Building Database-Powered Mobile Applications

Paul POCATILU
Bucharest University of Economic Studies
ppaul@ase.ro

Almost all mobile applications use persistency for their data. A common way for complex mobile applications is to store data in local relational databases. Almost all major mobile platforms include a relational database engine. These databases engines expose specific API (Application Programming Interface) to be used by mobile applications developers for data definition and manipulation. This paper focus on database-based application models for several mobile platforms (Android, Symbian, Windows CE/Mobile and Windows Phone). For each selected platform the API and specific database operations are presented.
Keywords: *Mobile Application, Data Persistence, Embedded Database, SQL, Mobile Platform*

1 Introduction

The use of mobile databases is present in numerous mobile applications from different areas: productivity, m-learning, games etc. Almost all mobile platforms include a relational database engine. These database engines are embedded into platform and are used by built in applications also (contacts, calendar, messaging etc.).
As it can be seen from Table 1, all major mobile operating systems include the API for database operations. The developers can access de databases using database engine native API or using wrapper libraries.

Table 1. Example of database management engines embedded on mobile platforms

Platform	Mobile database	API
Android	SQLite	Java
iOS	SQLite	C/C++
Symbian	SQLite	C++
Windows CE/Windows Embedded Compact/ Windows Mobile	EDB	C/C++
	SQL Server CE/ Compact	C/C#
Windows Phone	SQL Server Compact	C#

One of the most common database engine found on mobile devices is SQLite. SQLite makes available to the majority of existing functions in the management of relational databases. Data types supported by SQLite are INTEGER, REAL, TEXT and BLOB. Unlike other management systems databases, SQLite does not generate errors when a value has a data type different from associated column data type. Instead, the value is converted based on affinity types. SQLite does not support certain types of association (join), referential restriction and nested transactions.
EDB and SQL Server CE/Compact Edition/Compact are proprietary solutions developed by Microsoft available on mobile devices. Depending on implementation and version, there are APIs available for C/C++, C#, LINQ etc.
An EDB database consists of volumes stored as files. Each volume contains one or more databases. Databases contain records; each record is characterized by a set of attributes (properties).
SQL Server Compact is part of Microsoft's SQL Server family.
The objective of this paper is to present several embedded database engine APIs used in mobile applications development.

2 Android

Android uses SQLite database management system [1], [5]. For database operations are available classes **SQLiteOpenHelper**, **SQLiteDatabase** and **Cursor**.
To create a new database it is used a class derived from **SQLiteOpenHelper** abstract

class. There are two methods that need to be implemented: **onCreate()** and **onUpgrade()**:

- **void onCreate(SQLiteDatabase bd)** is called to create the database; the function body contains the code to create tables and other database objects (VIEW, TRIGGER etc.);
- **void onUpgrade(SQLiteDatabase db, int olVers, int newVers)** - is called when the database structure is modified (tables and other database objects).

The **SQLiteDatabase** class implements database operations. An instance of **SQLiteDatabase** is obtained by calling **getWritableDatabase()** or **getReadableDatabase()** methods, available **SQLiteOpenHelper** class and all classes derived from it:

```
SQLiteDatabase bd =
accesBD.getReadableDatabase();
```

SQLiteDatabase class exposes methods that allow direct execution of SQL commands. **execSQL()** is used for commands that don't return data (**CREATE, INSERT** etc.) and **rawQuery()** is used for SQL commands that return data as a **Cursor** (**SELECT**).

Databases created by an Android application are accessible only to that application. To access by other applications content providers are used. The following example is used to create a database with a single table:

```
public class AccesBD extends SQLiteOpenHelper {
  //table name
  public static String TABELA_INTILNIRI = "Intilniri";
  //table creating script
  public static String CREARE_TABELA_INTILNIRI = "CREATE TABLE " +
TABELA_INTILNIRI +"  (id INTEGER PRIMARY KEY AUTOINCREMENT, data INTEGER, subiect
TEXT,  loc TEXT)";
  //database name
  protected static String BAZA_DE_DATE = "pdm.db";

  public AccesBD(Context context){
      super(context, BAZA_DE_DATE, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase bd) {
            bd.execSQL(CREARE_TABELA_INTILNIRI);
  }

  @Override
  //is called when database structure changes
  public void onUpgrade(SQLiteDatabase bd, int versAnt, int verNoua) {
            bd.execSQL("DROP TABLE IF EXISTS " + TABELA_INTILNIRI);
            onCreate(bd);
  }
}
```

Also in the class **SQLiteDatabase** includes specialized methods for:

- adding records: **insert()**;
- deleting records: **delete()**;
- changing records: **update()**;
- data queries: **query()**.

