Bringing OOAD&P Together: A Synthesis Approach

Daniel V. Goulet, Professor
Department of Mathematics and Computing
University of Wisconsin-Stevens Point
Stevens Point WI 54481 USA
dgoulet@uwsp.edu

Robert Dollinger, Associate Professor
Department of Mathematics and Computing
University of Wisconsin-Stevens Point
Stevens Point WI 54481 USA
rdolling@uwsp.edu

Abstract

Modern software development draws on many concepts, strategies, processes, tools, and techniques: 3-Tier Architecture, Model Driven Architecture, UML, Unified Process, visual modeling, visual programming, round-trip engineering, object-think, use case driven, incremental and iterative, documentable, etc. Each has a different objective. Each has a different point-of-view. Each has a different level of abstraction. None address the melding of these various 'ways of doing' software development into a cohesive and coherent, 'best-of-breed' approach to software development. Laying out a strategy that can fall along a continuum from waterfall to agile, the authors bring their OOAD & P 'best-of-breed' decisions to select components for a synthesized strategy that is incremental, iterative, traceable, documentable, and teachable to beginning undergraduate software developers.

KEYWORDS: Object-Oriented Analysis, Design & Programming. Software Development Strategies.

1. Introduction

Problem: Given the large number of concepts and strategies that have evolved and matured in the last ten years or so in object-oriented analysis, design and programming [OOAD&P], how does a professor organize this material into a meaningful, cohesive and consistent instructional approach for beginning software developers? We have larger organizational structures like 3-Tier Architecture (Satzinger, 2005); Model Driven Architecture [MDA] (Brown, 2004); Unified Modeling Language [UML] (Fowler et al., 2004); Unified Process [UP] (Jacobson et al., 1999. Krutchten, 2004). We have tools like Visual Studio .NET [VS .NET] (Johnson et al., 2003) ; Rational Software Architect [RSA] (Mittal, 2005), XDE .NET (Rational 2003). We have concepts like visual modeling, visual programming, round trip engineering, object-think (Satzinger et al., 2001; West, 2004); use case driven (Bittern, 2003; Cockburn, 2001); iterative, traceable, documentable (Manassis, 2004; Boogs et al., 2003). And we have texts like (Dennis et al., 2002; Doke et al., 2002 & 2003; Satzinger et al., 2001 & 2005; Schach, 2004). Each provides a view of what is and what can be in the area of OOAD & P. But, the authors, when working in the classroom, found gaps or points of disjuncture that did not fit the idea of a seamless, traceable strategy of developing a software system from the statement of a business problem to the production of code implementing a solution to that problem. Their discomfort resulted in a synthesis of the myriad of ideas and approaches listed above into their 'best-of-breed' strategy that is both teachable and effective in educating a next generation of software developers. In what follow, the components of the synthesis are identified

first, followed by an example structure, and finally populating the structure with a small example to complete the synthesis development.

2. Components of the Synthesis

Structural Architecture

The *3-Tier Model* approach appears as the design strategy in many places in the computing literature, e.g. (Satzinger et al. 2005). The essence of the approach is to divide a basic software system into three tiers that are loosely coupled through message passing. These three tiers are the (i) User Interface, (ii) Problem Domain, and for our purposes in this paper (iii) Data Source – persistent storage in the operating environment. The Appendix Figure 23 depicts these tiers, along with some additional features in each tier that will be addressed later in the paper and forms the Structural Architecture for the synthesis.

Model Architecture

Model Driven Architecture, articulated by (Brown, 2004), views software development from three different models: the Computational Independent Model [CIM]; the Platform Independent Model [PIM]; and the Platform Specific Model [PSM]. These models view the problem through three different levels of abstraction, from high level to detail. The CIM, PIM and PSM form the Model Architecture for the synthesis.

Solution Architecture

The Unified Process, Figure 1, developed by Jacobson-Booch-Raumbaugh (Jacobson et al. 1999) and extended through the work of Krutchten (Krutchen 2004), forms the basis for most modern day object-oriented software development. The authors have taken their life cycle model, modified it and simplified it for instruction to focus on the iterative and incremental nature of the Unified Process approach and the traceability requirement. The "iterative nature" of the life cycle will follow the MDA divisions of "Computational Independent Model", "Platform Independent Model", and "Platform Specific Model" (Brown 2004). Figure 2 exhibits the modification.

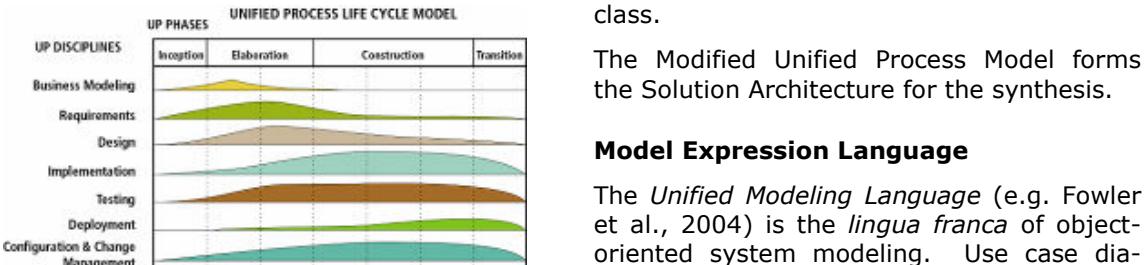


Figure 1: The Unified Process Life Cycle Model (Satzinger, 2005, p54)

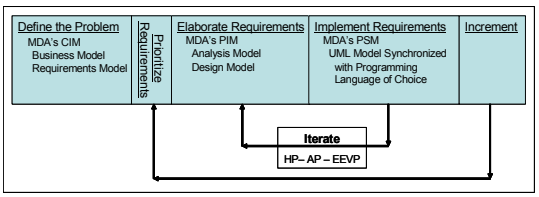


Figure 2: Modified Unified Process Model

The Requirements Phase focuses on the Computational Independent Model and has two iterations: (i) the Business Model, and (ii) the Requirements Model. The Elaboration Phase focuses on the Platform Independent Model and has two iterations: (i) the Analysis Model, and (ii) the Design Model. Each iteration adds more detail to the understanding and the solution of the problem. The Implementation Phase focuses on the Platform Specific Model. The iterative nature, here, will appear slightly different. In its most elemental form, iterations will occur with the iterative coding and testing of each structural segment of code, e.g., a class.

The Modified Unified Process Model forms the Solution Architecture for the synthesis.

Model Expression Language

The *Unified Modeling Language* (e.g. Fowler et al., 2004) is the *lingua franca* of object-oriented system modeling. Use case diagrams, class diagrams, and sequence diagrams are the foundational expressions for modeling object-oriented systems. The UML forms the Model Expression Language for the synthesis.

Model Package Structure

IBM Rational XDE .NET (Rational, 2003) and *Microsoft's Visual Studio .NET, version 2003*, together form the modeling package tool for the .NET language world, while *IBM Rational Solution Architect* is the modeling package tool for the Java world. Within these IDEs reside the UML models for the CIM and PIM, and the UML model and implementation coder of the PSM. Each IDE can be structured to explicitly show and contain the models in the Modified Unified Process Model. These IDEs form the Model Package Structure for the synthesis example.

