

## Bridge [GoF]

### Intent

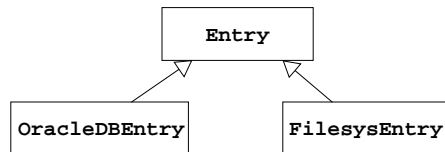
Decouple an abstraction from its implementation so that the two can vary independently.

### Also Known As

Handle/Body.

### Motivation

Suppose we have an abstraction `Entry`, and we want to implement its persistence in two different ways. One possibility is to provide subclasses for each way:

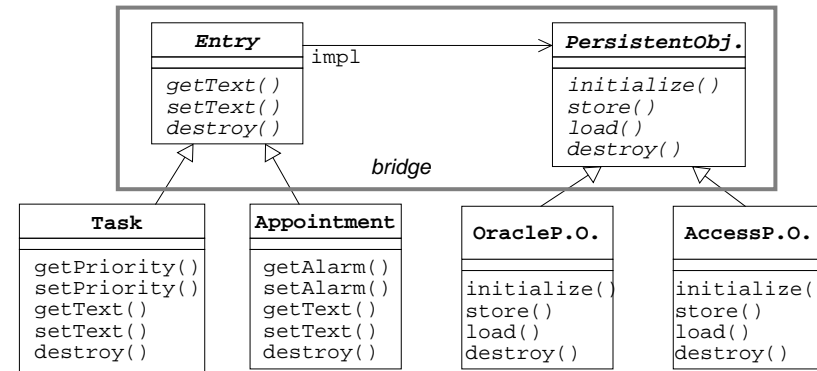


Design Patterns

Bridge [GoF]

1

A better solution is based on the idea that the abstractions (`Entry`, `Task`) are separated from their implementation (`Oracle DB`, `file system`). The Bridge pattern addresses this problem:

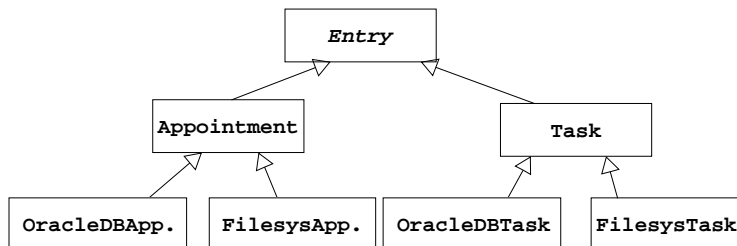


Design Patterns

Bridge [GoF]

Subclass `OracleDBEntry` is responsible to store the attributes of class `Entry` in table of the Oracle DB whereas subclass `FilesysEntry` stores them in files.

Consider the proliferation of classes if we introduce a specialized subclass with business meaning, say class `Task`, for class `Entry`:



Drawbacks:

1. It is inconvenient to extend the business classes to cover different kinds of `Entry` or different new persistent storage areas.
2. Above class hierarchy makes client code dependent from the media on which entries of different kinds will be stored.

Design Patterns

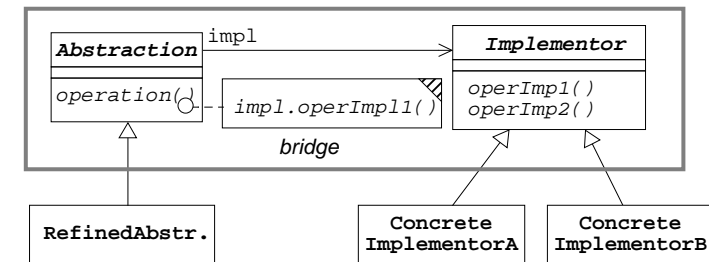
Bridge [GoF]

2

### Applicability

- Both, abstractions and implementations should be extensible by subclassing in different ways.
- Changes of the implementations should have no impact on clients.
- You want to avoid the proliferation of classes.

### Structure



Design Patterns

Bridge [GoF]

## Participants

- **Abstraction** (Entry):
  - defines the Abstraction's interface
  - maintains a reference to an object of type Implementor
- **RefinedAbstraction** (Task):
  - extends the interface defined by Abstraction
- **Implementor** (PersistentObject):
  - defines an interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface. Typically, the Implementor's interface provides only primitive methods.
- **ConcreteImplementor** (OraclePersistentObject, AccessP.O.):
  - implements the Implementor's interface.

