

Java Education & Development Initiative

Software Engineering

Version 1.2
July 3, 2006

Author

Ma. Rowena C. Solamo

Team

Jaqueleine Antonio
Naveen Asrani
Doris Chen
Oliver de Guzman
Rommel Feria
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson
Daniel Villafuerte

Requirements For the Laboratory Exercises**Minimum Hardware Configuration**

- **Microsoft Windows operating systems:**
 - **Processor:** 500 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Solaris™ operating system:**
 - **Processor:** 450 MHz UltraTM 10 workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Linux operating system:**
 - **Processor:** 500 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space

Recommended Hardware Configuration

- **Microsoft Windows operating systems:**
 - **Processor:** 780 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Solaris™ operating system:**
 - **Processor:** 500 MHz UltraTM 60 workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Linux operating system:**
 - **Processor:** 800 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space

Operating System

Below is a list of operating systems that Sun JavaTM Studio Enterprise 8 runs on.

- Solaris 9 and 10 Operating Systems on SPARC Platform
- Solaris 10 Operating System on x86 and AMDx64 Platform
- Windows XP and Windows 2000

Sun Java Studio Enterprise 8 is provided, but not supported on the following operating systems:

- Linux

Software

Sun Java Studio Enterprise 8 runs on J2SE JDK 5.0 (Java™ 2 JDK, Standard Edition), which consists of the Java Runtime Environment plus developers tools for compiling, debugging and running application written in the Java™ language.

For more information, please visit <http://developers.sun.com>.

Table of Contents

1	Introduction to Software Engineering.....	7
1.1	Software Engineering- A Layered View.....	7
1.1.1	Quality Focus.....	8
1.1.2	Process.....	8
1.1.3	Method.....	8
1.1.4	Tools.....	8
1.2	Quality within the Development Effort.....	10
1.2.1	What is quality?.....	10
1.2.2	How do we define quality?	10
1.2.3	How do we address the Quality Issues?.....	11
1.3	Software Quality Assurance and Techniques.....	12
1.3.1	Software Quality.....	12
1.3.2	Characteristics of a Well-engineered Software.....	12
1.3.3	Software Quality Assurance Activities.....	13
1.3.4	Formal Technical Reviews.....	13
1.4	The Software Process.....	17
1.4.1	Types of Software Process Models.....	18
1.4.2	Factors that Affect the Choice of Process Model.....	24
1.5	Understanding Systems.....	25
1.6	Understanding People in the Development Effort.....	29
1.6.1	End-users.....	29
1.6.2	Development Team.....	31
1.7	Documentation in the Development Effort.....	32
1.7.1	What is documentation?.....	32
1.7.2	Criteria for Measuring Usability of Documents.....	33
1.7.3	Important of Documents and Manuals.....	34
1.8	Exercises.....	35
1.8.1	Specifying Boundaries.....	35
1.8.2	Practicing the Walkthrough.....	35
1.9	Project Assignment.....	35
2	Object-oriented Software Engineering.....	36
2.1	Review of Object-oriented Concepts.....	36
2.1.1	Abstraction.....	37
2.1.2	Encapsulation.....	38
2.1.3	Modularity.....	38
2.1.4	Hierarchy.....	39
2.2	Object-oriented Process Model.....	43
2.3	Object-oriented Analysis and Design.....	44
2.3.1	Object-oriented Analysis.....	44
2.3.2	Object-oriented Design.....	45
2.4	Unified Modeling Language (UML).....	48
2.4.1	Modeling Activity.....	48
2.4.2	UML Baseline Diagrams.....	51
3	Requirements Engineering.....	61
3.1	Requirements Engineering Concepts.....	61
3.2	Requirements Engineering Tasks.....	62
3.2.1	Inception.....	62
3.2.2	Elicitation.....	64
3.2.3	Elaboration.....	67
3.2.4	Negotiation.....	67
3.2.5	Specification.....	68

3.2.6 Validation.....	68
3.2.7 Management.....	69
3.3 Requirements Analysis and Model.....	70
3.3.1 The Requirements Model.....	70
3.3.2 Scenario Modeling.....	71
3.3.3 Requirements Model Validation Checklist.....	80
3.4 Requirements Specifications.....	82
3.4.1 The Analysis Model.....	82
3.4.2 The Use Case Analysis Technique.....	83
3.4.3 Analysis Model Validation Checklist.....	111
3.5 Requirements Traceability Matrix (RTM).....	112
3.6 Requirements Metrics.....	115
3.7 Exercises.....	118
3.7.1 Creating the Requirements Model	118
3.7.2 Creating Analysis Model.....	118
3.8 Project Assignment.....	118
4 Design Engineering.....	120
4.1 Design Engineering Concepts.....	120
4.1.1 Design Concepts.....	121
4.1.2 The Design Model.....	122
4.2 Software Architecture.....	124
4.2.1 Describing the Package Diagram.....	124
4.2.2 Subsystems and Interfaces.....	125
4.2.3 Developing the Architectural Design.....	130
4.2.4 Software Architecture Validation Checklist.....	137
4.3 Design Patterns.....	138
4.3.1 Model-View-Controller Design Pattern.....	155
4.4 Data Design.....	157
4.4.1 JDBC Design Pattern	158
4.4.2 Developing the Data Design Model	163
4.5 Interface Design.....	167
4.5.1 Report Design.....	167
4.5.2 Forms Design.....	170
4.5.3 Screen and Dialog Design	172
4.6 Component-level Design.....	190
4.6.1 Basic Component Design Principles.....	190
4.6.2 Component-level Design Guidelines.....	190
4.6.3 Component Diagram.....	191
4.6.4 Developing the Software Component.....	191
4.7 Deployment-level Design.....	196
4.7.1 Deployment Diagram Notation.....	196
4.7.2 Developing the Deployment Model	196
4.8 Design Model Validation Checklist.....	197
4.9 Mapping the Design Deliverables to the Requirements Traceability Matrix.....	198
4.10 Design Metrics.....	199
4.11 Exercises.....	201
4.11.1 Creating the Data Design Model	201
4.11.2 Creating the Interface Design.....	201
4.11.3 Creating the Control Design.....	201
4.12 Project Assignment.....	202
5 Implementation.....	203
5.1 Programming Standards and Procedures.....	203
5.2 Programming Guidelines.....	204

5.2.1	Using Pseudocodes.....	204
5.2.2	Control Structure Guidelines.....	204
5.2.3	Documentation Guidelines.....	204
5.3	Implementing Packages.....	206
5.4	Implementing Controllers.....	208
5.4.1	Review on Abstract Classes and Interfaces.....	208
5.4.2	Abstract Classes.....	208
5.4.3	Interfaces.....	210
5.4.4	Why do we use Interfaces?.....	210
5.4.5	Interface vs. Abstract Class.....	210
5.4.6	Interface vs. Class.....	211
5.4.7	Creating Interfaces.....	211
5.4.8	Relationship of an Interface to a Class.....	213
5.4.9	Inheritance among Interfaces.....	213
5.5	Implementing Java Database Connectivity (JDBC).....	217
5.6	Implementing the Graphical User Interface.....	220
5.6.1	AWT GUI Components.....	220
5.6.2	Layout Managers.....	224
5.6.3	Swing GUI Components.....	230
5.7	Controlling the Version of the Software.....	261
5.8	Mapping Implementation Deliverables with the Requirements Traceability Matrix...	266
5.9	Implementation Metrics.....	266
5.10	Exercises.....	267
5.10.1	Defining the Internal Documentation Format.....	267
5.11	Project Assignment.....	267
6	Software Testing.....	268
6.1	Introduction to Software Testing.....	268
6.2	Software Test Case Design Methods.....	269
6.2.1	White-Box Testing Techniques.....	269
6.2.2	Black-Box Testing Techniques.....	275
6.3	Testing your Programs.....	278
6.4	Test-driven Development Methodology.....	282
6.4.1	Test-driven Development Steps.....	283
6.4.2	Testing Java Classes with JUnit.....	285
6.5	Testing the System.....	292
6.6	Mapping the Software Testing Deliverable to the RTM.....	297
6.7	Test Metrics.....	298
6.8	Exercises.....	299
6.8.1	Specifying a Test Case.....	299
6.8.2	Specifying System Test Cases.....	299
6.9	Project Assignment.....	299
7	Introduction to Software Project Management.....	300
7.1	Software Project Management.....	300
7.2	Problem Identification and Definition.....	302
7.3	Project Organization.....	305
7.3.1	The Project Team Structure	306
7.3.2	Project Responsibility Chart.....	307
7.4	Project Scheduling.....	309
7.4.1	Project Work Breakdown Structure (WBS).....	309
7.4.2	Work Breakdown Schedule Format.....	310
7.5	Project Resource Allocation.....	313
7.5.1	Resource Availability Data Base.....	313

7.6 Software Metrics.....	315
7.6.1 Size-oriented Metrics- Lines of Codes (LOC).....	315
7.6.2 Function-Oriented Metrics: Function Points (FP).....	316
7.6.3 Reconciling LOC and FP Metrics.....	318
7.7 Project Estimations.....	319
7.8 Writing the Project Plan.....	320
7.9 Risk Management.....	321
7.9.1 The Risk Table.....	322
7.9.2 Risk Identification Checklist.....	323
7.10 Software Configuration Management.....	327
7.10.1 Baseline.....	328
7.10.2 Software Configuration Tasks.....	328
7.11 Project Assignment.....	331
8 Software Development Tools.....	332
8.1 Case Tools.....	332
8.2 Compilers, Interpreters and Run-time Support.....	332
8.3 Visual Editors.....	332
8.4 Integrated Development Environment (IDE).....	332
8.5 Configuration Management.....	333
8.6 Database Management Tools.....	333
8.7 Testing Tools.....	333
8.8 Installation Tools.....	333
8.9 Conversion Tools.....	333
8.10 Document Generator.....	333
8.11 Software Project Management.....	333
Appendix A – Installing Java Studio Enterprise 8.....	334

1 Introduction to Software Engineering

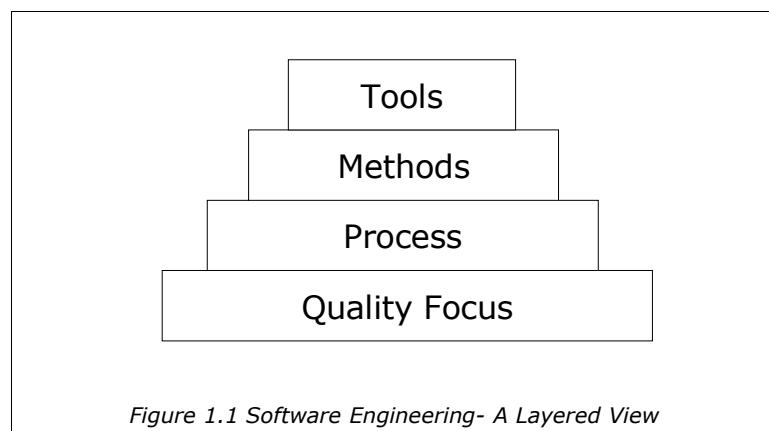
When people think about computers, the first thing that comes to their minds are the physical machines- monitor, keyboard, mouse and CPU. However, the software is the one that makes them useful. A **computer software** includes a *set of programs* that execute within a computer of any size and architecture, and *data* that are being processed by the programs and presented to users as hard or soft copies. It is built by software engineers through the employment of a software process that yields high-quality work products that meet the needs of people who will be using the system.

Nowadays, software is a very important technology of our lives because it affects nearly every aspects of it, including government, commerce, and culture. In this chapter, we will be discussing software engineering as a discipline in building quality computer software. A layered view will be used to outline the concepts needed to understand software engineering. Then, an understanding on the people involved in the software development effort will be discussed. It will be followed by the need for documentation and how to organize and document software engineering work products.

1.1 Software Engineering- A Layered View

Software Engineering is a discipline that applies principles of engineering to the development of quality software in a timely and cost-effective manner. It uses an approach that is systematic and methodological to produce quantifiable results. It makes use of measurement and metrics to assess quality, not only of the software but also the software process. They are also used to assess and manage the software development project.

Software Engineering is viewed differently by different practitioners. Pressman suggests to view software engineering as a layered technology¹. This view consists of four layers, namely, quality focus, process, methods and tools. Figure 1.1 illustrates this software engineering view.



¹ Pressman, Roger S., *Software Engineering, A Practitioner's Approach*, Sixth Edition, (Singapore: McGraw-Hill Internal Edition, 2005), p. 53-54

1.1.1 Quality Focus

At the very foundation of this layer is a total **focus on quality**. It is a culture where commitment to continuous improvement on the software development process is fostered. This culture enables the development of more effective approaches to software engineering.

1.1.2 Process

The **process** integrates the other layers together. It defines a framework that consists of key process areas that define and enable rational and timely delivery of the computer software. The key process areas are the basis for the software project management. They establish what technical methods are applied, what tools are used, what work products need to be produced, and what milestones are defined. They also include assurance that quality is maintained, and that change is properly controlled and managed.

1.1.3 Method

Methods define a systematic and orderly procedures of building software. They provide an overall framework within which activities of the software engineer are performed. These activities include a wide array of tasks such as requirements analysis, design, program construction, testing and maintenance.

Methodology is the science of systematic thinking using the methods or procedures used in a particular discipline. There are several software engineering methodologies that are used today. Some of them are briefly enumerated below.

Structured Methodologies:

- Information Engineering
- Software Development Life Cycle/Project Life Cycle
- Rapid Application Development Methodology
- Joint Application Development Methodology
- CASE*Method

Object-oriented Methodologies:

- Booch Method
- Coad and Yourdon Method
- Jacobson Method
- Rumbaugh Method
- Wirfs-Brock Method

1.1.4 Tools

Tools provide support to the process and methods. Computer-aided software engineering provides a system of support to the software development project where information created by one tool can be used by another. They may be automated or semi-automated.

Most tools are used to develop models. Models are patterns of something to make or they are simplification of things. There are two models that are generally developed by

a software engineer, specifically, the system model and the software model. The **system model** is an inexpensive representation of a complex system that one needs to study while a **software model** is a blueprint of the software that needs to be built. Like methodologies, several modeling tools are used to represent systems and software. Some of them are briefly enumerated below.

Structured Approach Modeling Tools:

- Entity-relationship Diagrams
- Data Flow Diagrams
- Structured English or Pseudocodes
- Flow Charts.