3. Bringing the Synthesis Together

Basic Definition

The synthesis is based on the object-oriented approach / program defined by many authors as "a collection of interacting or collaborating objects", e.g. (Satzinger et al. 2005, pp 60). Figure 3 is a visual model of that definition.

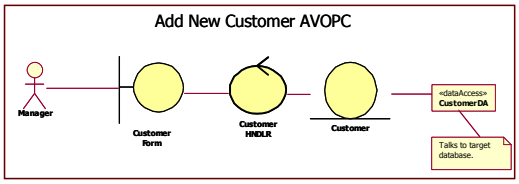


Figure 3: Collaborating Objects

Example Structure

For this paper, the authors develop a structure in the XDE .NET / VS .NET IDE (a similar structure exists for RSA). The structure is the repository for the components of the Model Architecture: the UML models for the CIM, PIM and the UML models and code for the PSM. The structure has embedded in its very fabric the ideas of iterative, incremental, traceable and documentable systems development. In an instructional setting, the authors develop the structure stepwise as the topics are introduced, bringing the new developer along in both an iterative and incremental way, with their work traceable from one step to the next.

CIM and PIM Structure

Create and name a 'Blank Solution' in VS. NET.

Add two XDE .NET blank 'Solution Items' and name one 'Computational Independent

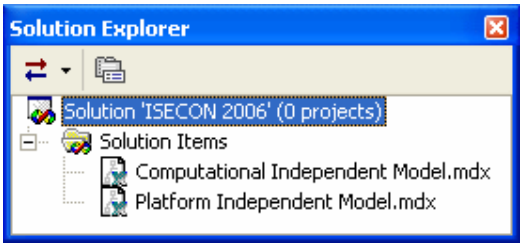


Figure 4: Solution Explorer for CIM & PIM

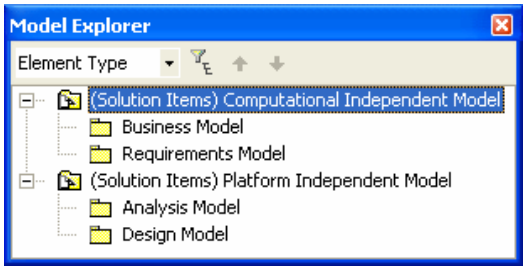


Figure 5: Model Explorer for CIM & PIM

Model' and the other 'Platform Independent Model' (Figure 4). The '.mdx' extension indicates these Solution Items are XDE .NET folders. Open the .mdx folder in Model Explorer and add packages for the Business Model, Requirements Model, Analysis Model and the Design Model, respectively (Figure 5).

Expand the Business Model package with sub-packages to hold the appropriate UML elements. Similarly, expand the Requirements Model package (Figure 6).

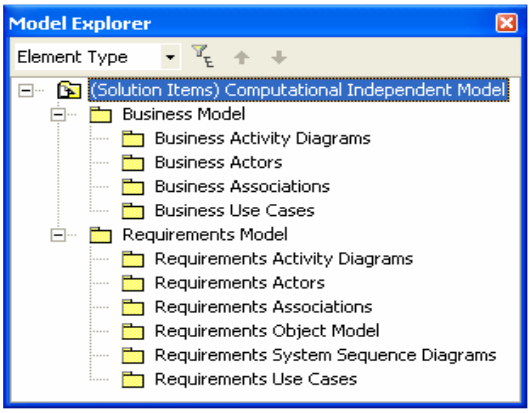


Figure 6: Expanded Business and Requirements Model for UML Elements

A note is in order for four of the sub-packages: The Business and Requirements

Association sub-packages are simply for house-keeping and reduce the clutter in the structure. They have no functional UML value.

The Requirements Object Model sub-package holds the domain class diagram for the initial classes identified in the software solution. The Requirements System Sequence Diagrams sub-package holds diagrams modeling the input and output messages between an actor and the system for a particular use case.

Expand the Analysis Model package with sub-packages to hold the appropriate UML elements (Figure 7). Similarly, expand the Design Model package.

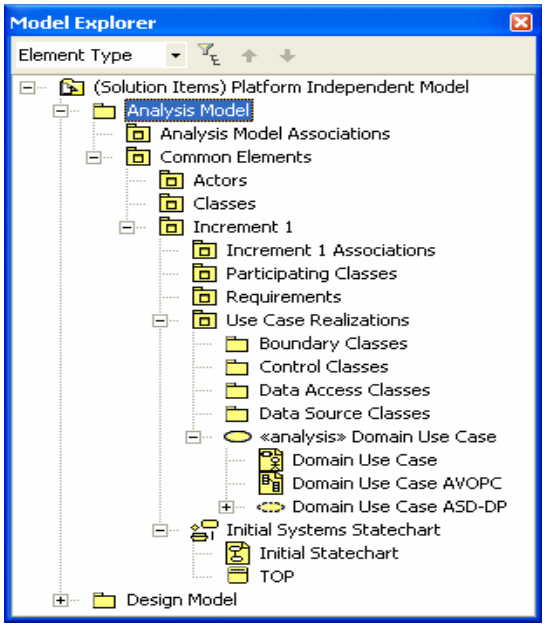


Figure 7: Expanded Analysis Model for UML Elements

PSM Structure

In VS .NET, the PSM is implemented through the creation of Projects within the current Solution holding the UML models. This is done in the normal way within VS .NET for project creation, e.g. a C# project.

The focus, here, is on the 2nd-Tier or Domain Layer. A similar activity occurs for the 1st and 3rd-Tiers or layers (Figure 8).

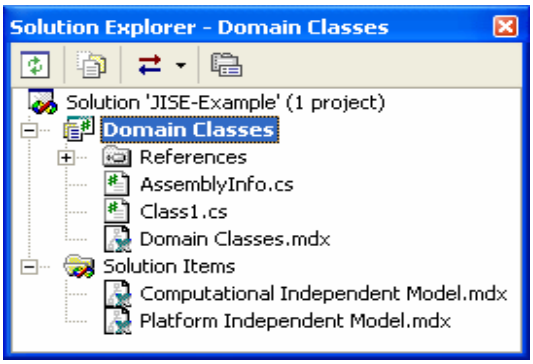


Figure 8: C# Project for Domain Classes

Next, the UML models created in the Design Model need to be 'hooked up' to the C# classes in the Domain Classes project. This is done using a three-step process: (i) Synchronize the C# Domain Classes Project producing an .mdx file that will link C# code with UML models (Figure 9). (ii) Open the Domain Classes.mdx file in Model Explorer (Figure 10).

Note, to also exhibit the flexibility of the synthesis, later in the development of the example a VB .NET implementation will be used for the UI and a C# implementation for the data storage.