## Collaborations

- Abstraction forwards client requests to its Implementor object.

## Sample Code

We consider the example given in the motivation section. An abstraction of different kinds of entries is provided by the (here: *abstract*) class Entry:

```
public abstract class Entry {

    private PrimaryKey pk;
    private PersistentObject impl;
```

## Consequences

The bridge pattern has the following consequences:

- Decoupling of interface and implementation
- Improved extensibility
- Hiding implementation details from clients

## Implementation

- Creating the *right* Implementor: you might use the Abstract Factory pattern
- Only *one* Implementor: the separation is still useful
- *Sharing* implementations: it's sometimes possible

```
// Constructors:
// The first one is used for the first-time initialization
// of any kind of an Entry.
// The second one is used to recreate an Entry given a
// primary key.
protected Entry(String tableName,
    String[] colNames, Class[] colTypes) throws Exception
{
    impl = PersistentObjectFactory.getInstance()
        .makePersistentObject(tableName, colNames, colTypes);
}

protected Entry(String tableName, String[] colNames,
    Class[] colTypes, PrimaryKey pk) throws Exception
{
    this.pk = pk;
    impl = PersistentObjectFactory.getInstance()
        .makePersistentObject(tableName, colNames, colTypes, pk)
}
```

```
// Helper method that returns the concrete implementation.
protected PersistentObject getPersistentObject()
    throws Exception
{
    return impl;
}

// Helper methods that return the abstraction of the
// primary key. Objects of class PrimaryKey are
// serializable.
public final PrimaryKey getPrimaryKey() throws Exception
{
    if (pk != null) {
        return pk;
    } else {
        pk = getPersistentObject().getPrimaryKey();
    }
    return pk;
}
```

Subclasses of `Entry` define different kinds of entries the application might use.  
For example, `Task` might look like:

```
public final class Task extends Entry
{
    private Object[] data = new Object[3];
    // data[0]: theDate, data[1]: theText, data[2]: thePriority

    static final String tableName = "TASK";
    static final String[] colNames = {
        "Datum", "Text", "Priority"
    };
    static final Class[] colTypes = {
        java.util.Date.class, java.lang.String.class,
        java.lang.Integer.class
    };
}
```

```
// Business methods:
public abstract void setDate(Date date) throws Exception;
public abstract Date getDate() throws Exception;
public abstract void setText(String text) throws Exception;
public abstract String getText() throws Exception;

public abstract void destroy() throws Exception;

// Enforcing the implementation of equals() and hashCode()
// methods:
public abstract boolean equals(Object o);
public abstract int hashCode();
}
```

Notice that in the code above the constructors use the factory class `PersistentObjectFactory` (Abstract Factory pattern, Singleton pattern) for creating the correct implementation. This class returns the correct implementation (code not shown).

```
// First-time constructor for inexistent row in table.
public Task(Date date, String text, int priority)
    throws Exception
{
    super(tableName, colNames, colTypes);
    data[0] = date;
    data[1] = text;
    data[2] = new Integer(priority);
    getPersistentObject().initialize(data);
}

// Constructor for already persisted Task object.
public Task(PrimaryKey pk) throws Exception
{
    super(tableName, colNames, colTypes, pk);
    data = getPersistentObject().load();
}
```

```
// Implementation of business methods
public void setDate(Date date) throws Exception {
    data[0] = date;
    getPersistentObject().store(data);
}
public Date getDate() throws Exception {
    //..
}
public void setText(String text) throws Exception {
    //..
}
public String getText() throws Exception {
    //..
}
public void setPriority(int priority) throws Exception {
    //..
}
public int getPriority() throws Exception {
    //..
}
public void destroy() throws Exception {
    getPersistentObject().destroy(); }
}
```

```
} // Task
```