Object-oriented Approach Modeling Tools:

- Unified Modeling Language (UML)

1.2 Quality within the Development Effort

As was mentioned in the previous section, quality is the mindset that must influence every software engineer. Focusing on quality in all software engineering activities reduces costs and improves time-to-market by minimizing rework. In order to do this, a software engineer must explicitly define what software quality is, have a set of activities that will ensure that every software engineering work product exhibits high quality, do quality control and assurance activities, and use metrics to develop strategies for improving the software product and process.

1.2.1 What is quality?

Quality is the total characteristic of an entity to satisfy stated and implied needs. These characteristics or attributes must be measurable so that they can be compared to known standards.

1.2.2 How do we define quality?

Three perspectives are used in understanding quality, specifically, we look at the quality of the product, quality of the process, quality in the context of the business environment².

Quality of the Product

Quality of the product would mean different things to different people. It is relative to the person analyzing quality. For end-users, the software has quality if it gives what they want, when they want it, all the time. They also judge it based on ease of use and ease in learning to use it. They normally assess and categorized quality based on external characteristics such as number of failures per type of failure. Failures are categorized as minor, major and catastrophic. For the ones developing and maintaining the software, they take a look at the internal characteristics rather than the external. Examples of which includes errors or faults found during requirements analysis, designing, and coding normally done prior to the shipment of the products to the end-users.

As software engineers, we build models based on how the user's external requirements relate to the developer's internal requirements.

Quality of the Process

There are many tasks that affects the quality of the software. Sometimes, when a task fails, the quality of the software suffers. As software engineers, we value the quality of the software development process. Process guidelines suggests that by improving the software development process, we also improve the quality of the resulting product. Common process guidelines are briefly examined below.

- *Capability Maturity Model Integration(CMMI)*. It was formulated by the Software Engineering Institute (SEI). It is a process meta-model that is based on a set of system and software engineering capabilities that must exists within an organization as the organization reaches different level of capability and

² Pfleeger, Shari Lawrence, *Software Engineering Theory and Practice*, International Edition, (Singapore: Prentice-Hall, 1999), p. 10-14

- maturity of its development process.
- *ISO 9000:2000 for Software*. It is a generic standard that applies to any organization that wants to improve the overall quality of the products, systems or services that it provides.
 - *Software Process Improvement and Capability Determination (SPICE)*. It is a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organization in developing an objective evaluation of the efficacy of any defined software process.

Quality in the Context of the Business Environment

In this perspective, quality is viewed in terms of the products and services being provided by the business in which the software is used. Improving the technical quality of the business process adds value to the business, i.e., technical value of the software translates to business value. It is also important to measure the value of the software in terms of business terminologies such as "how many sales orders were processed today?", dollar value on return on investments (ROI) etc. If the software does not add value to the business, why do we need it in the first place?

1.2.3 How do we address the Quality Issues?

We can address quality issues by:

1. *Use Quality Standards*. Quality standards are sets of principles, procedures, methodologies, and guidelines to bring about quality in the process such as CMMI, ISO 9000:2000 for Software and SPICE.
2. *Understand people involved in the development process including end-users and stakeholders*. This fosters an environment of collaboration and effective communication.
3. *Understand the systematic biases in human nature* such as people tend to be risk averse when there is a potential loss, people are unduly optimistic in their plans and forecasts, and people prefer to use intuitive judgment rather than quantitative models.
4. *Commit to quality*. A mindset focus on quality is needed to discover errors and defects so that they can be addressed immediately.
5. *Manage user requirements because it will change over time*. Requirements are the basis defining the characteristics of quality software.

1.3 Software Quality Assurance and Techniques

Software quality assurance is a subset of software engineering that ensures that all deliverables and work products are meet, and they comply with user requirements and standards. It is considered as one of the most important activity that is applied throughout the software development process. Its goal is to detect defects before the software is delivered as a final product to the end-users. It encompasses a quality management approach, effective software engineering technology (methods and tools), formal technical reviews, a multi-tiered testing strategy, control of software documentation and the changes made to it, a procedure to assure compliance with software development standards, and measuring and reporting mechanism.

1.3.1 Software Quality

A software has quality if it is fit for use, i.e., it is working properly. In order for it to work properly, it should conform to explicitly stated functional and performance requirements (user's external characteristics), explicitly documented development standards (quality standards), and implicit characteristics (developer's internal characteristics) that are expected of all professionally developed software.

Three important points should be raised from the definition of software quality.

1. Software Requirements are the foundation from which quality is measured. It is necessary to explicitly specify and prioritize them.
2. Standards define a set of development criteria that guide the manner by which the software is engineered.
3. Implicit characteristics must be identified and documented; they influence the way software is developed such as good maintainability.

1.3.2 Characteristics of a Well-engineered Software

To define a well-engineered software, one takes a look at specific characteristics that the software exhibits. Some of them are enumerated below:

- *Usability*. It is the characteristic of the software that exhibits ease with which the user communicates with the system.
- *Portability*. It is the capability of the software to execute in different platforms and architecture.
- *Reusability*. It is the ability of the software to transfer from one system to another.
- *Maintainability*. It is the ability of the software to evolve and adapt to changes over time. It is characterized by the ease of upgrading and maintaining.
- *Dependability*. It is the characteristic of the software to be reliable, secure and safe.
- *Efficiency*. It is the capability of the software to use resources efficiently.

1.3.3 Software Quality Assurance Activities

Software Quality Assurance is composed of a variety of activities with the aim of building quality software. It involves two groups of people- development team and SQA team. The SQA team has responsibility over the quality assurance planning, overseeing, records keeping, analyzing and reporting defects and rework. Activities involved are the following:

1. *The SQA team prepares the SQA Plan.* They do this during the project planning phase. They identify the:
 - evaluation to be performed;
 - audits and reviews to be performed;
 - standards that are applicable;
 - procedures for error reporting and tracking;
 - documents to be produced; and
 - amount of feedback required.
2. *The SQA team participates in the development of the project's software process description.* The development team selects a software development process and the SQA team checks it if it conform to the organizational policy and quality standards.
3. *The SQA team reviews software engineering activities employed by the development teams to check for compliance with the software development process.* They monitor and track deviations from the software development process. They document it and ensures that corrections have been made.
4. *The SQA team reviews work products to check for compliance with defined standards.* They monitor and track defects or faults found with each work products. They document it and ensure that corrections have been made.
5. *The SQA team ensures that deviations in the software activities and work products are handled based on defined standard operating procedures.*
6. *The SQA team reports deviations and non-compliance to standards to the senior management or stakeholders.*

1.3.4 Formal Technical Reviews

Work products are outputs that are expected as a result of performing tasks in the software process. These results contribute to the development of quality software. Therefore, they should be measurable and checked against requirements and standards. The changes to this work products are significant; they should be monitored and controlled. A technique to check the quality of the work products is the formal technical review. **Formal Technical Reviews (FTR)** are performed at various points of the software development process. It serves to discover errors and defects that can be removed before software is shipped to the end-users. Specifically, its goals are:

1. to uncover errors in function, logic or implementation for any representation of the software;
2. to verify that the software under review meets user requirements;

3. to ensure that the software has been represented according to defined standards;
4. to achieve software that is developed in a uniform manner; and
5. to make projects more manageable.

A general guideline of conducting formal technical reviews is listed below.

- *Review the work product NOT the developer of the work product.* The goal of the review is to discover errors and defect to improve the quality of the software. The tone of the review should be loose but constructive.
- *Plan for the agenda and stick to it.* Reviews should not last more than two hours.
- *Minimize debate and rebuttal.* It is inevitable that issues arise and people may not agree with its impact. Remind everybody that it is not time to resolve these issues rather have them documented, and set another meeting for their resolutions.
- *Point out problem areas but do not try to solve them.* Mention and clarify problem areas. However, it is not time for problem-solving session. It should be done and schedule for another meeting.
- *Write down notes.* It is a good practice to write down notes so that wording and priorities can be assessed by other reviewers. It should aid in clarifying defects and actions to be done.
- *Keep the number of participants to a minimum and insist on preparing for the review.* Writing down comments and remarks by the reviewers is a good technique.
- *Provide a checklist for the work product that is likely to be reviewed.* A checklist provides structure when conducting the review. It also helps the reviewers stay focus.
- *Schedule the reviews as part of the software process and ensure that resources are provided for each reviewer.* Preparation prevents drifts in a meeting. It also helps the reviewers stay focus on the review.
- *De-brief the review.* It checks the effectiveness of the review process.

Two formal technical reviews of work products used in industry are the **Fagan's Inspection Method** and **Walkthroughs**.

Fagan's Inspection Method

It was introduced by Fagan in 1976 at IBM. Originally, it was used to check codes of programs. However, it can be extended to include other work products such as technical documents, model elements, data and code design etc. It is managed by a moderator who as responsibility of overseeing the review. It would required a team of inspectors assigned to play roles that checks the work product against a prepared list of concerns. It is more formal than a walkthrough. It follows certain procedural rules that each member should adhere to. Those rules are listed as follows:

- Inspections are carried out at a number of points in the process of project

- planning and systems development.
- All classes of defects in documentation and work product are inspected not merely logic, specifications or function errors.
 - Inspections are carried out by colleagues at all levels of seniority except the big boss.
 - Inspections are carried out in a prescribed list of activities.
 - Inspection meetings are limited to two hours.
 - Inspections are led by a trained moderator.
 - Inspectors are assigned specific roles to increase effectiveness. Checklists of questionnaires to be asked by the inspectors are used to define the task to stimulate increased defect finding. Materials are inspected at a particular rate which has been found to give maximum error-finding ability.
 - Statistics on types of errors are key, and used for reports which are analyzed in a manner similar to financial analysis.

Conducting inspections require a lot of activities. They are categorized as follows:

- *Planning.* A moderator is tasked to prepare a plan for the inspection. He decides who will be the inspectors, the roles that they have to play, when and where they have to play the roles, and distributes the necessary documentation accordingly.
- *Giving of the overview.* A 30-minute presentation of the project for the inspectors are given. It can be omitted if everybody is familiar with the overall project.
- *Preparing.* Each inspector is given 1 to 2 hours alone to inspect the work product. He will perform the role that was assigned to him based on the documentation provided by the moderator. He will try to discover defects in the work product. He is discouraged to fix the defects or criticize the developer of the work product.
- *Holding the meeting.* The participants of the meeting are the inspectors, moderator and the developer of the work product. The developer of the work product is present to explain the work product, and answer questions that inspectors ask. No discussion about whether the defect is real or not is allowed. A defect list is produced by the moderator.
- *Reworking of the work product.* The defect list is assigned to a person for repair. Normally, this is the developer of the work product.
- *Following up the rework.* The moderator ensures that the defects on the work products are addressed and reworked. These are later on inspected by other inspections.
- *Holding a casual analysis meeting.* This is optionally held where inspectors are given a chance to express their personal view on errors and improvements. Emphasis is given to the way the inspection was done.

Walkthrough

A walkthrough is less formal than the inspection. Here, the work product and corresponding documentation are given to a review team, normally around 3 people, where comments of their correctness are elicited. Unlike the inspection where one has a moderator, the developer of the work product, moderates the walkthrough. A scribe is also present to produce an action list. An action list is a list of actions that must be done in order to improve the quality of the work product which includes the rework for the defects, resolution of issues etc.

Some guidelines must be followed in order to have a successful walkthrough. They are listed below:

- No manager should be present.
- Emphasize that the walkthrough is for error detection only and not error correction.
- Keep vested interest group apart.
- No counting or sandbagging.
- Criticize the product; not the person.
- Always document the action list.

Conducting a walkthrough, similar with inspection, would require many activities. They are categorized as follows:

- *Pre-walkthrough Activities*
 - The developer of the work product schedules the walkthrough preferably, a day or two in advance.
 - He distributes the necessary materials of the work product to the reviewers.
 - He specifically asks each reviewer to bring to the walkthrough two positive comments and one negative comment about the work product.
- *Walkthrough Proper*
 - The developer of the work product gives a brief presentation of the work product. This may be omitted if the reviewers are familiar with the work product or project.
 - He solicit comments from the reviewers. Sometimes, issues arise and presented but they should not find resolutions during the walkthrough. Issues are listed down in the action list.
 - An action list is produced at the end of the walkthrough.
- *Post-walkthrough Activities*
 - The developer of the work product receives the action list.
 - He is asked to submit a status report on the action taken to resolve the errors or discrepancies listed in the action list.
 - Possibly, another walkthrough may be scheduled.

1.4 The Software Process

The **software process** provides a strategy that a software development team employs in order to build quality software. It is chosen based on the nature of the project and application, methods and tools to be used, and the management and work products that are required. Pressman provides a graphical representation of the software process. According to him, it provides the framework from which a comprehensive plan for software development can be established. It consists of **framework activities**, **task sets** and **umbrella activities**.³

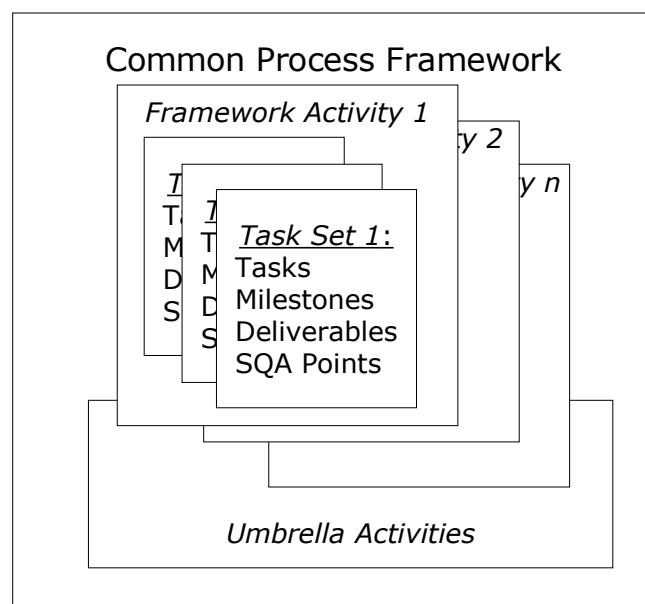


Figure 1.2 Pressman's Software Process

Framework of Activities

These are activities that are performed by the people involved in the development process applicable to any software project regardless of project size, composition of the development team, and complexity of the problem. They are also known as phases of the software development process.

Task Sets

Each of the activities in the process framework defines a set of tasks. These tasks would have milestones, deliverables or work products and software quality assurance (SQA) points. They are modified and adjusted to the specific characteristic of the software project, and the requirements of the software.