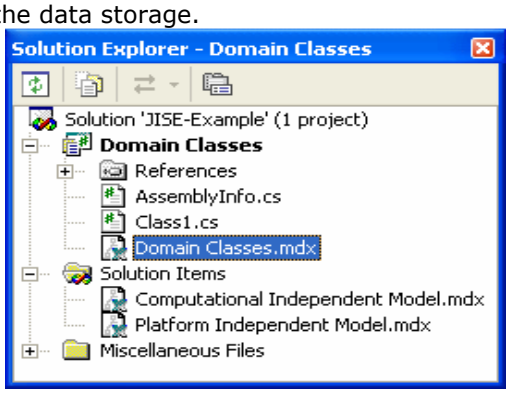


Figure 9: Synchronization of Domain Class Project

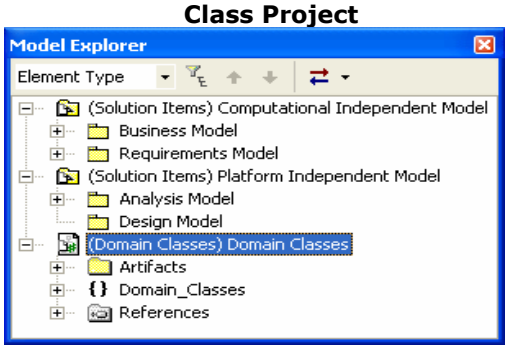


Figure 10: UML Structure Linking to Code

The package 'Artifacts' contains links to the C# code that appear in the C# Domain Classes Project. The Name Space, **{}** Domain_Classes, contains the design UML models. Copy the classes from the Design Model into the **{}** Domain_Classes name space. (iii) After completing activity (ii), and with the **{}** Domain_Classes name space selected, execute the command 'Generate Code'. At this point, not all the UML classes will generate code, since the Design Classes may contain notation not recognizable by the C# code generator, e.g., 'create' in the design class will not generate the constructor in the C# class. Correct the naming incompatibilities until the C# code generator completes the code generation process. At this point-in-time, C# code and UML models are hooked together. Round-trip engineering activities of 'Synchronize' and 'Generate Code' keep C# code and UML models in synch. Complete Model Explorer's organization of with a structure similar to the Design Model (Figure 11).

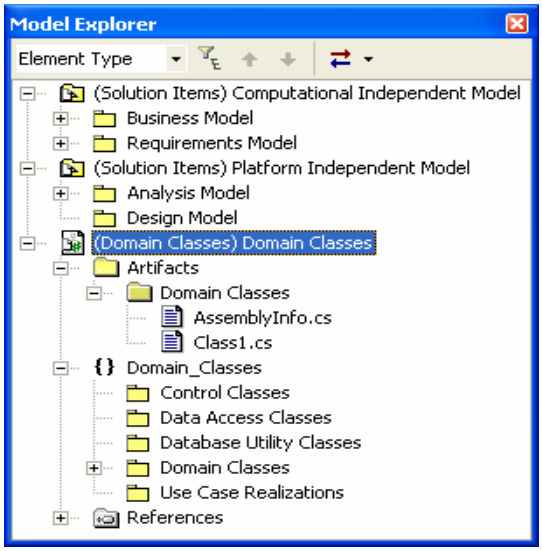


Figure 11: UML Code Model

4. Populating the Example Structure

Bradshaw Marina Example

The Bradshaw Marina case study forms the backdrop and example for populating the synthesis' example structure. It is simple enough to be understood at an introductory level, but complex enough to exhibit all the basic synthesis and OOAD & P concepts. The short version of the case study is that the Manager would like an information system that keeps track of leasing slips to customers for docking of boats. However, for simplification purposes, the 'Add New Customer' use case will be the only one developed and tracked after the initial setting of the problem via the Business and Requirements Models (Doke et al., 2003).

Computational Independent Model

Business Model: The objective of the Business Model is to clearly state the Business Objective for the problem being addressed and to identify the associated Business Process / Functions. The authors use a Word document template to guide the student in both the collection and organization of appropriate material. The completed Word template is stored with the VS .NET solution in a Documentation folder created in the VS .NET solution folder, so that all development artifacts travel as a unit – text documents, UML diagrams, and later, code. Figure 12 is skeleton version of the Business Model and Figure 13 exhibits the populated example structure.

Business Processes – Identified:

Relating to Leases

1. Customer leases a slip from Bradshaw Marina.
2. Customer transfers a lease to another slip.
3. Customer renews a lease for their current slip.

Relating to Customers

1. Manager creates a new Customer.
2. Customer changes some of their information.
3. Manager tracks Customers.

Relating to Boats

1. Customer registers a new boat [sailboat or powerboat].
2. Customer changes some information about a registered boat.

Relating to Docks and Slips

1. Manager adds a new dock with amenities.
2. Manager updates dock information.
3. Manager adds slip information for a dock.
4. Manager updates slip information for a dock.

Relating to Management Activities

1. Manager searches for a vacant slip.
2. Manager searches for a slip leased to a specific customer.
3. Manager receives standard operational reports on a regular basis.

Business Processes – Visual Model:
[See Appendix for Business Processes Visual Model Figure]

Add New Customer Business Use Case

Use Case Name: Add new customer
ID Number: 6
Use Case Type: Business
Stakeholders/Interests:
Customer – wants to join Bradshaw Marina
Manager – wants to increase customer base
Business Description:
1. Customer wants to join Bradshaw Marina.
2. Manager gives Customer a Customer Information Form.
3. Customer fills out form.
4. Manager verifies that the information is correct.
5. Manager files Customer Information Form in the Customer Folder.

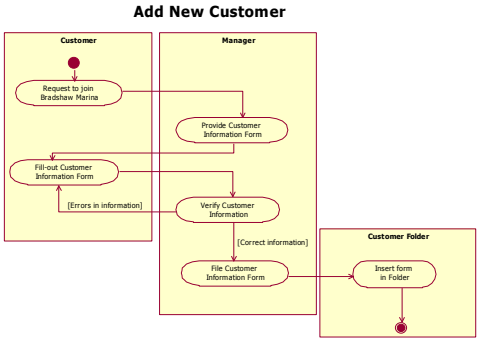


Figure 12: The Skeletal Bradshaw Marina Business Model

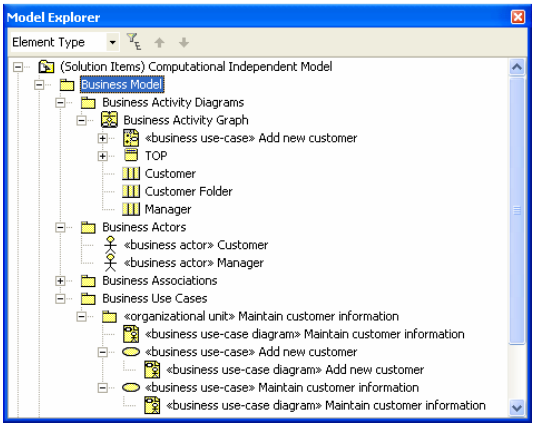


Figure 13: Completed Business Model Structure

Requirements Model: The objective of the Requirements Model is to identify those Business Processes that appear in the software solution. From the student's point-of-view, two conceptual things have happened: (i) the focus has changed from the business to the software solution, and (ii) a use case's initiator actor, 'requirements actor', has hands and "actually touch the automated system" (Satzinger, 2005; p 215).

Like the Business Model, the Requirements Model is a text listing of requirements and their associated UML models. The use case diagrams, descriptions and activity diagrams are updated to the requirements perspective and appropriately stereotyped.

Two modeling elements have been added to the structure of the Requirements model: (i) the Object Model (Appendix Figures 23 & 24) – that models the domain class diagram for the initial classes identified in the software solution; (ii) the System Sequence Diagrams (Appendix Figures 25 & 26) – that model input and output messages between an actor and the system for a particular use case (Satzinger, 2005; pp226-236)

Platform Independent Model

Analysis Model: The objective of the Analysis Model is to identify domain specific information as it relates to classes, i.e. attributes and custom methods, and to start fleshing out the 3-Tier, behavioral model from the point-of-view of the user-interface classes which are connected to or exchanging messages with the domain classes which are connected to or sharing messages with the data source.