Objects of kind `PrimaryKey` are a common abstraction of primary keys (also referred to as object identifiers OID). The field depends on the particular implementation used. For Oracle, for example, subclass `OraclePrimaryKey` is a concrete implementation:

```
public abstract class PrimaryKey implements Serializable {
    public abstract boolean equals(Object otherPrimaryKey);
    public abstract int hashCode();
}

public class OraclePrimaryKey extends PrimaryKey {
    int theKey;
    OraclePrimaryKey(int i) { theKey = i; }
    public boolean equals(Object otherPrimaryKey) {
        if ((otherPrimaryKey != null) &&
            otherPrimaryKey instanceof OraclePrimaryKey) {
            return true;
        } else { return false; }
    }
}
```

```
// Helpers.
public boolean equals(Object o) {
    if (o instanceof Task) {
        try {
            return getPrimaryKey().equals((Task) o
                                           .getPrimaryKey());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    return false;
}

public int hashCode() {
    try {
        return getPrimaryKey().hashCode();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

public String toString() {
    return data[0].toString() + " " + ... ;
}
```

```
public int hashCode() { return theKey; }
}
```

The implementation of a concrete subclass of `PrimaryKey` for Access contains a string as elementary key (code not shown).

An object of a subclass of `Entry` maintains a reference to `PersistentObject`, the abstract class that declares an interface to the underlying persistence implementations used:

```
public abstract class PersistentObject
{
    protected String tableName;
    protected String[] colNames;
    protected Class[] colTypes;

    protected PersistentObject(String tableName,
                               String[] colNames, Class[] colTypes)
    {
        this.tableName = tableName;
        this.colNames = colNames;
        this.colTypes = colTypes;
    }
}
```

```

    public abstract PrimaryKey getPrimaryKey() throws Exception;
    public abstract void initialize(Object[] data) throws Ex.;
    public abstract void store(Object[] data) throws Exception;
    public abstract Object[] load() throws Exception;
    public abstract void destroy() throws Exception;
}

```

Concrete subclasses of `PersistentObject` implement the concrete kinds of Entry objects. For example, a sketch of an implementation using Oracle might look like:

```

public final class OraclePersistentObject
    extends PersistentObject
{
    private HashMap javaSqlTypeMap = new HashMap();

    private ConnectionManager cm;
    private int internalKey = -1; // effective value set later
}

```

```

// Helper method.
private void setInternalKey() throws Exception {
    if (internalKey < 0) { // Not yet associated with a row.
        Connection con = null; Statement stmt = null;
        ResultSet rs = null;
        try {
            con = cm.getConnection();
            stmt = con.createStatement();
            // Get unique OID. Oracle-specific.
            StringBuffer sqlCmd = new StringBuffer("SELECT ");
            sqlCmd.append(tableName);
            sqlCmd.append(" _Seq.nextval FROM dual");
            rs = stmt.executeQuery(sqlCmd.toString());
            if (rs.next()) { internalKey = rs.getInt(1); }
        } catch (Exception e) { throw e; }
        finally {
            if (rs != null) rs.close();
            if (stmt != null) stmt.close();
            if (con != null) cm.close(con);
        }
    } // end of if (internalKey < 0)
}
}

```

```

// Constructors.
public OraclePersistentObject(String tableName,
    String[] colNames, Class[] colTypes) throws Exception
{
    super(tableName, colNames, colTypes);
    cm = ConnectionManager.getInstance();
    setInternalKey();
}

public OraclePersistentObject(String tableName,
    String[] colNames, Class[] colTypes,
    PrimaryKey pk) throws Exception
{
    super(tableName, colNames, colTypes);
    // The following line does not guarantee that
    // a corresponding row in the database really exists :- (
    this.internalKey = ((OraclePrimaryKey) pk).theKey;
    cm = ConnectionManager.getInstance();
}

```

```

// Implementor's public methods
public PrimaryKey getPrimaryKey() throws Exception {
    return new OraclePrimaryKey(internalKey);
}

// ... turn over