Umbrella Activities

These are activities that supports the framework of activities as the software development project progresses such as software project management, change

³ Pressman, Software Engineering A Practitioner's Approach, p. 54-55

management, requirements management, risk management, formal technical reviews etc.

1.4.1 Types of Software Process Models

There are many types of software process models that suggest how to build software. Common process models are discussed within this section.

Linear Sequential Model

The **Linear Sequential Model** is also known as the **waterfall model** or the **classic life cycle**. This is the first model ever formalized, and other process models are based on this approach to development. It suggests a systematic and sequential approach to the development of the software. It begins by analyzing the system, progressing to the analysis of the software, design, coding, testing and maintenance. It insists that a phase can not begin unless the previous phase is finished. Figure 1.3 shows this type of software process model.

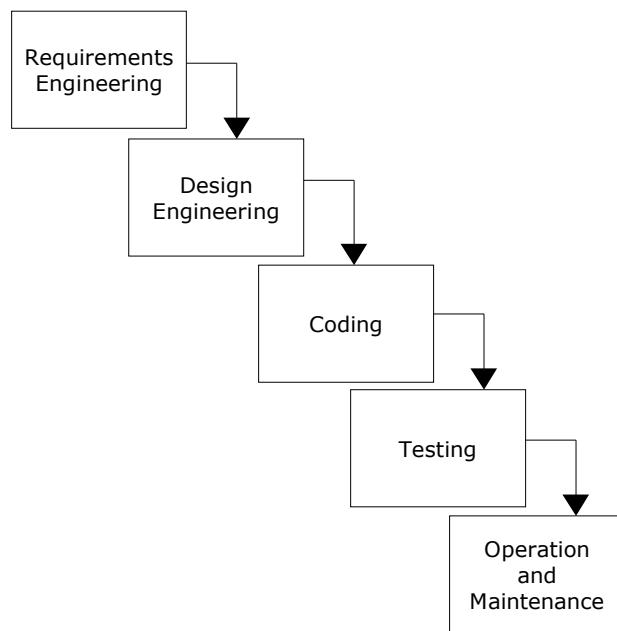


Figure 1.3 Linear Sequential Model

The advantages of this model are:

- It is the first process model ever formulated.
- It provides a basis for other software process models.

The disadvantages of this model are:

- Real software projects rarely follow a strict sequential flow. In fact, it is very difficult to decide when one phase ends and the other begins.

- End-user involvement only occurs at the beginning (requirements engineering) and at the end (operations and maintenance). It does not address the fact the requirements may change during the software development project.
- End-users sometimes have difficulty stating all of their requirements. Thus, it delays the development of the software.

Prototyping Model

To aid in the understanding of end-user requirements, prototypes are built. **Prototypes** are partially developed software that enable end-users and developers examine aspects of the proposed system and decide if it is included in the final software product. This approach is best suited for the following situations:

- A customer defines a set of general objectives for the software but does not identify detailed input, processing, or output requirements.
- The developer may be unsure of the efficiency of an algorithm, the adaptability of a technology, or the form that human-computer interaction should take.

Figure 1.4 shows this process model.

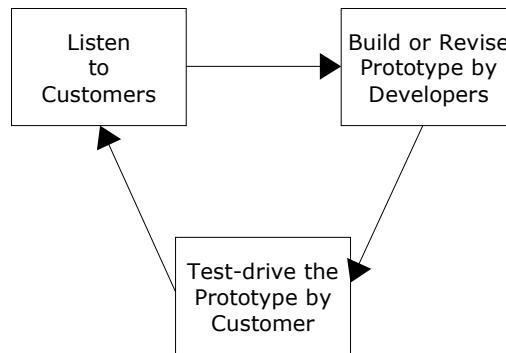


Figure 1.4 Prototyping Model

The advantage of this process model is:

- The end-users have an active part in defining the human-computer interaction requirements of the system. They get the actual "feel" of the software.

The disadvantages of this process model are:

- Customers may mistakenly accept the prototype as a working version of the software. Software quality is compromised because other software requirements are not considered such as maintainability.
- Developers tend to make implementation compromises in order to have a working prototype without thinking of future expansion and maintenance.

Rapid Application Development (RAD) Model

This process is a linear sequential software development process that emphasizes an extremely short development cycle. It is achieved through a modular-based construction approach. It is best used for software projects where requirements are well-understood, project scope is properly constrained, and big budget with resources are available. Everybody is expected to be committed to a rapid approach to development.

In this process model, the software project is defined based on functional decomposition of the software. Functional partitions are assigned to different teams, and are developed in parallel. Figure 1.5 shows this process model.

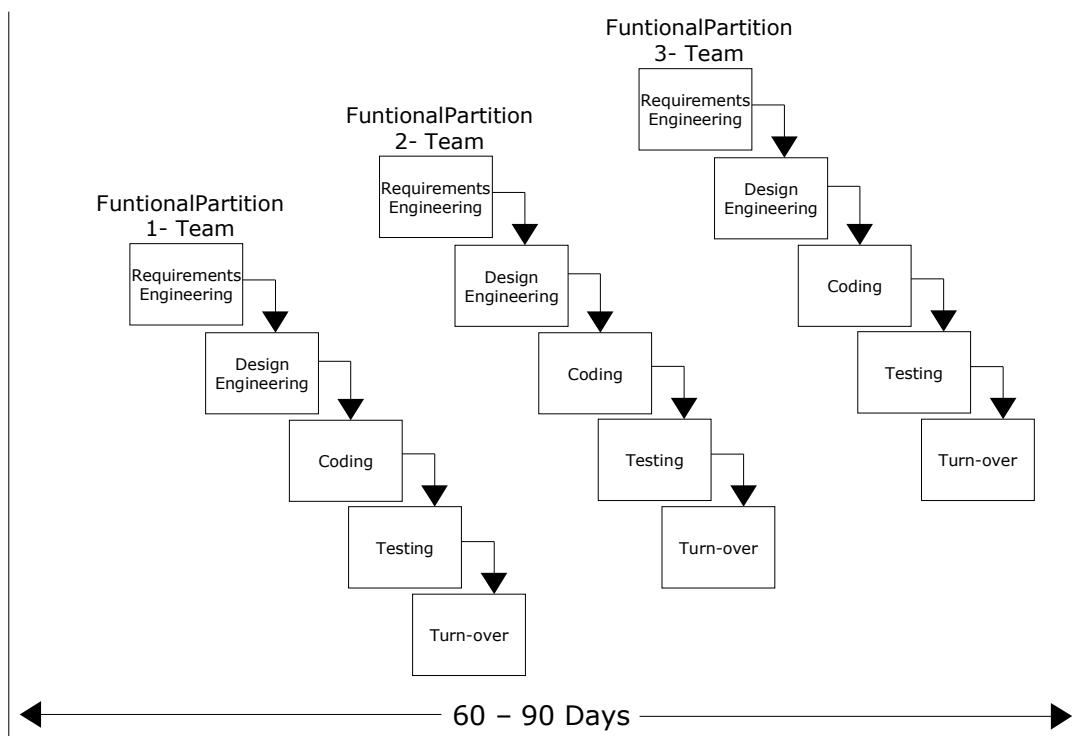


Figure 1.5 Rapid Application Development

The advantage of this model is:

- A fully functional system is created in a short span of time.

The disadvantages of this model are:

- For large but scalable projects, this process requires a sufficient number of developers to have the right number of development teams.
- Developers and customers must be committed to the rapid-fire of activities necessary to develop the software in a short amount of time.
- It is not a good process model for systems that cannot be modularized.
- It is not a good process model for systems that require high performance.
- It is not a good process model for systems that make use of new technology or

high degree of interoperability with existing computer programs such as legacy systems.

Evolutionary Process Models

This process model recognizes that software evolves over a period of time. It enables the development of an increasingly more complicated version of the software. The approach is iterative in nature. Specific evolutionary process models are ***Incremental Model***, ***Spiral Model***, and ***Component-based Assembly Model***.

Incremental Model

This process model combines the elements of a linear sequential model with the iterative philosophy of prototyping. Linear sequences are defined where each sequence produces an ***increment*** of the software. Unlike prototyping, the increment is an operational product. Figure 1.6 shows this process model.

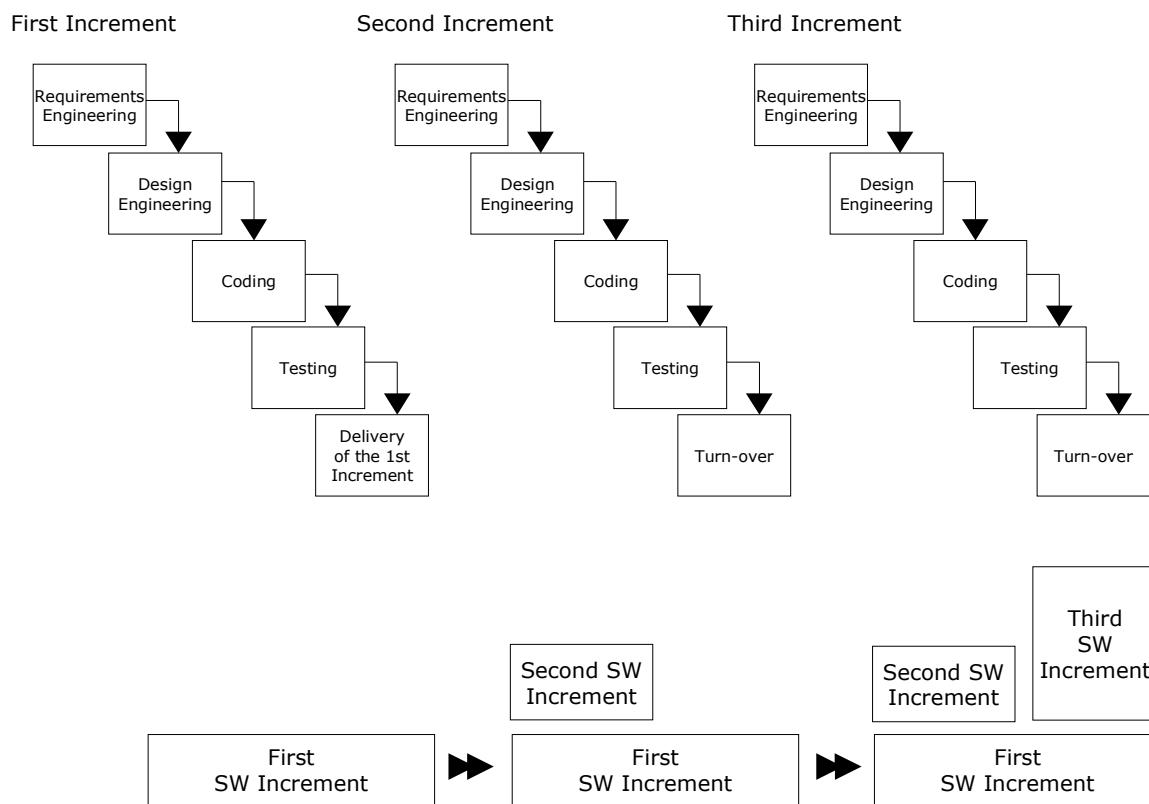


Figure 1.6 Incremental Process Model

Spiral Model

It was originally proposed by Boehm. It is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of linear sequential model. It provides potential rapid development of incremental versions of the software. An important feature of this model is that it has risk analysis as one of its framework of activities. Therefore, it requires risk assessment expertise. Figure 1.7 shows an example of a spiral model.

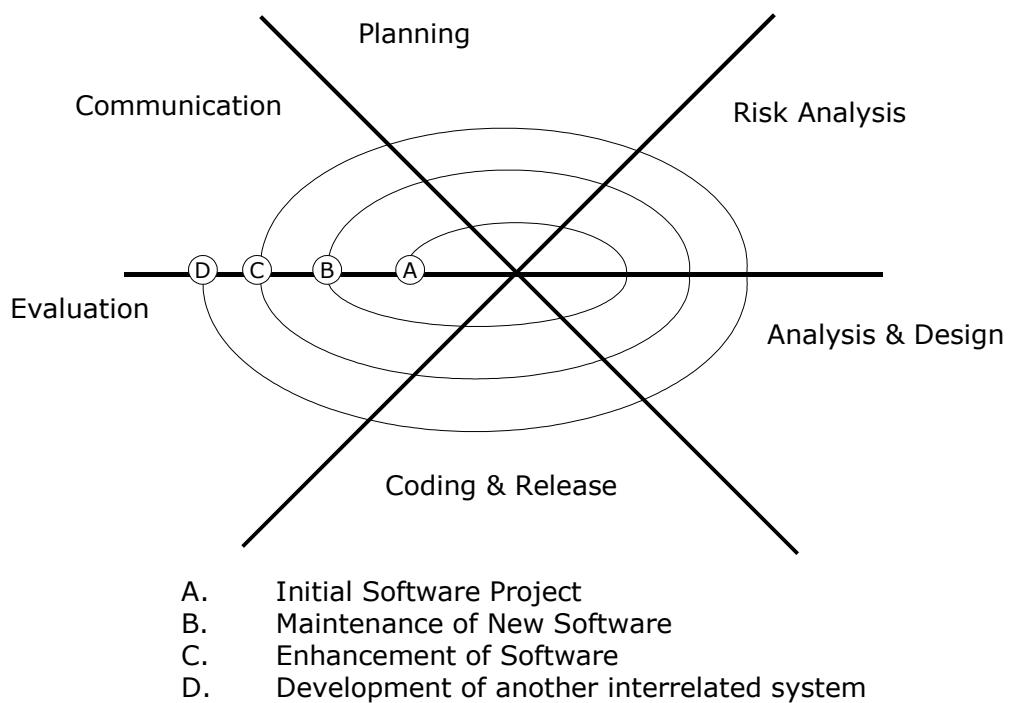


Figure 1.7 Spiral Model

Component-based Assembly Model

It is similar to Spiral Process Model. However, it makes use of object technologies where the emphasis of the development is on the creation of classes which encapsulates both data and the methods used to manipulate the data. Reusability is one of the quality characteristics that are always checked during the development of the software. Figure 1.8 shows the Component-based Assembly Model.

