJT()
{
//create an object of class ElectronicCircuit
ElectronicCircuit circuit=new ElectronicCircuit();
// create a power device model to describe a fixed
power supply with 24 volts.
PowerDeviceModel model1=new
PowerDeviceModel(24,PowerDeviceModel.FIXE
D);
// create a dc source based on model1
DCSource e1=new DCSource(model1,"Vin");
//connect the power supply ports to node 0 and
node 1.
Port [ ] port_e1= e1.getPorts();
port_e1[0].setNode(1);
port_e1[1].setNode(0);
// add component to circuit
circuit.addComponent(e1);
// create a power device model to describe a
variable power //supply with range from 0-5 volt.
PowerDeviceModel model2=new
PowerDeviceModel(PowerDeviceModel.VARIAB
LE);
model2.setRange(0,5);
DCSource e2=new DCSource(model2,"Vsupply");
// connect second power supply to nodes 0 and 4
Port [] port_e2= e2.getPorts();
port_e2[0].setNode(0);
port_e2[1].setNode(4);
circuit.addComponent(e2);
// create a BJT Model with hFE=50
BJTModel model3=new BJTModel();
model3.sethFE(50);
// Craete a common base NPN BJT based on the
model
```

```
BJT q1=new CommonBaseTransistor("Q1",BJT.NPN,model3);
// connect collector , base and collector to nodes 2,0 and 3 //respectively.
Port [ ] port_q1= q1.getPorts();
port_q1[0].setNode(2);
port_q1[1].setNode(0);
port_q1[2].setNode(3);
circuit.addComponent(q1);
// create a resistanse of 100 and connect it to nodes 3,4
ResistanceModel model4=new ResistanceModel(100);
Resistance re=new Resistance("RE",model4);
Port [ ] port_re= re.getPorts();
port_re[0].setNode(3);
port_re[1].setNode(4);
circuit.addComponent(re);
// create a resistanse of 800 and connect it to nodes 1,2
ResistanceModel model5=new ResistanceModel(800);
Resistance rc=new Resistance("RC",model5);
Port [] port_rc= rc.getPorts();
port_rc[0].setNode(1);
port_rc[1].setNode(2); circuit.addComponent(rc);
}
```

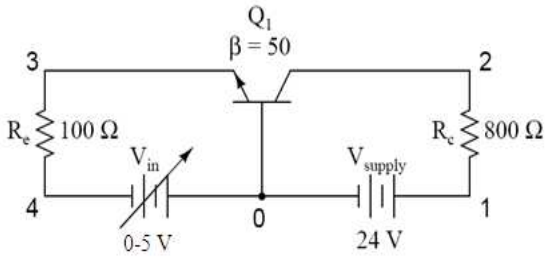


Fig. 5: Sample electronic circuit

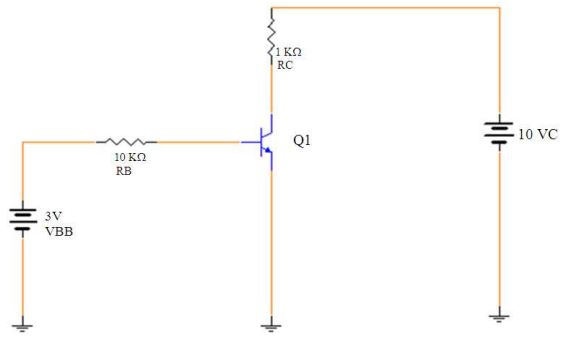


Fig. 6: standard bias circuit

**Checking the mode of a BJT transistor:** The following code creates a standard bias circuit with certain value for dc supplies and resistance as shown in Fig. 6 and then check if the transistor is in saturation mode or not.

```
BJTModel model=new BJTModel();
model.setVCEsat(0.2);
model.sethFE(50);
//Create StandardBiasCircuit circuit: (RB=10000,RC=1000,VBB=3,VCC=10)
BiasCircuit bs = new StandardBiasCircuit(10000,1000,3,10);
BJT tr=new CommonEmitterTransistor("Tr",bs,BJT.NPN, model);
System.out.println(tr.isSaturated()); // prints true
```

## DISCUSSION

The use of object oriented modeling for electronic design automation is presented, and as far as the authors knowledge this is the pioneer attempt.

Within the electronic application programming interface, different classes exist to model electronic devices and electronic circuits. Sample applications of the presented API are demonstrated to prove the efficiency of the models.

the beauty of OOD is its natural reusability capabilities, different states and behaviors can be added to existing classes, even the behavior implementation can be altered without any further problem in the code that use it as long as the behavior interface is left unaltered.

## CONCLUSION

Object Oriented Design can be an important tool for designing electronic circuits with various electronic devices, all of the main devices and circuits can be coded as classes with relevant state and behavior, and the whole electronic circuit will be an interaction between different created objects.

The proposed application programming interface that implements these classes in JAVA language can be used for simulation of real electronic circuits and for educational purposes.

## REFERENCES

Bahrami, A., 1998. Object Oriented Systems Development. 1st Edn., McGraw-Hill, Irwin, ISBN: 025625348X, pp: 432.

Herbert, S., 2006. Core Java, Advanced Features. 8th Edn., Prentice Hall, ISBN: 0-07-226385-7, pp: 1056.

Johnson, R. and M. John, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. 1st Edn., Addison-Wesley Professional, ISBN: 0201633612, pp: 416.

Kortright, E., 1997, Modeling and simulation with UML and Java. Proceeding of the Simulation Symposium, Dec. 1997, IEEE Computer Society, USA., pp. 43-48. DOI: 10.1109/SIMSYM.1997.586477

Perry, D., 2002. VHDL: Programming by Example. 4th Edn., McGraw-Hill Professional, ISBN: 0071400702, pp: 476.

Rosenthal, C. and J. Damore, 1999. Hot topics in electronic design automation. Computer, 32: 79-80. DOI: 10.1109/MC.1999.10133

Taubin, A., J. Cortadella, L. Lavagno, A. Kondratyev and A. Peeters, 2007. Design Automation of Real-Life Asynchronous Devices and Systems. Found. Trends Elect. Des. Autom., 2: 1-13. DOI: 10.1561/1000000006