The next listing shows a part of **BDIntilniri** class that implements methods for data manipulation (insert, update, delete and query). The **Intilniri** class includes the fields: **id**, **data**, **loc** and **subiect** and corresponding getter and setters methods.

```
class BDIntilniri
{
  AccesBD accesBD;
  protected static final String COL_ID = "id";
  protected static final String COL_DATA = "data";
  protected static final String COL_SUBIECT = "subiect";
```

```
  protected static final String COL_LOC = "loc";

  BDIntilniri(Context context)
  {
      accesBD = new AccesBD(context);
  }
  //
  //inserts an appointment in the table
  long adaugaInregistrare(Intilnire intilnire)
  {
      SQLiteDatabase bd = null;
      long rezInsert = 0;
      ContentValues valori = new ContentValues();

      try
      {
             bd = accesBD.getWritableDatabase();

             valori.put(COL_DATA, intilnire.getData());
             valori.put(COL_SUBIECT, intilnire.getSubiect());
             valori.put(COL_LOC, intilnire.getLoc());

             rezInsert = bd.insert(AccesBD.TABELA_INTILNIRI,  null, valori);
      }
      catch(SQLException ex)
      { ex.printStackTrace();}

      return rezInsert;
  }

//...
}
```

The usage of **BDIntilniri** class is presented in the following example:

```
AccesBD bdA = new AccesBD(this);
BDIntilniri bd = new BDIntilniri(this);

//current date and time
Calendar c = Calendar.getInstance();

//insert a new record
bd.adaugaInregistrare(new Intilnire(1, c.getTimeInMillis() + 3600000 , "Curs 1",
"2204"));
```

When using dedicated methods for database operations, if the *WHERE* clause is present, the corresponding parameter is initialized properly without its corresponding keyword. Instead of values it can be used question marks (?) that are replaced with corresponding **String** value from the following parameter (an array of strings).

Data query results are found in an object that implements **Cursor** interface. Records are managed through its results. Interface exposes methods to browse records (**move()**, **moveToFirst()**, **moveToLast()**, **moveToNext()**, **moveToPrevious()**, **moveToPosition()**) and to obtain the value of any field in the current record type methods **getTYPE()**, based on column index, depending on data type.

The next example presents record selection and processing using the WHERE clause arguments.

```
String [] paramWhere = new String[]{"2204"};
String cond = COL_LOC + "=?";
String orderBy = COL_DATA + " ASC";

String [] paramWhere = new String[]{"2204"};
```

```
String cond = COL_LOC + "=?";
String orderBy = COL_DATA + " ASC";

//...
Cursor rez = bd.query(AccesBD.TABELA_INTILNIRI, null, cond, paramWhere, null, null,
orderBy);

rez.moveToFirst();
//...

for (int i=0; i < nInreg; i++)
//sau while(rez.isAfterLast() != false)
{
       intilniri[i] = new Intilnire(rez.getInt(0), rez.getLong(1), rez.getString(2),
rez.getString(3));
       rez.moveToNext();
}
```

Typically, databases are saved to the associated application data directory (*/data/data/package_name*).

3 Symbian

Symbian C++ API provides several classes to access SQLite database engine [1], [2]. The **RSqlDatabase** class is used for database management. The class allows execution of SQL statements that doesn't return values. SQL commands that return data or use parameters are managed by **RSqlStatement** class. In order to access these classes *sqldb.h* header file and *sqldb.lib* library are required.

In order to create a new database **Create()** or **CreateL()** methods are used. The methods receive the parameters associated with the database file name and, optionally, specific security policy flags (**TSecurityPolicy** and **RSqlSecurityPolicy**) and configuration parameters. The flags determine database and its objects access mode and the access rights (read, write, etc.).

Opening a database is done with the **Open()** or **OpenL()** method that receives as parameters the database name and optional configuration parameters.