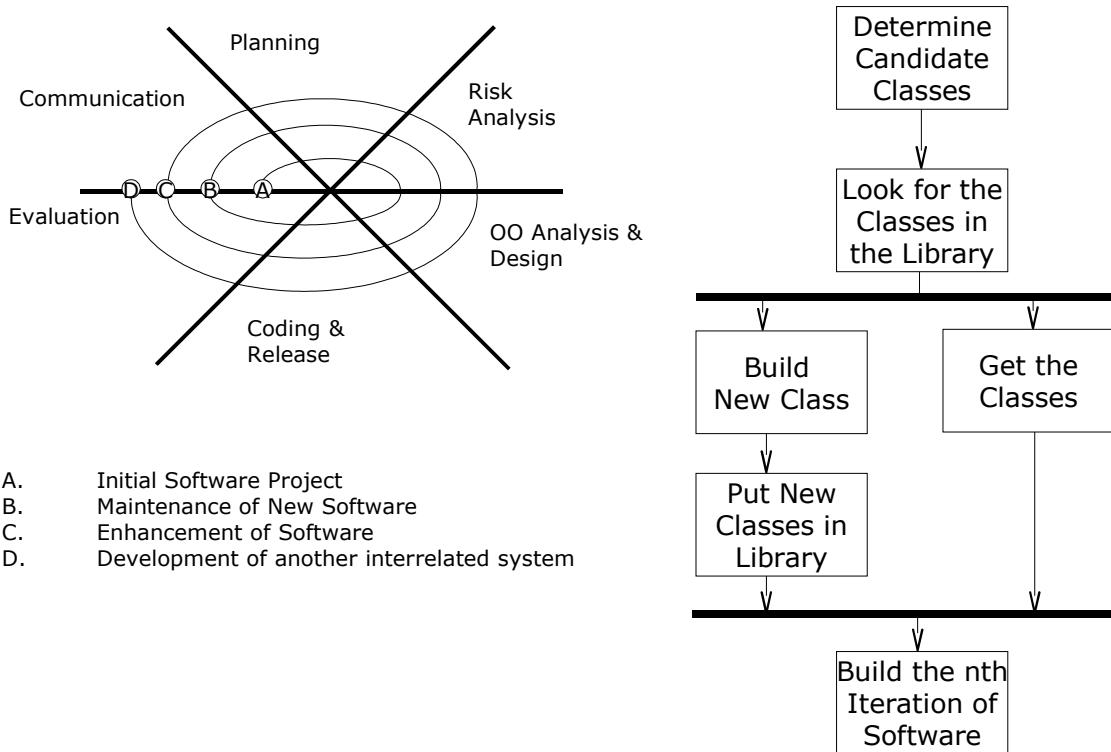


Figure 1.8 Component-based Assembly Model

Concurrent Development Model

The Concurrent Development Model is also known as ***concurrent engineering***. It makes use of state charts to represents the concurrent relationship among tasks associated within a framework of activities. It is represented schematically by a series of major technical tasks, and associated states. The user's need, management decisions and review results drive the over-all progression of the development. Figure 1.9 shows the concurrent development model.

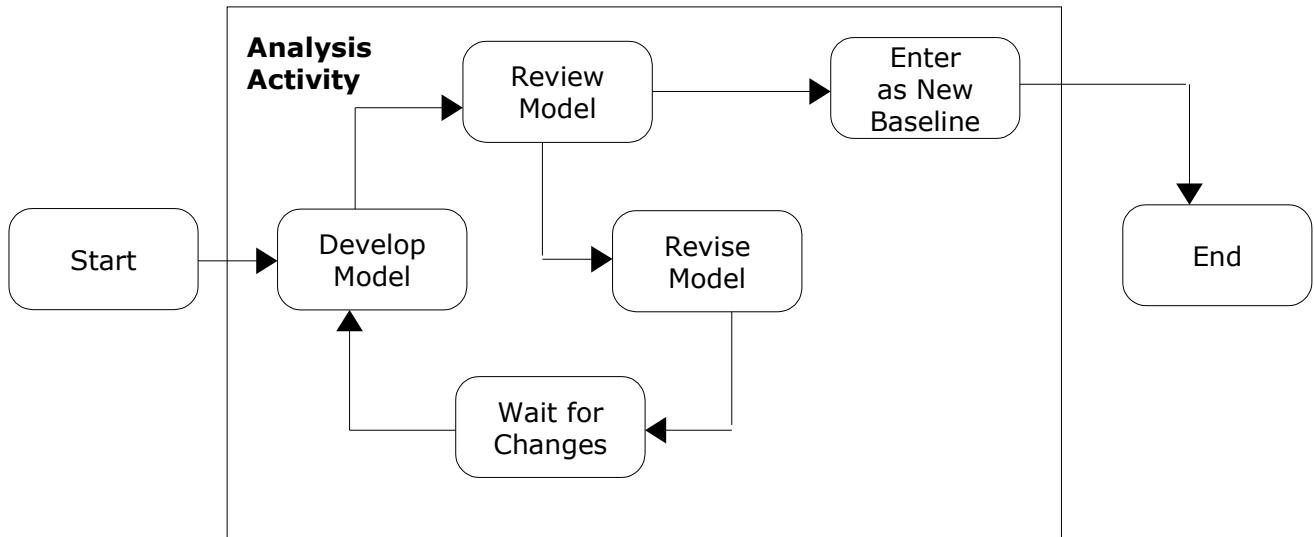


Figure 1.9 Concurrent Development Model

Formal Methods

The Formal Methods is a software engineering approach which encompasses a set of activities that lead to mathematical specification of the software. It provides a mechanism for removing many of the problems that are difficult to overcome using other software engineering paradigm. It serves as a means to verify, discover and correct errors that might otherwise be undetected.

1.4.2 Factors that Affect the Choice of Process Model

- Type of the Project
- Methods and Tools to be Used
- Requirements of the Stakeholders
- Common Sense and Judgment

1.5 Understanding Systems

The software project that needs to be developed revolves around systems. **Systems** consists of a group of entities or components, interacting together to form specific interrelationships, organized by means of structure, and working together to achieve a common goal. Understanding systems provides a context for any project through the definition of the boundaries of the projects. It asks the question, "What is included in the project? What is not?" In defining the system boundaries, a software engineer discovers the following:

- entities or group of entities that are related and organized in some way within the system, either they provide input, do activities or receive output;
- activities or actions that must be performed by the entities or group of entities in order to achieve the purpose of the system;
- a list of inputs; and
- a list of outputs.

As an example, Figure 1.10 shows the system boundaries of the case study. It shows elements of this system through the use of the context diagram.

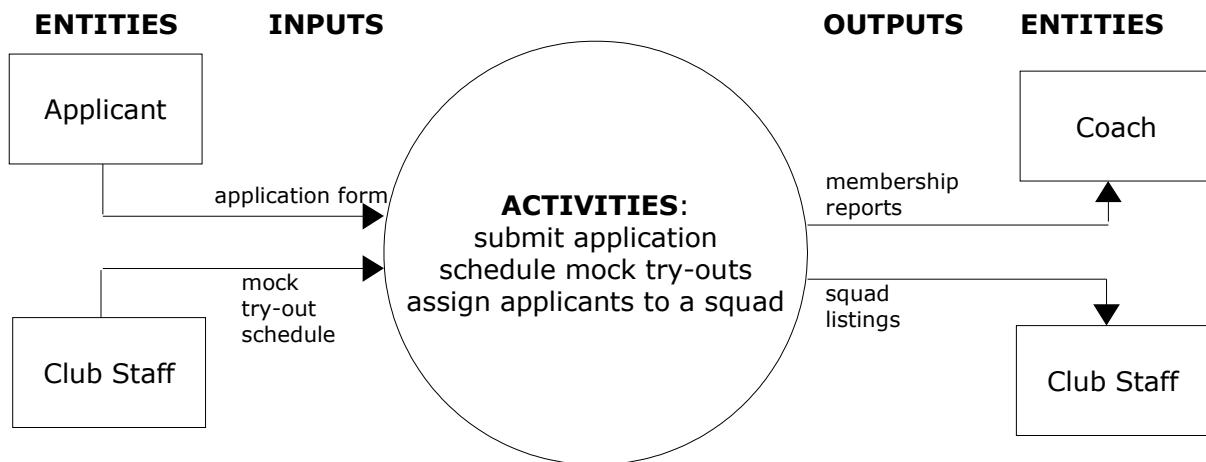


Figure 1.10 Club Membership Application System Boundaries

Entities that are involved in this system are the applicant, club staff and coach. They are represented as rectangular boxes. They are related with one another by performing certain activities within this system. The major activities that are performed are the submission of the application forms, scheduling of mock try-outs and the assignment of the applicant to a squad. They are represented by a circle in the middle that defines the functionality of maintaining club membership information. To perform these actions, a list of inputs are necessary, specifically, application forms and the schedule of the mock try-outs. They are represented by an arrow with the name of the data being passed. The arrow head indicates the flow of the data. The results that are expected from this system are the membership reports and importantly, the squad listings. Again, they are represented by an arrow with the name of the data being passed. The arrow head

indicates the flow of the data. The goal of this system is to handle club membership application.

General Principles of Systems

Some general principles of systems are discussed below. This would help the software engineer study the system where the software project revolves.

- *The more specialized a system, the less it is able to adapt to different circumstances.* Changes would have a great impact on the development of such systems. One should be careful that there is no dramatic changes in the environment or requirements when the software is being developed. Stakeholders and developers should be aware of the risks and costs of the changes during the development of the software.
- *The larger the system is, the more resources must be devoted to its everyday maintenance.* As an example, the cost of maintaining a mainframe is very expensive compared to maintaining several personal computers.
- *Systems are always part of larger systems, and they can always be partitioned into smaller systems.* This is the most important principle that a software engineer must understand. Because systems are composed of smaller subsystem and vice versa, software systems can be developed in a modular way. It is important to determine the boundaries of the systems and their interactions so that the impact of their development is minimal and can be managed and controlled.

Components of Automated Systems

There are two types of systems, namely, man-made systems and automated systems. Man-made systems are also considered manual systems. They are not perfect. They will always have areas for correctness and improvements. These areas for correctness and improvements can be addressed by automated systems.

Automated systems are examples of systems. It consists of components that support the operation of a domain-specific system. In general, it consists of the following:

1. *Computer Hardware.* This component is the physical device.
2. *Computer Software.* This component is the program that executes within the machine.
3. *People.* This component is responsible for the use of the computer hardware and software. They provide the data as input, and they interpret the output (information) for day-to-day decisions.
4. *Procedures.* This component is the policies and procedures that govern the operation of the automated system.
5. *Data and Information.* This component provides the input (data) and output (information).
6. *Connectivity.* This component allows the connection of one computer system with another computer system. It is also known as the network component.

Figure 1.11 shows the relationship of the first five components.

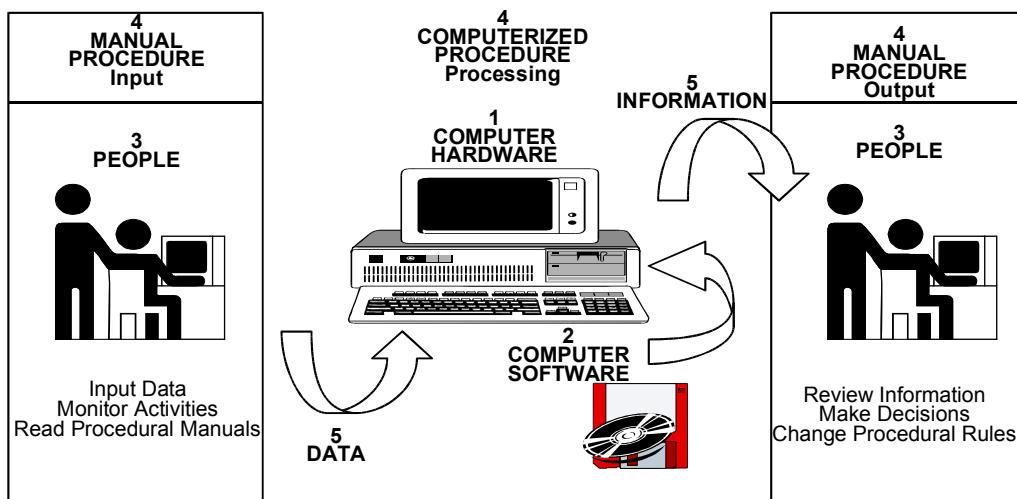


Figure 1.11 Components of An Automated System

Let's take a look of an application domain-specific illustration of an automated system. Figure 1.12 shows the automated system of the club membership application processing.

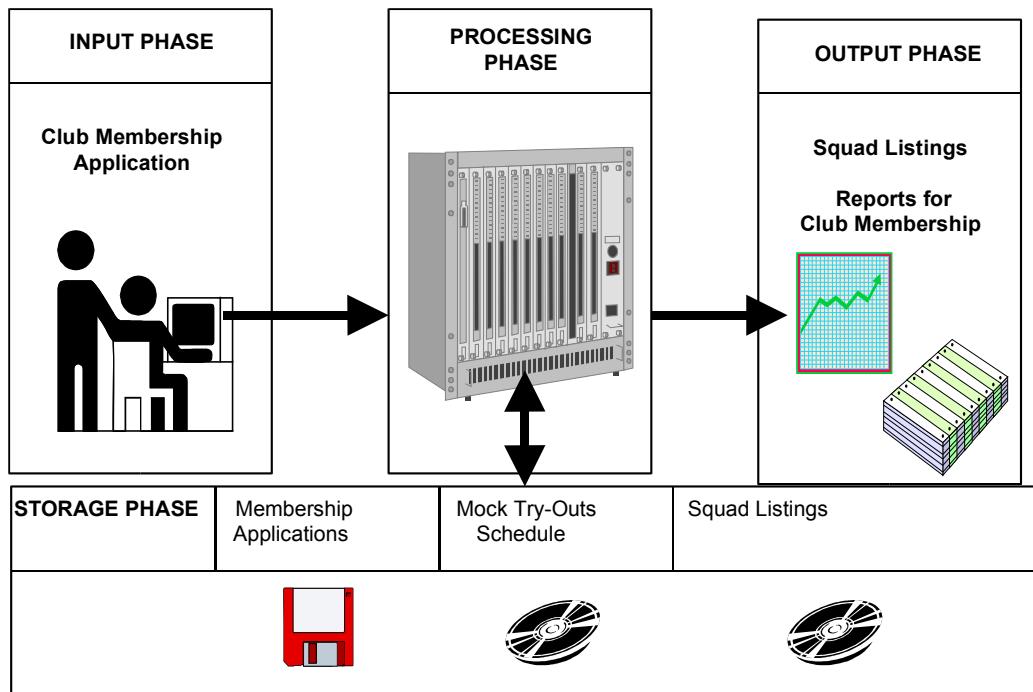


Figure 1.12 Club Membership Application Computer System

J.E.D.I

The information from the Club Membership Application is entered into the system. The applicant is scheduled for a mock-try out. This schedule is retrieved from storage. Once an applicant is assigned to a squad, this information is entered into the system so that the Squad Listing Report is produced.