The package structure contained in the Analysis Model is slightly different from the Business and Requirements Models – a difference set of modeling requirements and a different set of needs (Figure 14).

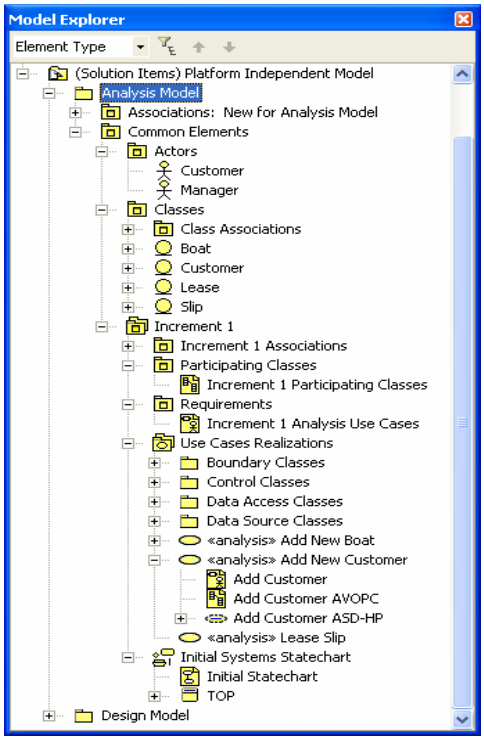


Figure 14: Analysis Model – Expanded

The new modeling elements that deserve attention are the Increment 1 Participating Classes (Appendix Figure 27), the Initial Systems Statechart (Appendix Figure 28), the Add Customer AVOPC [Analysis View Of Participating Classes] (Appendix Figure 29), and the Add Customer ASD-HP [Analysis Sequence Diagram – Happy Path] (Appendix Figure 30).

Taking them one at a time, the Increment 1 Participating Classes identifies a subset of requirements classes that are only needed to implement the Increment 1 Use Cases. The added detail information includes domain attributes, custom methods and visibility for each. The Initial Systems Statechart shows the overall communication structure of the software system as it moves from one state to another as a result of the occurrence of an event. The Add Customer AVOPC identifies those classes in the 3-Tier model that are needed in the execution of the Add Customer use case.

The Add Customer ASD-HP exhibits, visually for the first time, the statement that 'an object-oriented program is a collection of interacting or collaborating objects'.

Note that in the AVOPC and the ASD-HP diagrams, that the control class, CustomerHNDLR, and a facility for talking to the data source have been added. The messages in the ASD-HP have not been implemented as methods at this point, but act as a discovery activity for identifying addition methods required in the target objects.

Design Model: The objective of the Design Model, in general, is to add detail to the Analysis Model so that the resulting model can to be implemented on a target platform in a target programming language. The perspective within that general objective is to satisfy four sub-objectives: (i) to flesh-out the classes / class diagram; (ii) to add detail to the AVOPC; (iii) to convert the Analysis Sequence Diagrams to Design Sequence Diagrams with the appropriate expansion of interacting objects; and (iv) to convert and then connect the identified persistent classes with a persistent data source.

As part of the synthesis, a side trip into the 3-Tier model at the design level (Figure 1) is needed before proceeding. The object-oriented philosophies of encapsulation / responsibility and message passing are brought into play.

Working Appendix Figure 29 in a left-to-right direction, the actor only knows about the UI. The actor never interacts directly with the Problem Domain. The UI responds to events requested by the actor through its UI Event Handler, which only knows that it has to 'talk' to a Domain Handler, even though it appears that the UI is talking directly to the Domain Class.

The Domain Handler talks to the Domain Class and directs requests from the UI. The Domain Class has the responsibility to carry out the request passed to it by its Domain Handler. If the request is, for example, to place domain information into the Data Source, the Domain Class carries out that responsibility by talking to its data access class, the DomainDA. The action of the Domain Class is transparent to both the UI and the Data Source.

The DomainDA's responsibility is to properly form the information for interaction with the Data Source; in our above scenario, to construct a SQL Insert statement. The DomainDA passes the interaction request onto the manager for the Data Source, the DomainDM Class, responsible to execute the defined interaction request on the requested Data Source.

In this set of messages and structure, the actions of the User Interface are encapsulated and only loosely coupled to the Problem Domain; the actions of the Problem Domain are encapsulated and only loosely coupled to both the User Interface and the Data Source; and the actions of the Data Source are encapsulated and only loosely coupled to the Problem Domain. The actor 'feels' that it has interacted directly with the Data Source, but the encapsulation, loosely coupling and message passing says otherwise.

Flesh-out the Design Class Diagram: The Customer Class will be used by way of example.

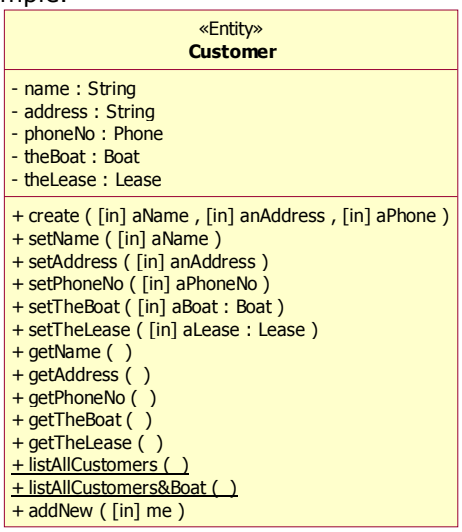


Figure 15: Customer Design Class

Adding detail to a class appears in two locations in XDE .NET. It appears on the visual model of the class (Figure 15), and in the Model Explorer statement of the class (Figure 16).

Add Detail to the AVOPC: The DVOPC appears in Appendix Figure 22 in its general form. For the Customer class example, insert 'frmAddNewCustomer' for UIForm, 'addNewCustomerEventHandler' for UIFormHNDLR, 'CustomerHNDLR' for DomainHNDLR, 'Cus tomer' for Domain Class, and 'CustomerDA' for DomainDA, and the Add New Customer DVOPC is complete.

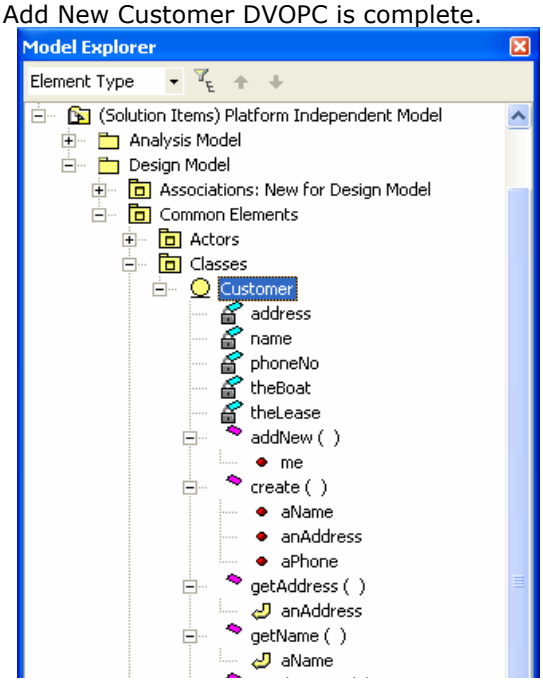


Figure 16: Portion of Expanded Customer Class Structure

Convert Analysis Sequence Diagram to Design Sequence Diagram: The Design Sequence Diagram (Appendix Figure 31) exhibits the most interesting set of changes and additional information in the synthesis. First, all objects from the DVOPC are present. Second, the messages, represent method invocations in the target object together with appropriate parame ters. The complete structure for the object-oriented program as a collection of collaborating objects is modeled.