Execution of SQL that does not return results achieved with the method **Exec()** which receives as parameter a descriptor initialized with SQL command. **RSqlDatabase** class includes methods to copy (**Copy()**) and delete (**Delete()**) databases. Method **Close()** is called to release resources. The following example shows a Symbian C++ sequence that creates a database with a single table.

```
_LIT(KBD, "c:\\pdm.db");
_LIT(KExecCreate, "CREATE TABLE Intilniri (id INTEGER PRIMARY KEY AUTOINCREMENT,
data INTEGER, subiect TEXT,  loc TEXT)");
RSqlDatabase bd;

//database creation
bd.CreateL(KBD);
CleanupClosePushL(bd);

//CREATE command execution
bd.Exec(KExecCreate);

CleanupStack::PopAndDestroy();
bd.Close();
```

SQL queries requests that return values or uses parameters are implemented using **RSqlStatement** class. Objects initialization is achieved by **Prepare()**/**PrepareL()** method calls**.** The methods receive as parameters an opened database handle and a descriptor initialized with a SQL command. An exception is thrown if the database is not initialized or the SQL command is invalid. The method **Next()** is called in order to receive the current record. The method returns **KSqlAtRow** value if the current

record is valid. The method **ColumnTYPE()** is called with the column index in order to obtain the corresponding value. TYPE represents the column data type (**Text**, **Int64**, **Binary**, etc.). The following example shows a simple selection.

```
_LIT(KBD, "c:\\pdm.db");
_LIT(KSelect, "SELECt * FROM Intilniri;");

RSqlDatabase bd;
RSqlStatement sqlSelect;

//open database
User::LeaveIfError(bd.Open(KBD));
CleanupClosePushL(bd);

//command initialization
User::LeaveIfError(sqlSelect.Prepare(bd, KSelect));
CleanupClosePushL(sqlSelect);

TBuf<20> subiect;
TBuf<20> loc;
TInt64 ms;

//query execution
while(sqlSelect.Next() == KSqlAtRow)
{
       ms = sqlSelect.ColumnInt64(1);
       TTime data(ms);
       sqlSelect.ColumnText(2, subiect);
       sqlSelect.ColumnText(3, loc);
       //process current record
}

CleanupStack::PopAndDestroy(2);
bd.Close();
```

If SQL commands use parameters, their names are included in the constant descriptor preceded by colon symbol (**:**). **ParameterIndex()** method is called to obtain the parameter associated index. The index is later used by methods of like **BindTYPE()** for initialization with the desired values (**BindText()**, **BindInt64(),** etc.). After the parameter initialization the SQL command is executed by calling **Exec()** methods. In order to reuse the **RSqlStatement** object (parameter initialization with other values) the method **Reset()** has to be called before. The following code sequence exemplifies record insertion using parameters.

```
_LIT(KBD, "c:\\pdm.db");
_LIT(KInsert, "INSERT INTO Intilniri(data, subiect) values(:vdata, :vsubiect);");
RSqlDatabase bd;
RSqlStatement sqlInsert;
TInt64 data1;
TInt64 data2
//...
User::LeaveIfError(bd.Open(KBD));
CleanupClosePushL(bd);
User::LeaveIfError(sqlInsert.Prepare(bd, KInsert));
CleanupClosePushL(sqlInsert);
//obtain parameter index
TInt p1 = sqlInsert.ParameterIndex(_L(":vdata"));
TInt p2 = sqlInsert.ParameterIndex(_L(":vsubiect"));
//parameter initialization for first record
User::LeaveIfError(sqlInsert.BindInt64(p1, data1));
User::LeaveIfError(sqlInsert.BindText(p2, _L("Curs 1")));
// INSERT execution
User::LeaveIfError(sqlInsert.Exec());
```

```
//reset
User::LeaveIfError(sqlInsert.Reset());
//parameter initialization for second record
User::LeaveIfError(sqlInsert.BindInt64(p1, data2));
User::LeaveIfError(sqlInsert.BindText(p2, _L("Curs 2")));
//INSERT execution
User::LeaveIfError(sqlInsert.Exec());
CleanupStack::PopAndDestroy(2);
bd.Close();
```

4 Windows Mobile

Windows CE includes a proprietary database system called EDB (Embedded Database) [1], [3]. The following steps are required in order to populate a database:

- create or open a volume and mount it using **CeMountDBVolEx()** function; the volumes are identified by **CEGUID** type. It is initialized when to volume is mounted.
- if database doesn't exists it is created by calling **CeCreateDatabaseWithProps()** function; if database exists it is opened by **CeOpenDatabaseInSession()** calling function; session identifier is obtained in advance by calling **CeCreateSession()** function;
- the records are written by calling **CeWriteRecordProps()** function, for data stream operation **CeStreamWrite()** is used; function **CeStreamSaveChanges()** is called to effectively write data;
- the handle are released calling **CloseHandle()** function.