1.6 Understanding People in the Development Effort

To help in the fostering of a quality mindset in the development of the software, one should understand the people involved in the software development process, particularly, their interest regarding the system and the software that needs to be developed. In this section, there are two major groups that are involved in the software development effort, specifically, end-users and development team.

1.6.1 End-users

End-users are the people who will be using the end-product. Much of the requirements will be coming from this group. They can be grouped into two according to their involvement within the system organization and development, namely, those who are directly involved and those who are indirectly involved.

Those who are directly involved

Table 1 shows the categorization of the end-users according to the job functions that they perform within the system.

<i>Operational Job</i>	<i>Supervisor Job</i>	<i>Executive Job</i>
They perform the operational functions of the system.	They perform supervisory actions on daily operations of the system. They are mostly measured and motivated by performance against budget.	They usually provide the initiative and serve as the funding authority of the systems development project.
They are more likely to be concerned with the human interface component of the system: <ul style="list-style-type: none"> • What type of keyboard will be using? • What kind of on-line display screen will the system have? • Will there be a lot of glare and will the characters be easy to read? 	They are more likely to be concerned with the operational efficiency of the functions that needs to be performed such as more outputs in less time.	They are less likely to be concerned with the day-to-day operations. They are more concerned with strategic issues and long-term profit-and-loss.
They have the local view of the system.	They also tend to have the same local and physical view of the system similar with the operational users but will have performance concerns.	They are most likely interested in the global view of the system.
They tend to think of the system in physical terms.	They are the users who have more contact with the software engineers.	They are generally able to work with abstract models of the system rather than the physical terms. They are more interested in results.

Table 1: Job Category

General Guidelines with End-Users

- The higher the level of the manager, the less he or she is likely to care about computer technology. It would be best to ask him or her over-all results and performance the system can provide. They are good candidates for interview regarding the report layouts and code design.
- The goals and priorities of management may be in conflict with those of the supervisory and operational users. This can be seen based on their different levels of concerns. As software engineer, try to discover areas of commonality. More on this on Chapter 3- Requirements Engineering.
- Management may not provide resources, funding or time that the users feel is necessary to build an effective system. Resource and financial constraints will occur. It is important to prioritize requirements. More on this on Chapter 3- Requirements Engineering.

Those who are indirectly involved

Mostly, these group includes the auditors, standard bearers, and quality assurance group. The general objective of this group is to ensure that the system is developed in accordance with various standard set such as:

- Accounting standards developed by the organization's accounting operations or firm.
- Standards developed by other departments within the organization or by the customer or user who will inherit the system
- Various standards imposed by the government regulatory agencies.

Some possible problem that may be encountered with this group. As software engineers, keep an eye on them and address them accordingly.

- They don't get involved in the project until the very end, particularly, the quality assurance group. It is important that they be involved in every activity that would require their expertise and opinion.
- They provide the necessary notation and format of documentation. They may be needed in the definition of the presentation and documentation of the system.
- They are more interested in substance rather than form.

1.6.2 Development Team

The development team is responsible in building the software that will support a domain-specific system. It may consists of the following: systems analyst, systems designer, programmer and testers.

System Analyst

His responsibility is understanding the system. Within this system, he identifies customer wants, and documents and prioritizes requirements. This involves breaking down the system to determine specific requirements which will be the basis for the design of the software.

System Designer

His job is to transform a technology free architectural design that will provide the framework within which the programmers can work. Usually, the system analyst and designer are the same person but it must be emphasized that the functions require different focus and skill.

Programmers

Based on the system design, the programmers write the codes of the software using a particular programming language.

Testers

For each work product, it should be reviewed for faults and errors. This supports the quality culture needed to developed quality software. It ensures that work products meet requirements and standards defined.

1.7 Documentation in the Development Effort

1.7.1 What is documentation?

It is a set of documents or informational products to describe a computer system. Each document is designed to perform a particular function such as:

- REFERENCE, examples are technical or functional specifications
- INSTRUCTIONAL, examples are tutorials, demonstrations, prototypes etc.
- MOTIVATIONAL, examples are brochures, demonstrations, prototypes.

There are several types of documentation and informational work products. Some of them are listed below:

- System Features and Functions
- User and Management Summaries
- Users Manual
- Systems Administration Manuals
- Video
- Multimedia
- Tutorials
- Demonstrations
- Reference Guide
- Quick Reference Guide
- Technical References
- System Maintenance Files
- System Test Models
- Conversion Procedures
- Operations/Operators Manual
- On-line help
- Wall Charts
- Keyboard Layouts or Templates
- Newsletters

Good documents cannot improve messy systems. However, they can help in other ways. The following table shows how documentation support the software development process.

If the user manuals are developed during....	Then, the manuals can...
Product definition	<ul style="list-style-type: none"> Clarify procedures and policies Identify unfriendly elements Increase changes of user satisfaction
Design and Coding	<ul style="list-style-type: none"> Clarify bugs and errors Identify causes of unreliability Force designer to make early decisions
Distribution and Use	<ul style="list-style-type: none"> Help users adapt to the product Warn against bugs in the system Disclaim liability

Table 2 Documentation Significance

There are two main purpose of documentation. Specifically, they:

- provide a reasonably permanent statement of a system's structure or behavior through reference manuals, user guides and systems architecture documents.
- serve as transitory documents that are part of the infrastructure involved in running real projects such as scenarios, internal design documentation, meeting reports, bugs etc.

1.7.2 Criteria for Measuring Usability of Documents

A useful document furthers the understanding of the system's desired and actual behavior and structure. It serves to communicate the system's architectural versions. It provides a description of details that cannot be directly inferred from the software itself or from executable work products. Some criteria for measuring usability of documents are listed below:

- Availability.** Users should know that the documents exists. It must be present when and where needed.
- Suitability.** It should be aligned to users tasks and interests. It should be accurate and complete. Related documents must be located in one manual or book.
- Accessibility.** It should fit in an ordinary 8.5in x 11in paper for ease of handling, storage, and retrieval. It should be easy to find the information that users want. Each item of documentation should have a unique name for referencing and cross-referencing, a purpose or objective, and target audience (who will be using the document). Referrals to other manuals and books should be avoided.
- Readability.** It should be understandable without further explanation. It should not have any abbreviations. If you must use one, provide a legend. It should be written in a fluent and easy-to-read style and format.

1.7.3 Important of Documents and Manuals

Documents and manuals are important because:

- They save cost. With good manuals, one needs less personnel to train the users, support on-going operations, and maintain the system.
- They serve as sales and marketing tools. Good manuals differentiate their products- a slick manual means a slick product- especially, for off-the-shelf software products.
- They serve as tangible deliverables. Management and users know little about computer jargons, programs and procedures, but they can hold and see a user manual.
- They serve as contractual obligations.
- They serve as security blankets. In case people leave, manuals and technical documents serve as written backup.
- They serve as testing and implementation aids. It is important to include the following items as part of the user's manual- system test scripts and models, clerical and automated procedures, hands-on training for new personnel and design aid.
- They are used to compare the old and new systems.

1.8 Exercises

1.8.1 Specifying Boundaries

1. Model the system boundary of the Coach Information System. Use Figure 1.10 as your guide.
2. Model the system boundary of the Squad and Team Maintenance System. Use Figure 1.10 as your guide.

1.8.2 Practicing the Walkthrough

1. Review the system boundary model of the *Coach Information System* by performing a walkthrough. Prepare an action list.
2. Review the system boundary model of the *Squad & Team Maintenance System* by performing a walkthrough. Prepare an action list.

1.9 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in Chapter 1. Particularly, they are:

1. Defining the System Boundaries
2. Creating the System Boundary Model
3. Performing Walkthrough

WORK PRODUCTS:

1. System Boundary Model
2. Action List

2 Object-oriented Software Engineering

Object-oriented Software Engineering is the use of object technologies in building software. Object technology is a set of principles that guide the construction of the software using object-oriented approach. It encompasses all framework of activities including analysis, design and testing, and the choice of methodologies, programming languages, tools, databases and applications to engineer a software.

In this chapter, we lay the foundation for understanding object-orientation by presenting an explanation of its fundamental concepts such as objects and classes, abstraction, encapsulation, modularity and hierarchy. We also present a general object-oriented process model that follows a component-based assembly. We also introduce object-oriented analysis and design activities, list down some methodologies, and expected work products. Finally, we will discuss the Unified Modeling Language (UML) and the modeling activity.

2.1 Review of Object-oriented Concepts

At the very heart of object-orientation, we have the objects. **Objects** are representations of entities which can be physical (such as club membership application form and athletes), conceptual (squad assignment) or software (linked list). It allows software engineers to represent real world objects in software design. More technically, it is defined as something that represents a real world object which has a well-defined boundary and identity that encapsulates state and behavior.

Attributes and relationships of an object define its **state**. It is one of the possible conditions by which an object exists, and it normally changes overtime. In software, the values stored within the attributes and the links of the object with other objects define this state. Operations, methods and state machines, on the other hand, define its **behavior**. It determines how an object acts and reacts to message requests from other objects. It is important that each object should be uniquely identified within the system even if they have the same values in the attributes and behavior. Figure 2.1 depicts examples of objects with their state and behavior that may exists in the case study, Ang Bulilit Liga.

Three objects are present- 2 athletes and 1 coach. In this picture, they are illustrated as circles where attributes are found in the inner circle surrounded by methods. Objects are uniquely identified by their Ids such as in the case of the two athletes, *Joel* and *Arjay*. Notice that attributes are enclosed by methods. This suggests that only the object can change the values of their own attributes. The changes in the values of the attributes can be triggered by a request, called a **message**, by another object. In the picture, the coach (*JP*) assigns an athlete (*Joel*) to a squad by executing his method *assignToSquad()*. This method sends a request message, *updateSquad("Training")*, to the athlete (*Joel*) to update his *squad* attribute.

A **class** is a description of a set of objects that share the same attributes, relationships, methods, operations and semantics. It is an abstraction that focuses on the relevant characteristics of all objects while ignoring other characteristics. Objects are instances of classes. In our example, *Joel* and *Arjay* are instances of the class *athletes*, while *JP* is an instance of the class *coach*.

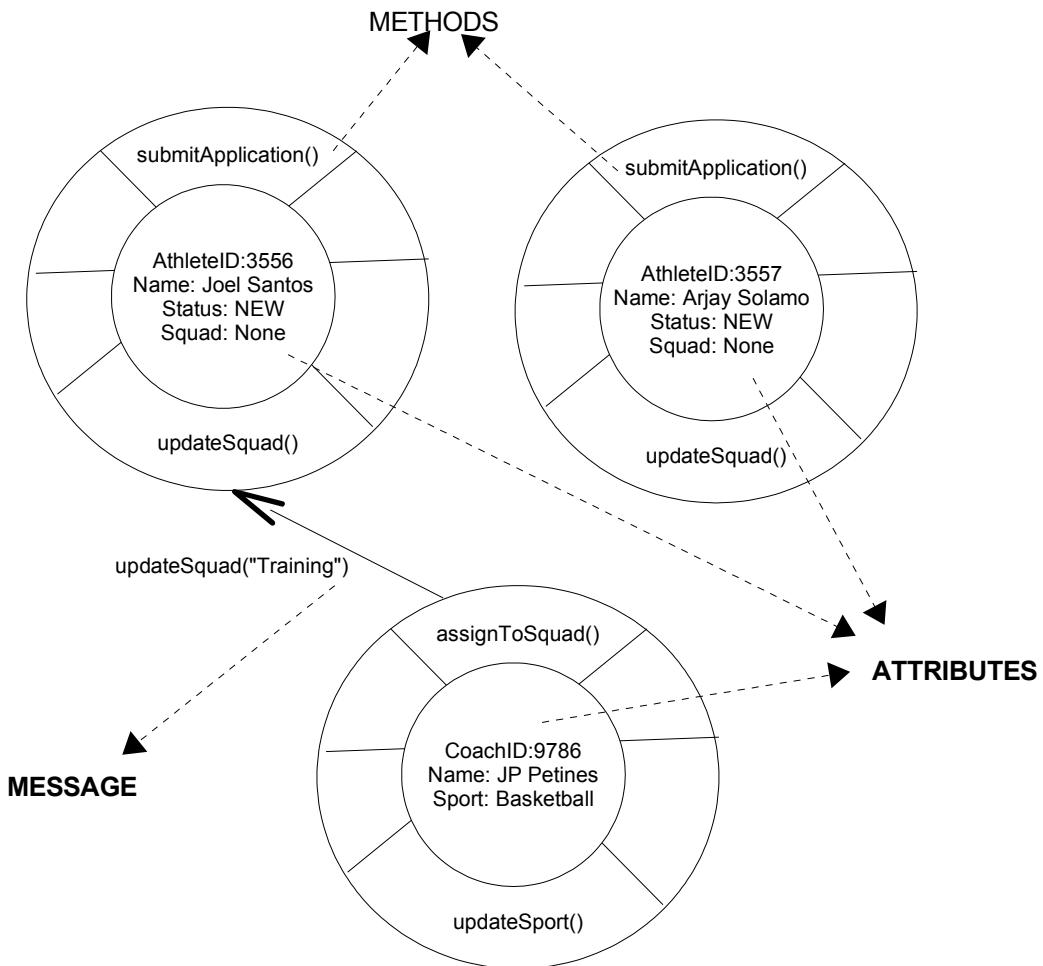


Figure 2.1 Club Membership Application Object Examples

Four basic principles that are generally associated with object-orientation are abstraction, encapsulation, modularity and hierarchy. This concepts are interrelated and supports one another.

2.1.1 Abstraction

Abstraction is defined as the essential characteristics of an entity that distinguishes it from all other kinds of entities¹. It is a kind of representation that includes only the things that are important or interesting from a particular point of view. It is domain and perspective dependent, i.e., what is important in one context may not necessarily be important in another. It allows us to manage the complexity of the system by concentrating only on those characteristics that are essential or important in the system, and ignoring or de-emphasizing the characteristics that are not. Objects are represented by those features that are deemed relevant to the current purpose, and hides those features that are not.

Examples of abstraction within the case study are:

¹ Object-oriented Analysis and Design using the UML, Student's Manual, (Cupertino, CA: Rational software Corporation, 2000), p. 2-15

- An applicant submits a club membership application to the club staff.
- A club staff schedules an applicant for a mock try-outs.
- A coach assigns an athlete to a squad.
- A squad can be a training squad or a competing squad.
- Teams are formed from a squad.