Connectivity to the Data Source: In general, Domain Classes are representations of elements that need to be stored persistently. The lens, focusing on the 3rd – Tier or Data Access Layer, provides the information to make the connection between the application and the associated persistent storage or data source. The role of the Data Access Layer is two-fold: (i) provide the functionality for transferring data back and forth between the application and the data source, and (ii) fill the conceptual gap between the classes modeled in the 'Participating Classes' package and the classes used by the data source.

Platform Specific Model

The Structural Architecture, the Model Architecture and the Solution Architecture all come into complete focus in the development of the PSM. Here, the synthesis perspective of the 3-Tier Model is the driving force. The traceability thread through the CIM and PIM has provided UML models ready for implementation in UML models in the PSM. The iterative focus of the Modified Unified Process has added detail ending with a Design Model ready to be transformed into code.

The 1st – Tier or UI Layer: The 1st-Tier or UI Layer is the easiest to construct and implement. Since the connection to the Domain Layer is a message to the Domain Class Handler, any UI that can create this message will work, e.g., a Windows Form, a Web Form, or another system. Here, by way of example, a simple VB .NET Windows Form is used. The UIForm Handler in the original 3-Tier model is nothing other than the event handler in the Windows Form code that accepts a 'button click' (Figure 17).

frmAddNewCustomer

AddNewCustomerEventHandler

```
Private Sub btnAddCustomer_Click(ByVal
sender As System.Object, ByVal e As Sys-
tem.EventArgs) Handles btnAddCustomer.Click
    ' Get customer attributes for form's text
boxes
    customerName = txtName.Text
    customerAddress = txtAddress.Text
    customerPhone = txtPhone.Text
    ' Button click hands off event to Customer
Handler
    CustomerHNDLR.addNew(customerName,_
                customerAddress, customer-
Phone)
End Sub
```

Figure 17: Bradshaw Marina Add Customer-Form and Event Handler

The 2nd – Tier or Domain Layer: The UML Domain Classes of CustomerHNDLR and Customer are used to generate target language skeletal code, which is then populated with internal method logic. Figure 18 has code snippets for the CustomerHNDLR and Customer classes.

Customer Handler [abbreviated]

```
Public Class CustomerHNDLR
  Shared aCustomer As Customer
  Shared myDataManager As DataManager
  Public Shared Function addNew(
    ByVal aName As String, _
    ByVal anAddress As String, _
    ByVal aPhoneNo As String)
     aCustomer = New Customer(aName, _
          anAddress, _
          aPhoneNo)
   myDataManager =getDataManager()
  aCustomer.addNew(myDataManager)
  End Function
End Class
```

Customer Class [abbreviated]

```
Public Class Customer
  ' Attributes
   Private address As String
   Private name As String
   Private phoneNo As String
   Public Sub New( _
      ByVal aName As String, _
      ByVal anAddress As String, _
      ByVal aPhone As String)
    setName(aName)
    setAddress(anAddress)
    setPhoneNo(aPhone)
   End Sub


  ' Data Access Methods
   Public Sub addNew( _
     ByVal myDataManager As _
              DataManager)
    myDataManager.Save(Me)
   End Sub
End Class
```

Figure 18: VB .NET Code Snippets for the CustomerHNDLR and Customer Classes

The 3rd – Tier or Data Access Layer: Domain Classes need to be decoupled from the details of persistently storing or of retrieving objects, and the discussion for doing that is a little more detailed than for the 1st and 2nd Tiers, as there are several ap-

proaches possible. The Data Access Layer decouples the Domain Model from the specifics of persistent storage, which interposes a layer of Data Access Classes where the specifics of persistently saving, retrieving or updating of each type of object are dealt with. Two types of functionality are implemented by the Data Access Classes. The first deals with the specifics of the Domain Class corresponding to the Data Access Classes. The second deals with the specifics of the persistent storage where the object is saved. For example, as part of the first kind of functionality, the Data Access Class would take care of building the particular SQL string such that the relevant fields of an object would be inserted in a database table. For another Domain Class the SQL string will be different with a different Data Access Class building it.

A compact, one class based solution called the DataManager solves the problem. The DataManager class is a useful abstraction exposing the two kinds of functions a Data Layer Class implements: (i) functions specific to the application, and (ii) functions specific to the data source used for persistent storage. The understanding that one can separate the two kinds of functions substantially simplifies and almost completely automates the development of the DataManager class and its specific subclasses. A two step approach is needed: First, factor out the data source specific functionality into a sub-layer which is completely independent of the specifics of the entity types of the given application. Second, use reflection and dynamic code generation techniques (Troelsen, 2003) to automatically generate code for the basic persistency related operations associated to the application's entity types. This results in a data access class generator. The advantages of the approach are two-fold. The first step creates a prefabricated DataManager class capturing the specifics of dealing with the data source used as the persistent storage for the application's data. The second step insures application independence of the DataManager, which acts as a code generator with the Domain Class used as parameter to create the specific code.

The DataManager Abstract Class: The DataManager abstract class is a nice way to provide a uniform access to datasource related functionality. Most of its methods are abstract and are implemented in specialized subclasses of the abstract class. All applications refer to this class in a manner that is independent of the specifics of the application itself or of the data source. The functionality of the DataManager can be defined at a platform independent level and defines two categories of methods. The first category refers to the data source and defines general purpose operations related to the housekeeping of all data sources like: Connect(), Open(), Close(), etc. Each of these methods will be implemented in a specific way by the subclasses inheriting from the DataManager class. The second category is application related and includes methods like: Save(), Modify(), Delete(), Retrieve(). The code for these methods will be generated automatically by the specialized subclasses in order to address both the specifics of the application and the specifics of the data source type.

The partial code for the DataManager abstract class is given in Figure 19.

```
namespace DataManagers
{
   public class DataManager
   {
      //data source related methods
      public abstract void Connect(String dataSource);
      public abstract void Open();
      public abstract void Close();
      ...
      //application related methods
      public abstract void Save(Object entityInstance);
      public abstract void Modify(
                  Object entityInstance);
      public abstract void Delete(
                  Object entityInstanceKey);
      public abstract void Retrieve(
                  Object entityInstanceKey);
      ...
   }
                              }
```

Figure 19: Implementation Layout of the Abstract DataManager Class

Observe the two categories of abstract methods as previously described: data source related and application related methods. Applications never directly create an instance of the DataManager abstract class; instead instances of its specializations are created. However, all calls to data source related functions would be expressed in

terms of themethods defined by the DataManager abstract class.