To read records from an existing database:

- open and mount an existing volume by calling **CeMountDBVolEx()**;
- open a session with **CeCreateSession();**
- open the database by calling **CeOpenDatabaseInSession(),** using the existing session identifier;
- if necessary, find a specific record using **CeSeekDatabaseEx()** function;
- read records with **CeReadRecordProps()** function, if carried out operations on the data stream the **CeStreamRead()** function is used;
- release the handles using **CloseHandle()**.

Operations can be performed and the transaction level. In this case it is necessary to call **CeBeginTransaction()** function at beginning and **CeEndTransaction()** to the end in order to save updates made.

The functions **CeFindFirstDatabaseEx()** and **CeFindNextDatabaseEx()** are used to search a database in a volume. Database information are obtained by using **CeOidGetInfoEx2()** function.

Attributes are stored with the records in the database. **CEPROPVAL** structure is used access properties from a database [1], [3]. The properties are identified by a code and an associated data type. These components are coded in **proprid** field. The **val** field is presented as a union of type **CEVALUNION**, and it store a property value for the current record. Property types are shown in Table 2.

Table 2. EDB Data Types

Type	**C/C++ type**	**Field**
CEVT_I2	short	iVal
CEVT_UI2	USHORT	uiVal
CEVT_I4	long	lVal
CEVT_UI4	ULONG	ulVal
CEVT_FILETIME	FILETIME	filetime
CEVT_LPWSTR	LPWSTR	lpwstr
CEVT_BLOB	CEBLOB	blob
CEVT_BOOL	BOOL	boolVal
CEVT_R8	double	dblVal
CEVT_STREAM	-	-
CEVT_RECID	CEGUID	
CEVT_AUTO_I4	long	lVal
CEVT_AUTO_I8	double	dblVal

The following listing shows how to display all records from all databases within a volume.

```
CEGUID guid;
CEVOLUMEOPTIONS cevo = {0};
cevo.wVersion = 1;
CEOIDINFOEX oidInfo = {0};
wchar_t buff[250];
HANDLE hSes, hBD, hBDS;
BOOL rez;
rez = CeMountDBVolEx(&guid, L"pim.vol", &cevo,OPEN_EXISTING);
if (rez == FALSE) { /*erorr*/ }
hBD = CeFindFirstDatabaseEx(&guid, 0);
if (hBD != INVALID_HANDLE_VALUE)
{
       oidInfo.wVersion = CEOIDINFOEX_VERSION;
       oidInfo.wObjType = OBJTYPE_DATABASE;
       //creare sesiune
       hSes = CeCreateSession(&guid);
       if (hSes == INVALID_HANDLE_VALUE) {/* error */}
       CEOID oidBD = CeFindNextDatabaseEx(hBD, &guid);
       while (oidBD != 0)
       {
       //obtain database information
       rez = CeOidGetInfoEx2(&guid, oidBD, &oidInfo);
       if (rez != TRUE) {/* error */}
       //open database
       hBDS = CeOpenDatabaseInSession(hSes, &guid, &oidBD,
            oidInfo.infDatabase.szDbaseName, NULL,    CEDB_AUTOINCREMENT, NULL);
       if (hBDS == INVALID_HANDLE_VALUE) {/* error */}
       PCEPROPVAL pInreg = NULL;
       PBYTE pBuffInreg  = NULL;//memory is allocated by function
       WORD wProp;//number of properties
       DWORD dwLgInreg;// record lengths
       //memory is allocatd by function
       CEOID ceoid  = CeReadRecordPropsEx(hBDS, CEDB_ALLOWREALLOC, &wProp, NULL,
             &(LPBYTE)pBuffInreg, &dwLgInreg, NULL);
       int k = 0;
       while(ceoid != 0)
       {
        pInreg = (PCEPROPVAL)pBuffInreg;
        //for each field
        for (int i = 0; i < wProp; i++)
        {
         switch(LOWORD(pInreg->propid))
         {
         case CEVT_LPWSTR:
              //process string values
              break;
         //integers
         case CEVT_I2:
         case CEVT_I4:
         case CEVT_UI2:
         case CEVT_UI4:
         case CEVT_AUTO_I4:
         case CEVT_BOOL:
              //process integer values
              break;
         case CEVT_R8:
              //process floating point values
              break;
         default:
              //other types
              break;
       }
        OutputDebugString(buff);
       //next field
       pInreg++;
       }
       LocalFree(pBuffInreg);
```

```
        //next record
        ceoid  = CeReadRecordPropsEx(hBDS, CEDB_ALLOWREALLOC,  &wProp, NULL,
            &(LPBYTE)pBuffInreg, &dwLgInreg, NULL);
          k++;
        }
        CloseHandle(hBDS);
        //next database
        oidBD = CeFindNextDatabaseEx(hBD, &guid);
        }
        CloseHandle(hBD);
        CloseHandle(hSes);
}
CeUnmountDBVol(&guid);
```