2.1.2 Encapsulation

Encapsulation is also known as **information hiding**. It localizes features of an entity into a single blackbox abstraction, and hides the implementation of these features behind an interface. It allows other objects to interact with one another in such a way that they don't need to know how the implementation fulfills the interface. This is achieved through the object's **message interface**. This interface is a set of pre-defined operations used so that other objects can communicate with it. It ensures that data within the attributes of the object are accessible through an object's operation. No other object can directly access these attributes, and change their values.

Consider the interaction among objects in Figure 2.1 where a coach (*JP*) assigns an athlete (*Joel*) to a squad. *updateSquad()* is the message interface that changes the value of the *squad* attribute of the athlete (*Joel*). Notice, that the change will only occur when the coach (*JP*) executes *assignToSquad()* which triggers a request to *updateSquad ("Training")* of the *squad* attribute (*Joel's*). The coach (*JP*) does not need to know how the athlete (*Joel*) updates the *squad* but is assured that the method is executed.

Encapsulation reduces the "ripple effect of changes" on codes, where a change in one object's implementation will cause another change in another object's implementation and so on. With encapsulation, one can change the implementation without changing the other object's implementation as long as the interface is unchanged. Thus, encapsulation offers two kinds of protection to objects: protection against corruption of internal state and protection against code change when another object's implementation changes.

2.1.3 Modularity

Modularity is the physical and logical decomposition of large and complex things into small and manageable components. These components can be independently developed as long as their interactions are well-understood. The concepts of packages, subsystems and components in object-orientation support modularity. They will be explained further in the succeeding sections of this chapter.

Modularity, like abstraction, is another way of managing complexity. Because it breaks something that is large and complex into smaller manageable components, it makes it easier for a software engineer to manage and develop the software by managing and developing these smaller components. Then, iteratively integrate them.

For example, "*Ang Bulilit Liga*" case study can be divided into smaller subsystems as shown in Figure 2.2.

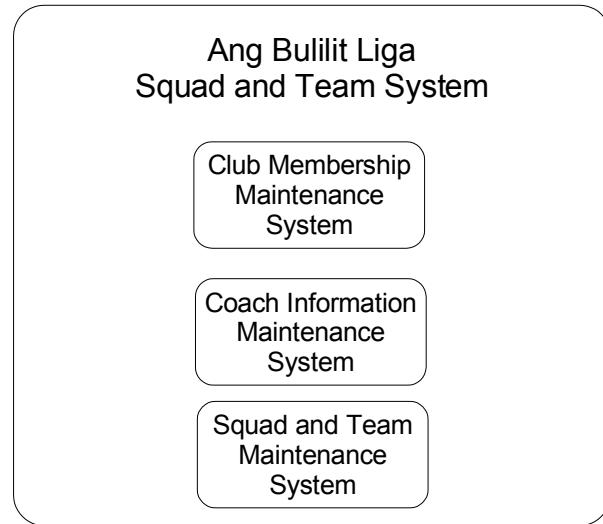


Figure 2.2 Ang Bulilit Liga Subsystems

2.1.4 Hierarchy

Hierarchy can be any ranking of ordering of abstraction into a tree-like structure. There are different kinds of hierarchy, and they are listed below.

- Aggregation
- Class
- Containment
- Inheritance
- Partition
- Specialization
- Type

Figure 2.3 shows an example of the Squad Hierarchy.

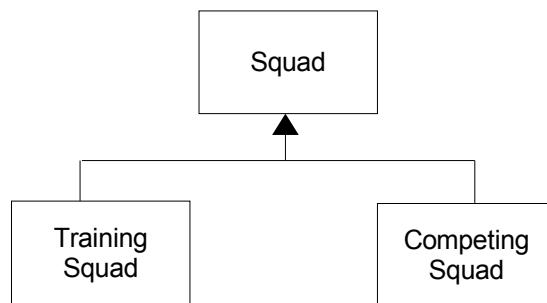


Figure 2.3 Squad Hierarchy

Generalization is a form of association wherein one class shares the structure and/or behavior of one or more classes. It defines a hierarchy of abstractions in which a subclass inherits from one or more superclass. It **is an is a kind** of relationship. In Figure 2.3, the *Squad* class is the superclass of the *Training Squad* and *Competing Squad* classes.

Inheritance is a mechanism by which more specific elements incorporate the structure and behavior of more general elements. A subclass inherits attributes, operations and relationships from a superclass. Figure 2.3 is elaborated in Figure 2.4, all attributes and methods of the *Squad* superclass are inherited by the subclasses *Training Squad* and *Competing Squad*.

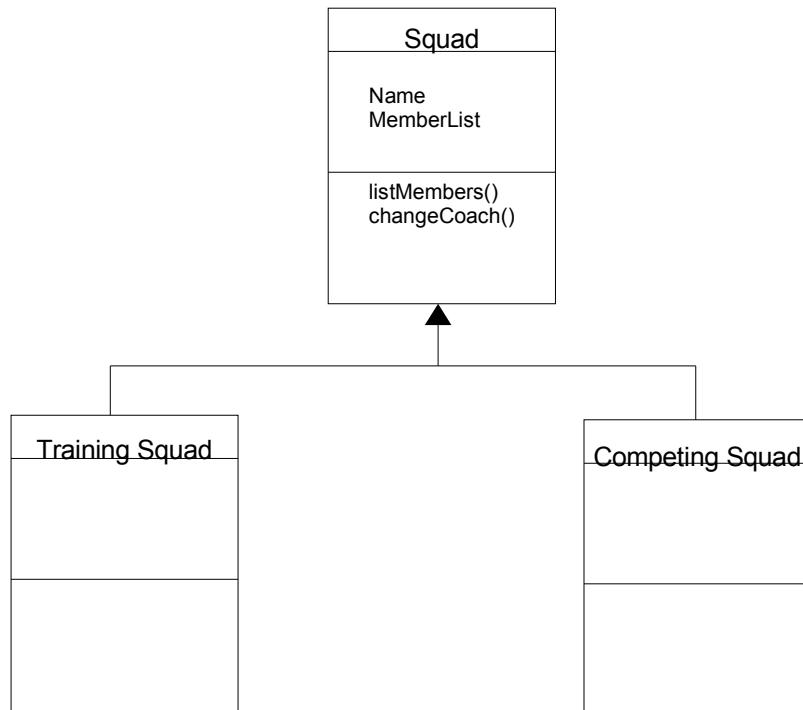


Figure 2.4 Elaborated Squad Hierarchy

Polymorphism is the ability to hide many different implementation behind a single interface. It allows the same message to be handled differently by different objects. Consider the classes defined in Figure 2.5 which will be used to discuss polymorphism¹. A superclass *Person* is modeled with two subclasses *Student* and *Employee*.

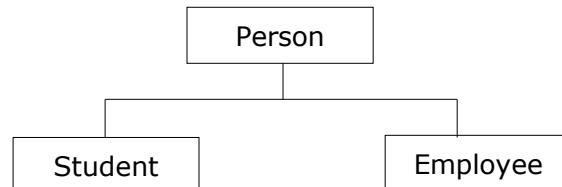


Figure 2.5 Polymorphism Sample

¹ Examples are lifted from the Introduction to Programming Language JEDI Course Materials. Their use has prior approval from the authors.

In Java, consider the following code to implement the classes.

```
public class Person
{
    public String getName(){
        System.out.println("Person Name:" + name);
        return name;
    }
}

public class Student extends Person
{
    public String getName(){
        System.out.println("Student Name:" + name);
        return name;
    }
}

public class Employee extends Person
{
    public String getName(){
        System.out.println("Employee Name:" + name);
        return name;
    }
}
```

Notice that both *Student* and *Employee* have different implementation of the *getName()* method. Consider the following Java Main Method where *ref* is a reference to a class *Person*. The first time that the *ref.getName()* is invoked, it will execute the *getName()* method of *Student* since *ref* references a *studentObject*. The second time that the *ref.getName()* is invoked, it will execute the *getName()* method of *Employee* since *ref* references a *employeeObject*.

```
public static main( String[] args )
{
    Person      ref;

    Student     studentObject = new Student();
    Employee    employeeObject = new Employee();

    ref = studentObject; //Person reference points to a
                        // Student object
    String temp = ref.getName(); //getName of Student
                                //class is called
    System.out.println( temp );

    ref = employeeObject; //Person reference points to an
                        // Employee object
    String temp = ref.getName(); //getName of Employee
                                //class is called
    System.out.println( temp );
}
```

Aggregation is a special kind of association between objects. It models a whole-part relationship between an aggregate (whole) and its parts. Figure 2.6 shows an example of an aggregation. Here, the team is composed of athletes.



Figure 2.6 Team Aggregation

2.2 Object-oriented Process Model

An object-oriented approach in developing software follows the component-based process model. It moves through an evolutionary spiral path. It is highly iterative in nature, and supports reusability. It focuses on the use of an architecture-centric approach by identifying a software architecture baseline upfront.

Software Architecture defines the overall structure of software. It defines the ways in which that structure provides conceptual integrity of a system. It involves making decisions on how the software is built, and normally, controls the iterative and incremental development of the system.

To manage the development in an architecture-centric way, a mechanism to organize work products is used. It is called package. A **package** is a model element that can contain other model elements. It allows us to modularize the development of the software, and serves as a unit of configuration management.

A **subsystem** is a combination of a package (i.e., it can contain other model elements), and a class (i.e., it has behavior). It allows us to define the subsystems of the software. It is realized by one or more interfaces which define the behavior of the system.

A **component** is a replaceable and almost independent part of a system that fulfills a clear function in the context of a well-defined architecture. It may be a source code, runtime code or executable code. It is a physical realization of an abstract design. Subsystems can be used to represent components in the design. A model of a subsystem and component is shown in Figure 2.7.

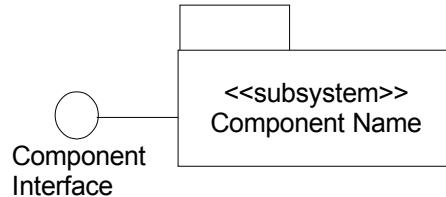


Figure 2.7 Subsystem and Component Model

2.3 Object-oriented Analysis and Design

Two important framework of activities in any software development process is the analysis of the system and the design of the software. This section briefly discusses object-oriented analysis and design phases. It will be expanded later on in the succeeding chapters.

2.3.1 Object-oriented Analysis

The main objective of object-oriented analysis is to develop a series of models that describes the computer software as it works to satisfy a set of customer-defined requirements. The intent is to define a set of classes, their relationships and behavior that is relevant to the system being studied. Since customer requirements influence the creation of the models, this phase is also known as requirements engineering.

Two major models are developed during this phase, the requirements model and analysis model. They depict information, function and behavior within the context of the system, and are modeled as classes and objects.

There are five basic analysis principles applied in this phase.

1. *The information domain is modeled.* The system is analyzed for major data that are needed within the system.
2. *Module function is described.* The system is analyzed to identify major functions that define what needs to be done within the system.
3. *Model behavior is represented.* Using the major data and functions, the behavior of the system is analyzed to understand its dynamic nature. This also includes its interactions with its environment.
4. *The models are partitioned to expose greater detail.* The model elements refined and re-defined to show details that will lead to the design of the software.
5. *Early models represent the essence of the problem while later provide implementation details.* The iterative development of the models should facilitate a smooth transition to the design phase.

Object-Oriented Analysis Methodologies

There are several methods that are used in object-oriented analysis. They are briefly listed below.

1. ***Booch Method.*** This method encompasses both a "micro development process" and a "macro development process".
2. ***The Coad and Yourdon Method.*** This is often viewed as one of the easiest object-oriented analysis methods to learn.
3. ***The Jacobson Method.*** This method is also known as Object-oriented Software Engineering. It is differentiated from others by heavy emphasis on the use-case.

4. **Rumbaugh Method.** It is also known as the Object Modeling Technique which creates three models- object model, dynamic model, and a functional model.
5. **Wirsig-Brock Method.** It does not make a clear distinction between analysis and design tasks. A continuous process that begins with the assessment of customer specification and ends with a proposed design.

All of these methods follows a common set of steps in analyzing a system. Each of these steps are elaborated as the way the work products are developed in the requirements engineering chapter.

STEP 1. Identity customer requirements for the object-oriented system. It would require identifying use-cases or scenarios, and building the requirements model.

STEP 2. Select classes and objects using the requirements model as the guideline.

STEP 3. Identify attributes and operations for each classes.

STEP 4. Define structures and hierarchies that will organize the classes.

STEP 5. Build an object-relationship model.

STEP 6. Build an object-behavior model.

STEP 7. Review the object-oriented analysis model against requirements and standards.

Object-oriented Analysis Main Work Products

Two major models are developed during this phase of the software development process. Namely, they are the Requirements Model and Analysis Model. Their development will be elaborated in the Requirements Engineering chapter. For this section, we briefly define them.

1. **The Requirements Model.** It is a model that attempts to describe the system and its environment. It consists of a use-case model (use-case diagrams and specifications), supplementary documents and glossary.
2. **The Analysis Model.** It is the most important model to be developed in this phase because it serves as a foundation for developing the software necessary to support the system. It consists of an object model (class diagrams) and behavioral model (sequence and collaboration diagrams),

2.3.2 Object-oriented Design

The objective of the object-oriented design phase is to transform the analysis model created in object-oriented analysis into a design model that serves as a blueprint for software construction. It describes the specific organization of data through specifying and redefining attributes, and defining the procedural details of individual class

operations. Similar with analysis, there are five guiding design principles that are applied in this phase.

1. Linguistic modular units
2. Few interfaces
3. Small interfaces and weak coupling
4. Explicit interface
5. Information hiding

Object-oriented Design Methodologies

There are several methods that are used in object-oriented design, and they correspond to the methodologies that were enumerated in object-oriented analysis methodologies. They are briefly listed below.

1. **Booch Method.** Similar with his analysis methodology, it involves a "micro-development" process and a "macro-development" process.
2. **Coad and Yourdon Method.** It addresses not only the application but also the infrastructure for the application.
3. **Jacobson Method.** It emphasizes the traceability of the Object-oriented Software Engineering analysis model.
4. **Rumbaugh Method.** It encompasses a design activity that uses two different levels of abstraction. Namely, they are system design which focuses on the layout for the components that are needed to complete the product, and object design which focuses on the detailed layout of objects.
5. **Wirfs-Brock Method.** It defines a continuous tasks in which analysis leads seamlessly into design.