Managing the Data Source: Part of the functionality of the abstract DataManager is application independent, which means that it can be prefabricated and reused across applications, and is specific to the type of the data source used to persistently store the data. It is represented at the level of the abstract DataManager class by functions like: locate/connect, login/authenticate, open data source/connection, send data, receive data, close data source/connection, analyze and report errors. There can be several types of data sources: flat files, relational databases, XML data sources, remote data sources represented by a proxy etc. Each and every type has its own implementation of these abstract functions, which means that a specialized subclass is defined for each type of data source.

```
namespace DataManagers
{
  public enum ConnectionType
                    {Odbc,OleDb,Oracle,Sql}
  public class DBDataManager:DataManager
  {
     // override data source related methods
     public override void Connect(String dataSource)
     {
     //infer connection type
      switch(connectionType){
          case ConnectionType.Odbc:
          //create connection and command for
ODBC
          case ConnectionType.OleDB:
          //create connection and command for
OleDB
            ...
          }
     }
     public override void Open();
     public override void Close();
     //override application related methods
     public override void Save(
                    Object entityInstance){…}
     public override void Modify(
                    Object entityInstance){…}
     public override void Delete(
                     Object entityInstance){…}
     public override void Retrieve(
                     Object entityKey){…}
     //data source type specific methods
     public void ExecuteSQLString(String SqlString)
     {
          this.connection.Open();
          this.command.CommandText=SqlString;
          try{
               this.command.ExecuteNonQuery();
          }catch(Exception ex)
               this.errorMessage=ex.Message;
          this.connection.Close();
     }
```

Figure 20: Implementation Layout of the DBDataManager Class

The implementation for a version of the DataManager class for relational databases is the DBDataManager class shown in Figure 20.

The DBDataManager class inherits from the DataManager abstract class and provides implementations (overrides) for all abstract methods defined in the base class. Also some data source specific methods are provided like ExecuteSQLString(). By instantiating the DBDataManager class with the right parameter and calling the right methods, the basic database related tasks are performed for connecting to, opening and closing the database, etc.

Managing the Application: Managing the application is more challenging in that new code needs to be automatically generated in each application in order to capture the specifics of the Domain Classes. The same code cannot be used over several applications as in the case of the data source functions. The specifics of an available data source can be known ahead of time before any application would be developed, which is not the case with the specifics of the Domain Classes. What can done is to automate the process of writing the code for these methods, which means that our prefabricated DataManager class would act as a code generator and would be used to automatically produce the code for saving, deleting, updating or retrieving a Domain Class instance given as a parameter. The code is application dependent, and is different for each and every Domain Type, since specific data members and properties have to be dealt with. The issue is addressed by identifying at run time the type of the object given as a parameter and by using reflection techniques to reveal the structure of the object. The code to be generated is also data source specific, that is, each implementation of the abstract Data Manager class will generate its code for a function like Save(entityInstance)

in a different way. The sample code for the Save(entityInstance) method of the DBDataManager subclass is given in Figure 21.

The code generates the INSERT SQL statement to save whatever object is given as a parameter into the proper database table. After building the INSERT statement, the Save() method makes a call to the ExecuteSQLString(sqlString) method with the SQL string as parameter in order to actually save the object.

Using the DataManager Class Implementing the 3-Tier Model: Execution of the Add New Customer Use Case: With DataManager class and its implementations for various data source types properly developed, it becomes easy to build persistence related functionality in applications. The application will create one or more instances of the DataManager's subclasses

```
public void Save(Object entityInstance)
{
      Type classType=entityInstance.GetType();
      string sqlString="INSERT ";
      sqlString+=classType.Name;
      sqlString+="(";
      //generate comma separated list of names
      PropertyInfo[]
pInfo=classType.GetProperties();
      int i;
      for(i=0;i<pInfo.Length-1;i++)
            sqlString+=pInfo[i].Name+",";
      sqlString+=pInfo[i].Name+")";
      sqlString+=" VALUES (";
      //generate comma separated list of values
      for(i=0;i<pInfo.Length-1;i++)
        if(pInfo[i].PropertyType==
                        sqlString.GetType())
      sqlString+="'"+pInfo[i].GetValue(
                entityInstance, new Object[]{})+"',";
        else
          qlString+=pInfo[i].GetValue(
                entityInstance, new Object[]{})+",";
      if(pInfo[i].PropertyType==sqlString.GetType())
        sqlString+="'"+pInfo[i].GetValue(
                entityInstance, new Object[]{})+"')";
      else
        sqlString+=pInfo[i].GetValue(
                entityInstance, new Object[]{})+")";
      ExecuteSQLString(sqlString);
}
```

Figure 21: Implementation Layout of the Save Function in the Case of a Relational Database Used as Data Source

according to the types of data sources used in the application. The code inside the entity classes will be entirely data source independent, since all related operations will refer to the abstract methods defined in the DataManager class, called through the right data manager object, instance of one of the FlatFileDataManager, XMLDataManager or DBDataManager classes. Take for example, the VB .NET application summarized in the code snippets of Figure 17. The frmAddNewCustomer form accepts the Customer information, and upon the 'Add Customer' button clip, invokes the AddNewCustomerEventHandler, which scrapes the screen and passes this information onto the CustomerHNDLR, which creates a Customer object and it will have it to save himself by a call to his addNew() method.

5. Summary and conclusions

Working with the myriad of concepts, processes and tools is a daunting task when trying to create an understandable learning environment for undergraduate OOAD & P students. Though not perfect and with still important issues remaining to be solved for proper code structuring, the authors propose a synthesis of many of these concepts, processes and tools to create an approach that is iterative, incremental, traceable, documentable and understandable for next generation software developers. It solves four of the most vexing problems in teaching the OO process, namely, (i) the traceability of the development process from a business problem to code implementation, (ii) the disjunction between UML modeling and code generation, (ii) the process of keeping model documentation in sync with code implementation, and (iv) resolving the disjuncture when going from Domain Classes to persistent storage. The key has been the recognition that, as developers, we view a systems development project from many perspectives and through many lenses. In recognizing that each perspective or lens has it strengths and limitations, and that combining the best of which each has to offer, a cohesive and coherent instructional environment is possible.

6. References

Bittner, Kurt. Spence, Ian. *Use Case Modeling*. Addison-Wesley. Boston, MA. 2003.

Boggs, Wendy. Boggs, Michael. *Master Rational XDE*. Sybex. San Francisco, CA. 2003.

Brown, Alan. *An Introduction to Model Drive Architecture, Part I: MDA and Today's Systems.*The Rational Edge. February 2004 http://www-128.ibm.com/developerworks/rational/library/3100.html .

Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley. Boston, MA. 2001.

Doke, E. Reed. Satzinger, John W. Williams, Susan Rebstock. Douglas, David E. *Object-Oriented Application Development using Microsoft Visual Basic .NET.* Course Technology. Boston, MA. 2003.

Doke, E. Reed. Satzinger, John W. Williams, Susan Rebstock. *Object-Oriented Application Development using Java.* Course Technology. Boston, MA. 2002.

Dollinger, Robert. Goulet, Daniel V. Gibbs, David. "Structuring Databases from UML to Code". Proceedings of ED-MEDIA 2005 – World Conference on Educational Multimedia, Hypermedia & Telecommunications. June 27-July 2, 2005. Montreal, Canada.

Fowler, Martin. Scott, Kendall. *UML Distilled, 3e.* Addison-Wesley. Boston, MA. 2004.