Functions and structures for EDB database operations are defined in the file header *windbase_edb.h* and *EDB* symbol should be defined.

SQL Server CE databases are accessible through ADO.NET using .NET Compact Framework platform [1], [7]. It includes the **System.Data.SqlServerCe** namespace that exposes classes such as: **SqlCeConnection**, **SqlCeCommand**, **SqlCeDataReader**, **SqlCeDataAdapter**, **SqlCeEngine**, **SqlCeResultSet**, **SqlCeUpdatableRecord**.

5 Windows Phone

Windows Phone includes support for SQL Server Compact database. Database access is done through LINQ (Language Integrated Query) [1], [4]. NET platform includes LINQ to SQL component for relational data management using objects. Therefore, projects must include a reference to **System.Data.Linq** library and the related namespace. In order to access database objects in Windows Phone applications is necessary to create classes associated with the relational model. The classes associated to database tables are decorated **Table** attribute. Associated column properties are marked with the attribute **Column**. If a property is the primary key, **Column** attribute constructor is receive **IsPrimaryKey** parameter initialized to true. If field values are generated automatically, the **IsDbGenerated** property is initialized to true. In order to index data in a table the attribute **Index** is used.

For dynamic data linking and notifications related to data changes, the interfaces **INotifyPropertyChanging** (the event **PropertyChanging** fires before changing the value of property) and **INotifyPropertyChanged** (the **PropertyChanged** event is fired after changing occurs) are implemented.

Database connection is managed using **DataContext** class. It is responsible for translating the object model in a database. In applications, the database associated class is derived from **DataContext** class. The database connection string is *"Data Source=isostore:/database_name.sdf"* and it is passed as a parameter to **DataContext** class constructor. In addition to filename, the connection string allows transmission of database-specific parameters like user, password etc.

Database associated class include tables as objects of type **Table<T>**. The class **Table<T>** includes **InsertOnSubmit()** method to add new records. The method receives as parameter an object of associated table class type. Update operations are made by changing the associated properties values using a table object. Deleting a record is made by **DeleteOnSubmit()** method and deleting multiple records by using **DeleteAllOnSubmit()** method. The methods receive as parameter the object that will be deleted, respectively the collection of records that will be deleted (as a result of a query). Data query is made through LINQ. In order to save changes the method **SubmitChanges()** form existing **DataContext** class has to be called.

Creating a new database is done by calling method **CreateDatabase()** the **DataContext** class. To delete an existing database **deleteDatabase()** method is called.