All of these methodologies follow a common set of steps that are performed in the design. Each of these steps are elaborated as work products are produced in the Design Engineering chapter.

STEP 1. Define the subsystems of the software by defining the data-related subsystems (entity design), control-related subsystem (controller design) and human interaction-related subsystems (boundary design). This should be guided by the software architecture of choice.

STEP 2. Define Class and Object Design.

STEP 3. Define Message Design.

OO Design Main Work Products

There are several work products that are developed during this phase. They are briefly introduced below. Collectively, they are called the **design model**. Their development will be elaborated in the Design Engineering chapter.

1. **Software Architecture.** This refers to the overall structure of the software. It also includes the ways in which the structure provides conceptual integrity

for a system. It is modeled using the package diagrams.

2. **Data Design.** This refers to the design and organization of data. It includes code design and persistent design for the database. It uses the class diagrams and sequence diagrams.
3. **Interface Design.** This refers to the design of the interaction of the system with its environment, particularly, the human-interaction aspects. It includes the dialog and screen designs. Report and form layouts are also included. It uses the class diagrams and state transition diagrams.
4. **Component-level Design.** This refers to the elaboration and design of control classes. This is the most important design because the functional requirements are represented by the control classes. It uses the class diagrams, activity diagrams and state transition diagrams.
5. **Deployment-level Design.** This refers to the design of how the software will be deployed for operational use. It uses the deployment diagram.

2.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the standard language for specifying, visualizing, constructing, and documenting all the work products or artifacts of a software system. In the earlier days, UML has different terminologies as shown in Table 3.

UML	Class	Association	Generalization	Aggregation
Booch	Class	Uses	Inherits	Containing
Coad	Class & Object	Instance Connection	Gen-Spec	Part-whole
Jacobson	Object	Acquaintance Association	Inherits	Consists of
Odell	Object Type	Relationship	Subtype	Composition
Rumbaugh	Class	Association	Generalization	Aggregation
Shlaer/Mellor	Object	Relationship	Subtype	n/a

Table 3 Different UML Terminologies

However, its notation was unified by Booch, Rumbaugh and Jacobson. The Object Management Group maintains a set of standards for using UML for technical exchange of models and design. Now, other contributors augments the definition of UML.

UML is not a method or methodology. It does not indicate a particular process. It is not a programming language. It is basically a standard set of modeling tool used to develop work products of the software, which most of it will be models.

2.4.1 Modeling Activity

Software development is a complex activity; it is extremely difficult to carry out the necessary task if all details are in one's memory. In any development project that aims to produce useful work products, the main focus of both analysis and design activities is on models. A **model** is a pattern of something to be made. It is both abstract and visible. Software is not tangible for the users and by nature, are abstracts. However, it is constructed by a developing team who need to see each other's model.

A model is a representation of something in the real world. They are useful in a variety of ways because they differ from the things that they represent. Particularly, they are:

- built quicker and easier than the real objects they represent.
- used in simulation to better understand the thing they represent.
- modified to evolve as one learns about a task or problem.

- used to represent details of the models that one chooses to see and others ignored; they are basically an abstraction of the real objects.
- a representation of real or imaginary objects in any domain.

A useful model has just the right amount of detail and structure, and represents only what is important for the task at hand. Usually, developers model complex situation within a human activity system. They need to model what different stakeholders think about the situation. Therefore, they need to be rich in meaning. The models of the analysis phase (requirements and analysis model) must be accurate, complete and unambiguous. Without this, the work of the development team will be much more difficult. At the same time, it must not include premature decisions about how the system is built to meet user's requirements. Otherwise, the developing team may later find their freedom of actions to restricted. Most models are in the form of diagrams that are accompanied by textual descriptions, and logical or mathematical specifications of processes and data.

Systems analysts and designers use diagrams to create models of systems. Diagrams are used extensively in order to:

- understand object structures and relationships.
- share ideas with others
- solicit new ideas and possibilities
- test ideas and make predictions

There are a lot of modeling techniques that can be found. Some general rule are listed below. Modeling techniques should be:

- Simple. Show only what needs to be shown.
- Internally consistent. Diagrams should support one another.
- Complete. Show all that needs to be shown
- Hierarchically Represented. Break down the system into smaller components. Then, show the details of the lower-level.

The Unified Modeling Language (UML) provides a set of modeling diagrams used for modeling systems. The diagrams are composed of four general elements.

- *Icons.* They represent the atomic symbols of the diagrams. They are abstract shapes that are connected to other shapes.
- *Two-dimensional Symbols.* They are similar to icons except that they have compartments that can contains other symbols, icons or strings. Icon do not contains other symbols.
- *Paths.* They represent the link from one icon or symbol to another. They, normally, represent the flow from one shape to another.
- *Strings.* They are used to represent labels, descriptions, names etc. of icons, symbols and paths.

The UML Specification provides a formal definition of how to use UML Diagrams. It provides the grammar of UML (syntax) which includes the meaning of the elements, and the rules of how the elements are combined (semantics).

A single diagram illustrates or documents some aspect of a system. A model, on the other hand, provides a complete view of a system at a particular stage and from a

particular perspective. For example, the models developed during the analysis phase give a complete view of the system from a problem domain. It tries to capture essentially the what-aspect of the system such as what functions, what data etc. It will consist of several diagrams to cover all aspects of the problem domain. The models, developed during design phase, gives a complete view of the software system to be built. It is a view coming from the solution domain. It will probably consist of diagrams that represent components of the software such as dialog designs for the screens, database elements, controller functions etc.

The models that are produced during the project development changes as the project progresses. Normally, they change in terms of:

- *Level of Abstraction.* As the project progresses, the model would become less abstract and be more concrete. For example, one may start off with classes that represent the kind of objects that we will find in the system such as athletes, coach, squad and team. By the time, one gets to the end of the design, one can already implement the classes with attributes and operations. The classes would also have additional supporting classes such as brokers, proxies for the target deployment platform.
- *Degree of Formality.* The degree of formality with which operations, attributes and constraints are defined will increase as the project progresses. Initially, classes and attributes may be loosely defined using Structured English or whatever language is used by the development team. By the time it reaches the end of the design and ready for implementation, attributes and operations are defined using the target programming language such as Java. For example, at the analysis phase, one uses athlete as the name of the class. At the design phase, it can be a persistent class holding information about an athlete and be named as DBAthlete class.
- *Level of Detail.* As the project progresses, the different models represent the same view but shows a different level of detail. The models get more details as it progresses through the development process. For example, the first use-case diagram may show only the obvious use-cases that are apparent from the first iteration. At the second iteration, the use-case diagram may be elaborated with more detail, and additional use-cases may emerge. After the third iteration, it may include more structured description on how users will interact with the use-cases and with relationships with other use-cases.

Any phase of the project will consist of a number of iterations, and that number will depend on the complexity of the system being developed. As the project progresses, the level of abstraction, degree of formality and level of detail should be established properly so that work products will be useful.

2.4.2 UML Baseline Diagrams

There are nine (9) baseline diagrams that are specified in UML. They are briefly discussed in this section. Their use will be later on presented in detail as work products are produced within the major phases of the development process.

Use Case Diagram

The **Use Case Diagram** provides a basis of communication between end-users and developers in the planning of the software project. It captures important user-visible functions which may be small or large. It achieves a discrete goal for the user. It attempts to model the system environment by showing the external actors and their connection to the functionality of the system. Figure 2.8 shows an example of a Use Case Diagram.

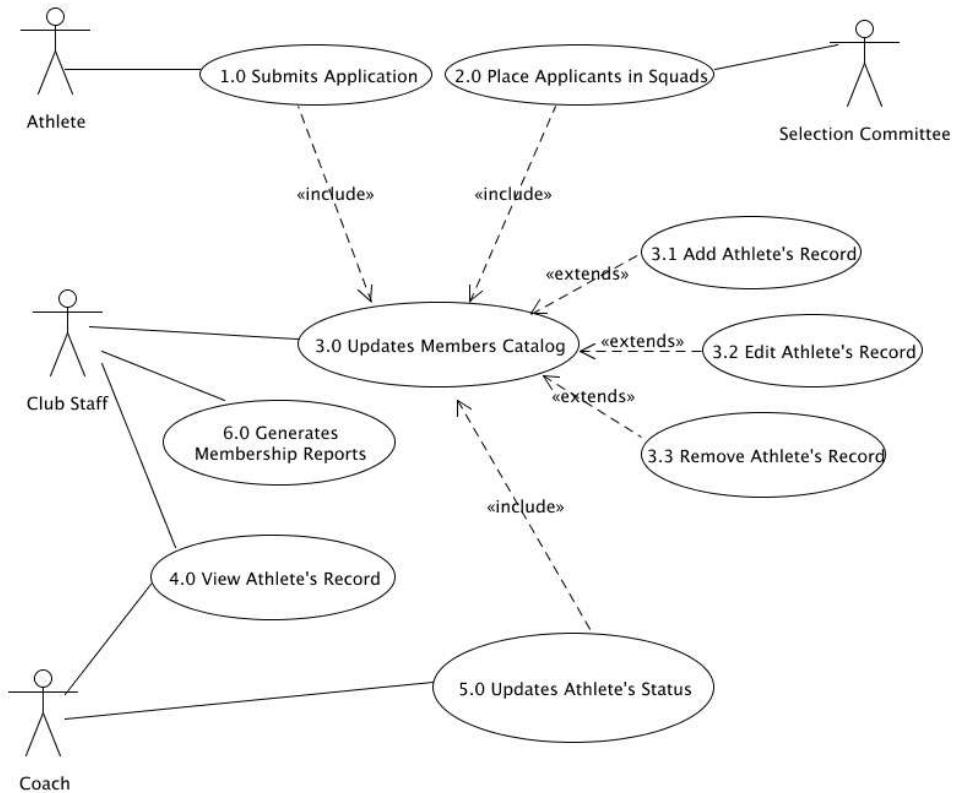


Figure 2.8 Sample Use Case Diagram

Class Diagram

The **Class Diagram** shows the static structure of the domain abstractions (classes) of the system. It describes the types of objects in the system and the various kinds of static relationship that exists among them. It shows the attributes and operations of a class and constraints for the way objects collaborate.

This modeling tool produces the most important diagram of the system because it is used to model an understanding of the application domain (essentially part of a human activity system), and objects are also understood as part of the resulting software system. Figure 2.9 shows an example of a class diagram.

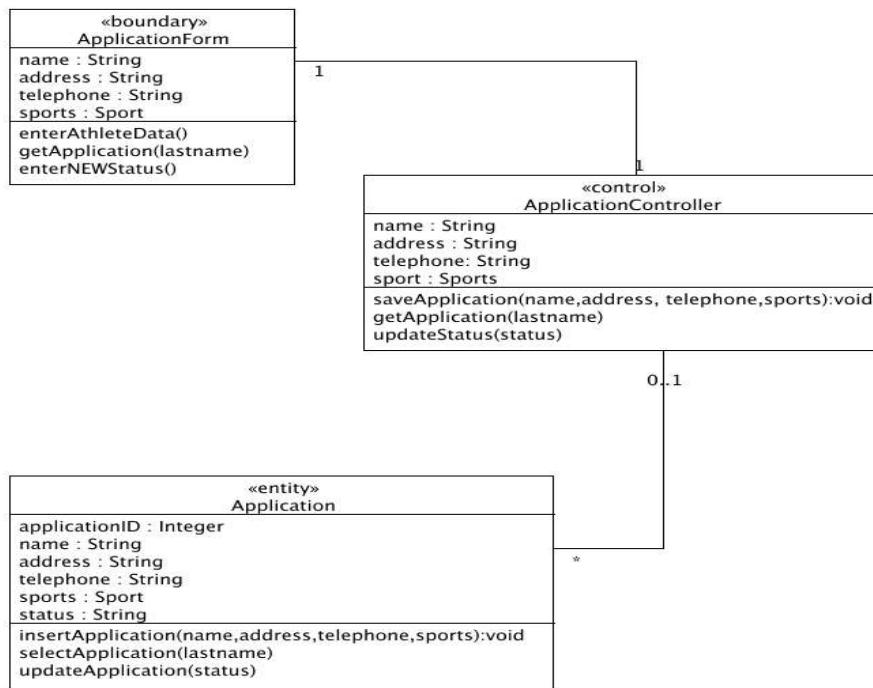


Figure 2.9 Sample Class Diagram

Package Diagram

The **Package Diagram** shows the breakdown of larger systems into logical grouping of smaller subsystems. It shows groupings of classes and dependencies among them. A dependency exists between two elements if changes to the definition of one element may cause changes to the other elements. Figure 2.10 shows an example of a package diagram.

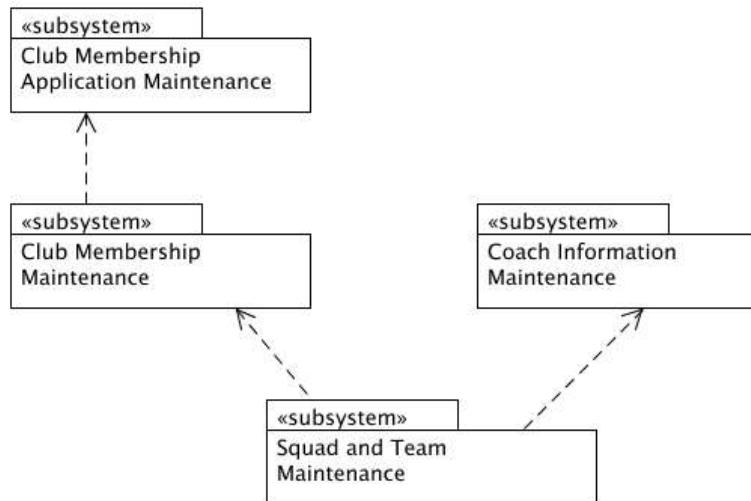


Figure 2.10 Sample Package Diagram

Activity Diagram

The **Activity Diagram** shows the sequential flow of activities. Typically, it is used to show the behavior of an operation, use-case flow of events, or event trace. It complements the use case diagram by showing the work flow of the business. It encourages the discovery of parallel processes, which helps eliminate unnecessary sequences in the business process. It complements the class diagram because it graphically shows the flow of each operations (similar to a flow chart). Figure 2.11 shows an example of an activity diagram.