Jacobon, Ivar. Booch, Grady, Rumbaugh, James. *The Unified Software Development Process.* Addison-Wesley. Boston, MA. 1999

Johnson, Brian. Skibo, Craig. Young, Marc. *Inside Microsoft Visual Studio .NET 2003*. Microsoft Press. Redmond, WA. 2003.

Krutchten, Philippe. *The Rational Unified Process: An Introduction.* Addison-Wesley. Boston, MA. 2004.

Manassis, Enricos. *Practical Software Engineering: Analysis and Design for the .NET Platform.* Addison-Wesley. Boston, MA. 2004.

Mittal, Kunal. "Introducing IBM Rational Software Architect". http://www-128.ibm.com/developerworks/rational/library/05/kunal/. 15 February 2005.

Rational XDE Model Structure Guidelines for Microsoft .NET. IBM Staff paper. http://www-128.ibm.com/developerworks/rational/library/content/03July/2500/2554/2554_net.pdf May 2003 version.

Satzinger, John W. Jackson, Robert B. Burd, Stephen D. *Object-Oriented Analysis and Design with the Unified Process*. Course Technology. Boston, MA. 2005.

Satzinger, John W. Orvik, Tore U. *The Object-Oriented Approach.* Course Technology. Boston, MA. 2001

Schach, Stephen R. *Introduction to Object-Oriented Analysis and Design with UML and the Unified Process.* McGraw Hill. Boston, MA. 2004.

Troelsen, Andrew. *C# and the .NET Platform, 2e.* Apress. Berkeley, CA. 2003.

West, David. *Object Thinking*. Microsoft Press. Redmond, WA. 2004.

Appendix

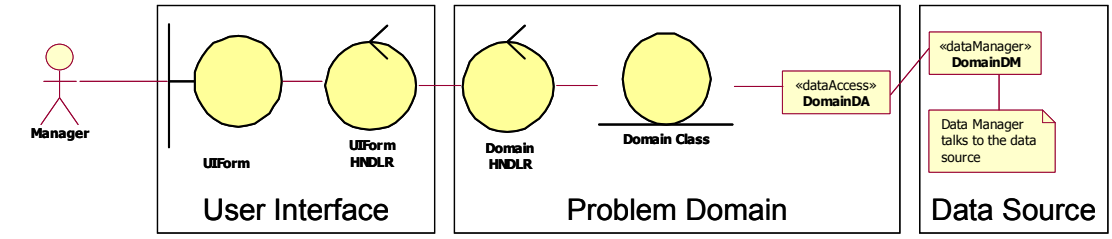
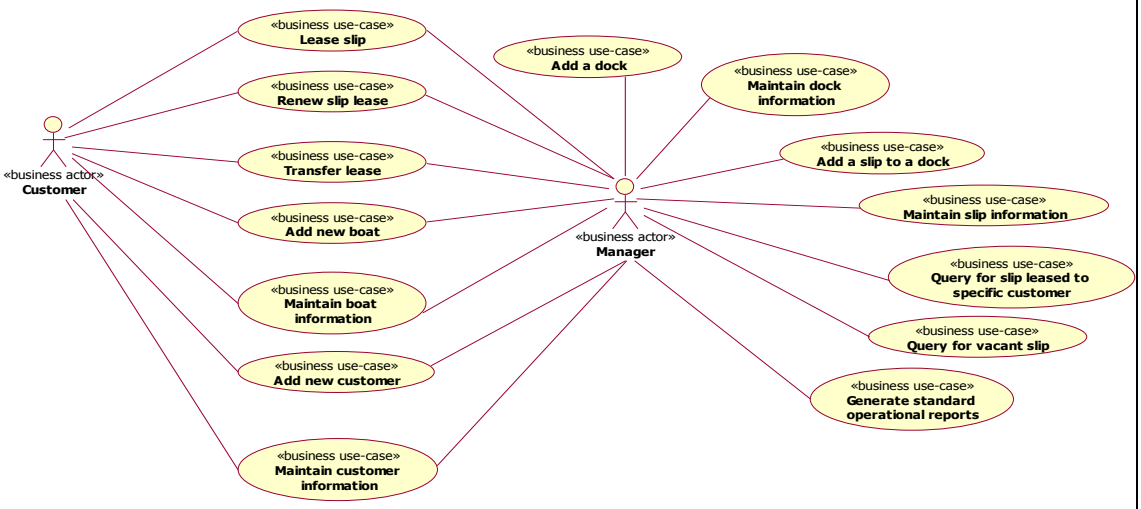