DatabaseExists() method returns **true** if a database exists.
To change the database structure and object of **DatabaseSchemaUpdater** type need to be initialized by calling **CreateDatabaseSchemaUpdater()**. The class provides methods for adding tables and relationships and columns. The method **Execute()** is called to apply changes.
The existing databases created using the development environment or through dedicated applications, can be included in the project as resources or as content. This is a database table associated class:

```
[Table]
public class Test : INotifyPropertyChanged, INotifyPropertyChanging
{
  //test description
  string descriere;
  //test date
  DateTime data;

  public event PropertyChangedEventHandler PropertyChanged;
  public event PropertyChangingEventHandler PropertyChanging;

  [Column(IsPrimaryKey=true, IsDbGenerated=true)]
  public int Id  { get;  set;   }

  [Column]
  public string Descriere
  {
    get { return descriere;  }
    set
    {
      if(PropertyChanging != null)
         PropertyChanging(this,
            new PropertyChangingEventArgs("Descriere"));
      descriere = value;

      if (PropertyChanged != null)
         PropertyChanged(this,
             new PropertyChangedEventArgs("Descriere"));
      }
   }

  [Column]
  public DateTime Data
  {
    get { return data; }
    set
    {
       if(PropertyChanging != null)
          PropertyChanging(this,
              new PropertyChangingEventArgs("Data"));
       data = value;

       if (PropertyChanged != null)
           PropertyChanged(this,
               new PropertyChangedEventArgs("Data"));
     }
  }

  [Column]
  public int NumarIntrebari { get;  set; }
}
```

The associated database class is defined as follows:

```
public class TestDataContext : DataContext
```

```
{
   public static string connString = "Data Source=isostore:/Teste.sdf";
   public TestDataContext(string connString) : base(connString)        { }
   public Table<Test> Teste;
}
```

This code sequence is used to create the database:

```
using TestDataContext bd = new TestDataContext(TestDataContext.connString)
{
  //if the database not exists it will be created
  if (!bd.DatabaseExists())
  {
     bd.CreateDatabase();
  }
}
```

This code sequence is used to add new records in the table:

```
Test test = new Test
{
    Descriere = editDescriere.Text,
    //if the current value is null, the current date is used
    Data = dp.Value ?? DateTime.Now,
    NumarIntrebari = Int32.Parse(editIntrebari.Text)
 };

using (var bd = new TestDataContext(TestDataContext.connString))
{
   //insert records
   bd.Teste.InsertOnSubmit(test);
   //commit the changes
   bd.SubmitChanges();
}
```

In order to update a record, the following code sequence is used:

```
using (var bd = new TestDataContext(TestDataContext.connString))
{
 //get the test with the given id
 var test = (from Test test in bd.Teste
             where test.Id == id
             select test).First();
 //apply the changes
 //dp is a DatePicker
 test.Data = dp.Value ?? DateTime.Now;
 //editDescriere is a TextBox
 test.Descriere = editDescriere.Text;
 // editIntrebari is a TextBox
 test.NumarIntrebari = Int32.Parse(editIntrebari.Text);
 // commit the changes
 bd.SubmitChanges();
}
```

6 Conclusion and future work

Every mobile modern operating system and platform includes a database engine and makes APIs available to developers. The development complexity of database-powered mobile applications varies from platform to platform.

Future work includes database performance analysis on each presented platform. Another aspect that has to be analyzed is related to database security.

References

[1] P. Pocatilu, *Programarea dispozitivelor*

mobile, ASE Publishing House, Bucharest, 2012
[2] I. Litovski, R. Maynard, Inside Symbian SQL: A Mobile Developer's Guide to SQLite, John Wiley & Sons, 2010
[3] D. Boling, *Programming Microsoft Windows Embedded CE 6.0*, Microsoft Press, 2008
[4] Rob Miles, *Windows Phone Programming in C#*, available online at: http://www.robmiles.com/c-yellow-book/Rob%20Miles%20Windows%20Phone%20Blue%20Book.pdf, [October 2011]
[5] S. Hashimi, S. Komatineni, D. MacLean, *Pro Android 3*, Apress, 2011
[6] P. Yao, D. Durant, *.NET Compact Framework Programming with C#*, Prentice Hall PTR, 2004
[7] A. Wigley, D. Mothand, P. Foot, *Microsoft Mobile Development Handbook*, Microsoft Press, 2007

Paul POCATILU graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 1998. He achieved the PhD in Economics in 2003 with thesis on Software Testing Cost Assessment Models. He has published as author and co-author over 45 articles in journals and over 40 articles on national and international conferences. He is author and co-author of 10 books, (Mobile Devices Programming and Software Testing Costs are two of them). He is associate professor in the Department of Economic Informatics of the Academy of Economic Studies, Bucharest. He teaches courses, seminars and laboratories on Mobile Devices Programming, Economic Informatics, Computer Programming and Project Management to graduate and postgraduate students. His current research areas are software testing, software quality, project management, and mobile application development.