```

```

public void initialize(Object[] data) throws Exception
{
    Connection con = null; PreparedStatement stmt = null;
    try {
        con = cm.getConnection();
        StringBuffer
            sqlCmd = new StringBuffer("INSERT INTO ");
        sqlCmd.append(tableName); sqlCmd.append(" ( ");
        sqlCmd.append("OID");
        for (int i = 0; i < colNames.length; i++) {
            sqlCmd.append(", "); sqlCmd.append(colNames[i]);
        }
        sqlCmd.append(" ) VALUES ( "); sqlCmd.append(" ? ");
        for (int i = 0; i < colNames.length; i++) {
            sqlCmd.append(", ? ");
        }
        sqlCmd.append(" )");
        stmt = con.prepareStatement(sqlCmd.toString());
        stmt.setInt(1, internalKey);
        for (int i = 0; i < colNames.length; i++) {
            dynamicDispatchSetXxxMethod(stmt, i+2, data[i]);
        }
    }
}

```

```

public Object[] load() throws Exception
{
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    Object[] data = null;
    // Use SQL SELECT, code not shown...
}
return data;
}

public void destroy() throws Exception {
    // Use SQL DELETE, code not shown...
}

```

```

        stmt.executeUpdate();
    } catch (Exception e) {
        throw e;
    } finally {
        if (stmt != null) stmt.close();
        if (con != null) cm.close(con);
    }
}

public void store(Object[] data) throws Exception
{
    Connection con = null;
    PreparedStatement stmt = null;
    // Use SQL UPDATE, code not shown...
}

```

```

// Further private helper methods.
private void setupDefaultJavaSqlTypeMap() {
    javaSqlTypeMap.put(java.lang.String.class,
        "VARCHAR2(40)");
    // Integer --> "NUMBER"
}
private java.util.Date
    convertTimestamp(java.sql.Timestamp t) throws Exception
    //...
}
private java.sql.Timestamp convertDate(java.util.Date d) {
    //...
}

```

```

private void dynamicDispatchSetXxxMethod(
    PreparedStatement stmt, int pos, Object o) throws Exception
{
    //..
}
private Object dynamicDispatchGetXxxMethod(
    ResultSet rs, int pos) throws Exception {
    //...
}
private String sqlTypeName(int type) {
    //...
}
} // OraclePersistentObject

```

```

private void setPrimaryKey() throws Exception {
    if (internalKey == null) {
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        java.util.Date date = new java.util.Date();
        // Not bullet-proof, however:
        String internalKey = String.valueOf(date.getTime());;
        try {
            con = cm.getConnection();
            stmt = con.createStatement();
            String sqlCommand = "INSERT INTO ";
            sqlCommand += tableName + " ( OID ) ";
            sqlCommand += "VALUES ( '" + internalKey + "' )";
            stmt.execute(sqlCommand);
        } catch (Exception e) {
            throw e;
        } finally {
            if (stmt != null) stmt.close();
            if (con != null) cm.close(con);
        }
    }
}
}

```

Implementing entries using Access differs. For example, we use a different subclass of `PrimaryKey` to handle primary keys, as well a different kinds or Java to SQL type mappings. An excerpt of class `AccessPersistentObject` might look like:

```

public class AccessPersistentObject extends PersistentObject {
    // utilities for conversions
    private HashMap javaSqlTypeMap = new HashMap();
    private HashMap sqlJavaTypeConversionMap = new HashMap();
    private HashMap javaSqlTypeConversionMap = new HashMap();
    private ConnectionManager cm;

    // Assume that internalKey is appropriately set in the
    // constructors.
    private String internalKey;

    //...

    public PrimaryKey getPrimaryKey() throws Exception {
        return new AccessPrimaryKey(internalKey);
    }
}

```

```

private void setupDefaultJavaSqlTypeMap() {
    javaSqlTypeMap.put(java.lang.String.class, "VARCHAR");
    javaSqlTypeMap.put(java.lang.Integer.class, "LONG");
    ...
}

private void setupDefaultSqlJavaTypeConversionMap() {...}
private void setupDefaultJavaSqlTypeConversionMap() {...}

// Conversion methods, not shown...

} // AccessTable

```

## Known Uses

Java AWT is built upon the Bridge pattern. JNDI uses the Bridge pattern in two ways:

- The JNDI service API (used by *clients*) corresponds to the `Abstraction (Entry)` class.
- The `Implementor (PersistentObject)` class defines the hooks for the implementor of a particular *service provider*.

## Related Patterns

- Abstract Factory and/or Factory Method can create and configure particular bridges.