Figure 22: 3-Tier Model Example



Business Process – Visual Model for Figure 13

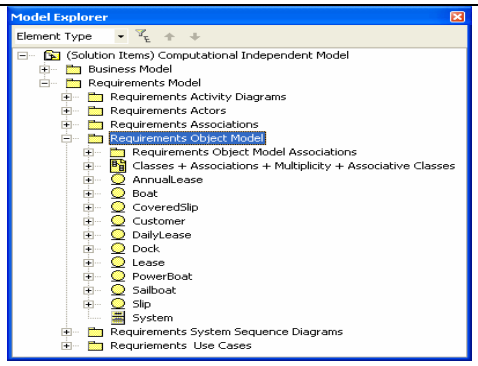


Figure 23: Object Model: Structure

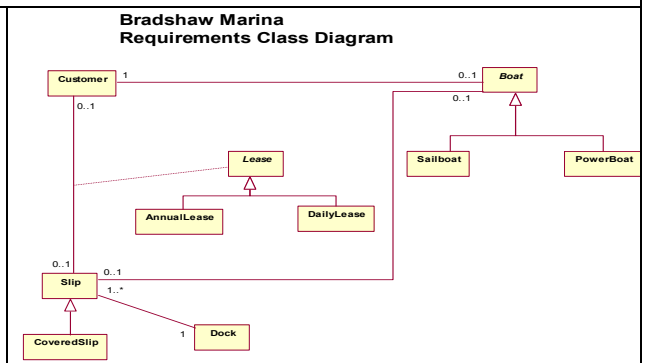


Figure 24: Object Model: Visual

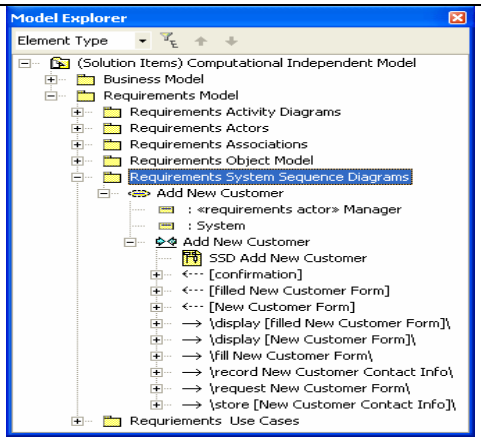


Figure 25: SS Diagram: Structure

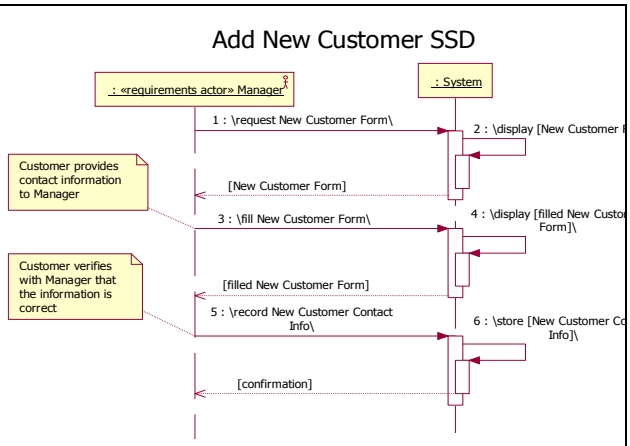


Figure 26: Systems Sequence Diagram: Visual

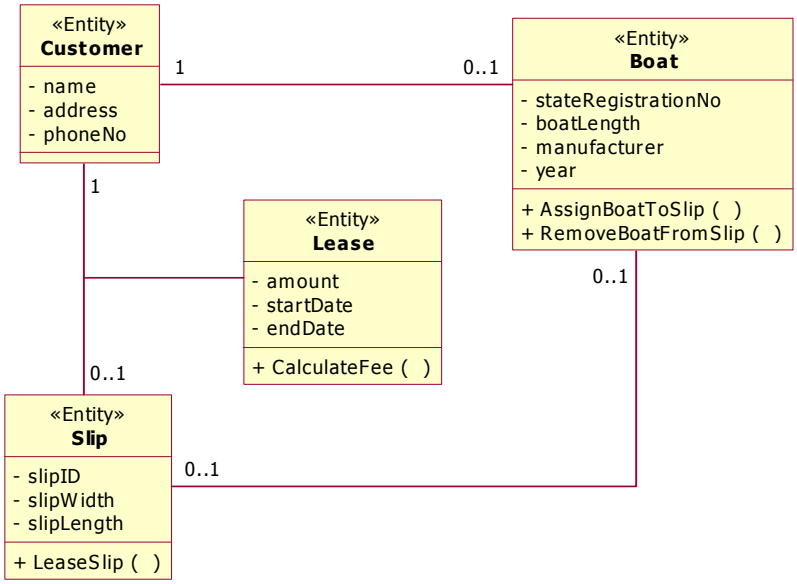


Figure 27: 1st Increment Class Diagram

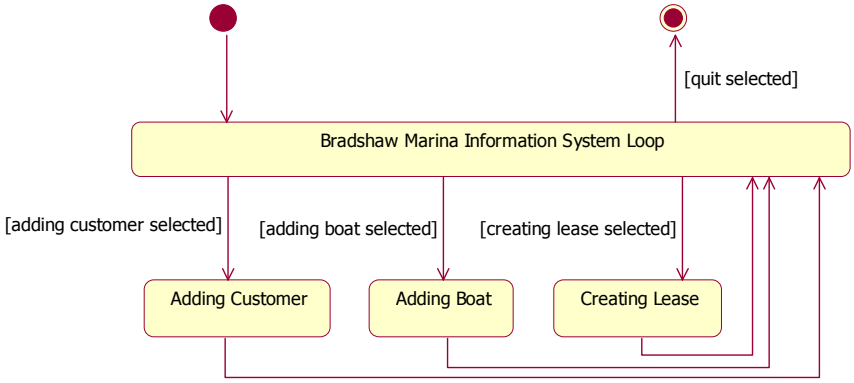


Figure 28: Initial Systems Statechart

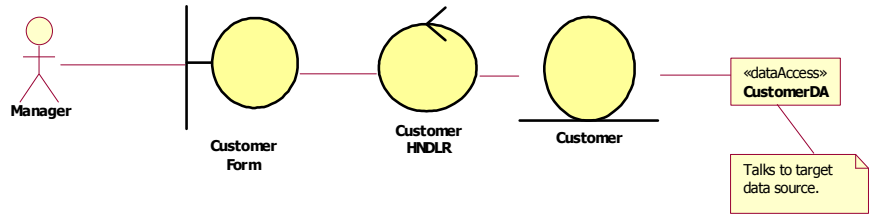


Figure 29: Add New Customer AVOP

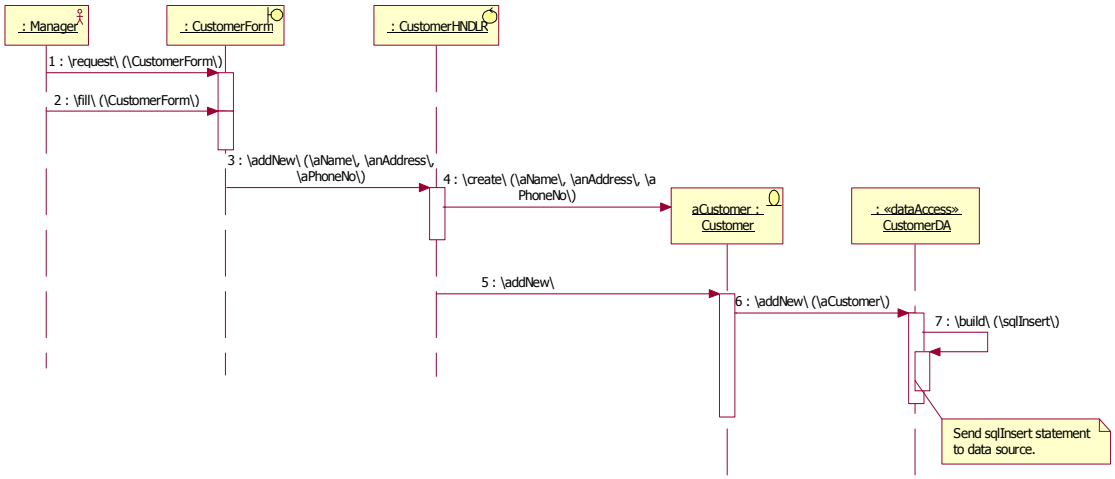


Figure 30: Add New Customer [Analysis Sequence Diagram – HP]

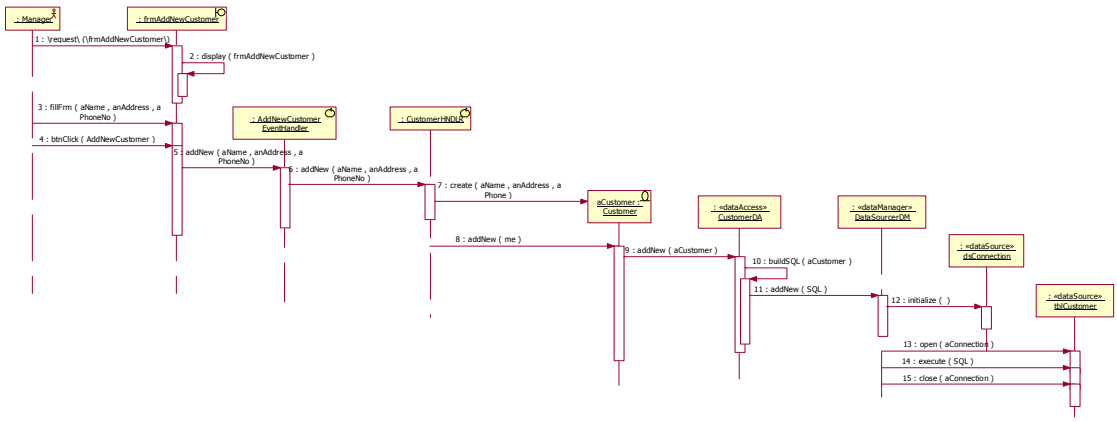


Figure 31: Design Sequence Diagram – Add New Customer