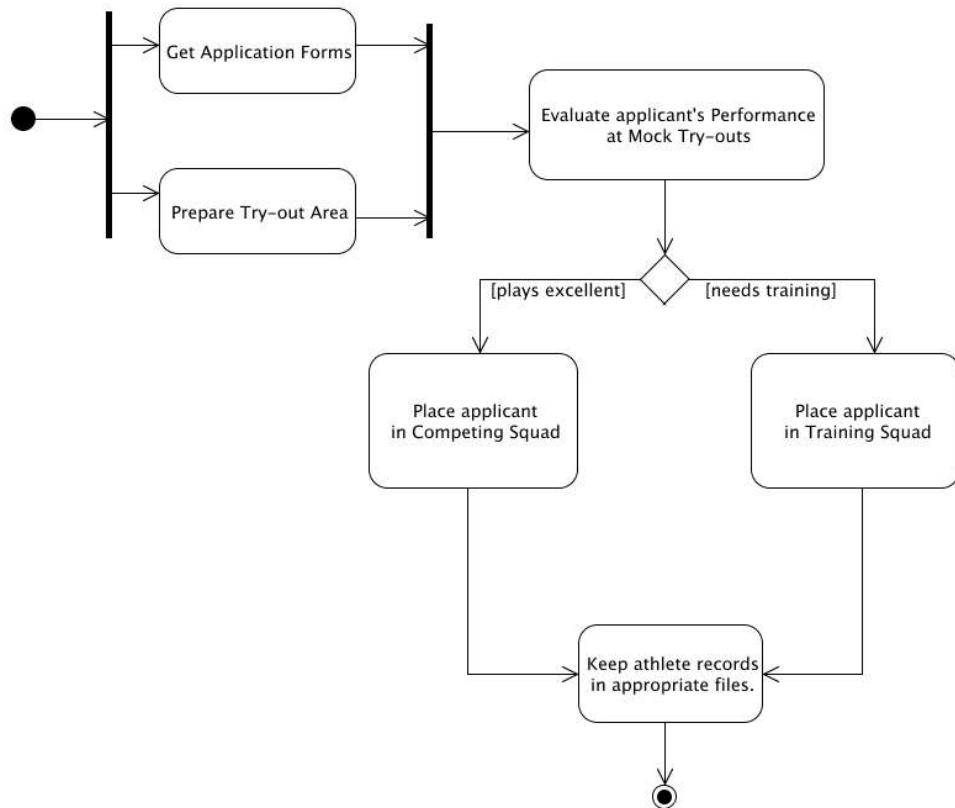


Figure 2.11 Sample Activity Diagram

Sequence Diagram

The **Sequence Diagram** shows the dynamic collaborative behavior between objects for a sequence of message sends between them in a sequence of time. Time sequence is easier to see in the sequence diagram read from top to bottom. Choose sequence diagram when only the sequence of the operations need to be shown. Figure 2.12 shows an example of a sequence diagram.

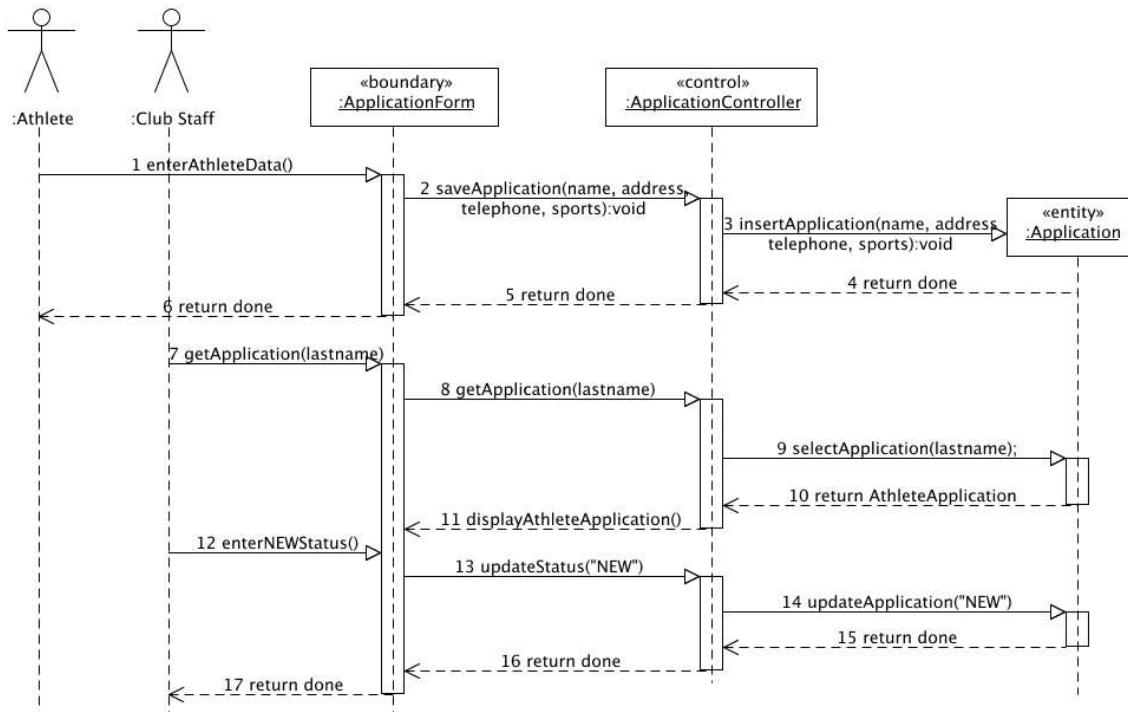


Figure 2.12 Sample Sequence Diagram

Collaboration Diagram

The **Collaboration Diagram** shows the actual objects and their links which represent the "network of objects" that are collaborating. Time sequence is shown by numbering the message label of the links between objects. Collaboration diagram is best suited when the objects and their links facilitate understanding of the interaction among objects, and the sequence of time is not important. Figure 2.13 shows an example of a collaboration diagram.

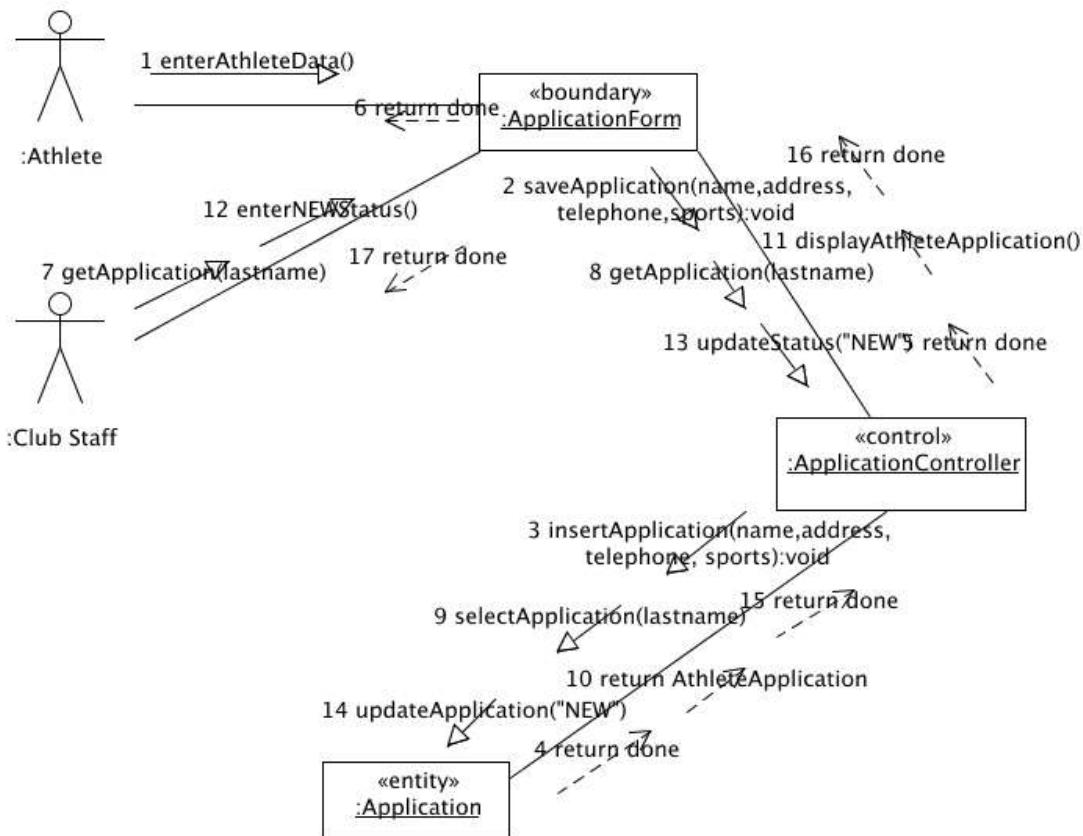


Figure 2.13 Sample Collaboration Diagram

State Transition Diagram

It shows all the possible states that objects of the class can have and which events cause them to change. It shows how the object's state changes as a result of events that are handled by the object. Good to use when a class has complex life cycle behavior. Figure 2.14 shows an example of a state transition diagram.

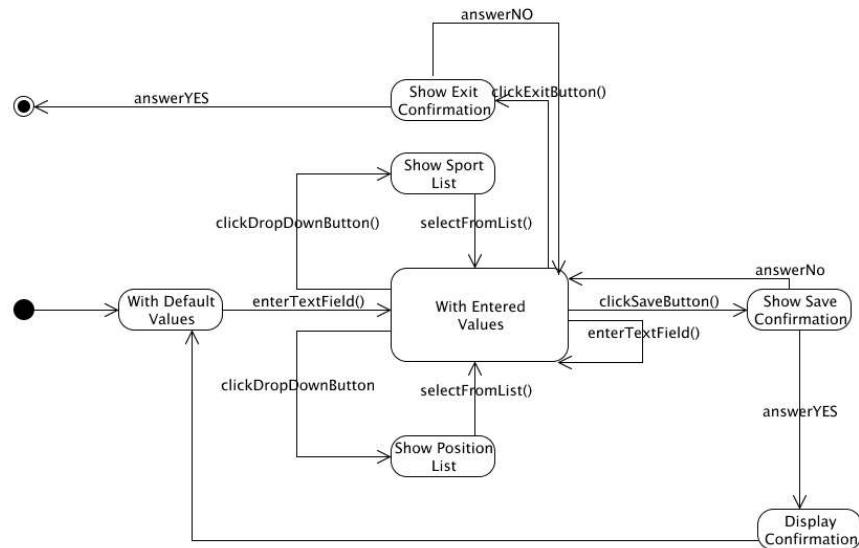


Figure 2.14 Sample State Transition Diagram

Component Diagram

The Component Diagram shows the software components in terms of source code, binary code, dynamically linked libraries etc. Figure 2.15 is an example of a component diagram.

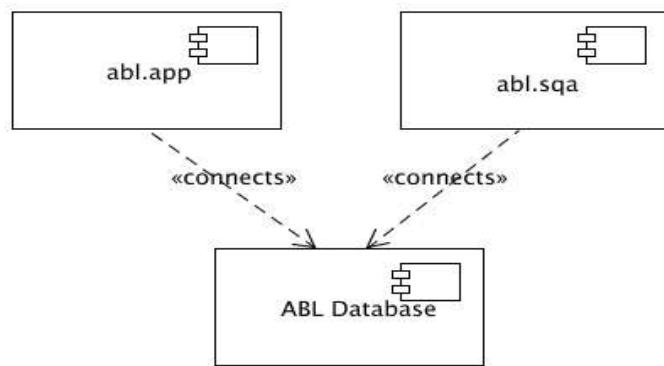


Figure 2.15: Sample Component Diagram

Deployment Diagram

The Deployment Diagram shows the physical architecture of the hardware and software of the system. It highlights the physical relationship among software and hardware components in the delivered system. Components on the diagram typically represent physical modules of code and correspond exactly to the package diagram. Figure 2.16 is an example of a deployment diagram.

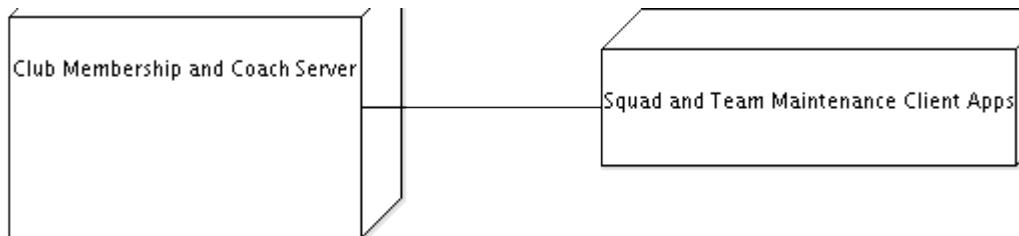


Figure 2.16 Sample Deployment Diagram

Creating a UML Project Using Java™ Studio Enterprise 8

Java™ Studio Enterprise 8 is an Integrated Development Environment (IDE) that incorporates a multi-window editor and mechanism for managing files that make up the projects including files for models, source codes and test cases. It has links to the compiler so that codes can be compiled within and debugger to help the programmer step through the code to find errors.

To create a UML Project, simply follow the steps.

1. Start the Java™ Studio Enterprise 8 Application. A window similar to the window shown in Figure 2.17 is displayed.

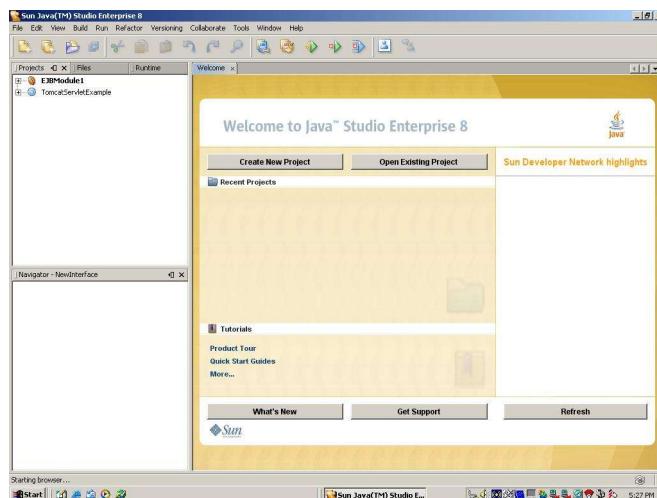


Figure 2.17: Welcoming Window of Java™ Studio Enterprise

2. Click the **Create New Project** button. This will display the **New Project** Dialog box as shown in Figure 2.18. In the **Categories** list, select **UML**. In the **Projects** list, select **Platform-Independent Model**.

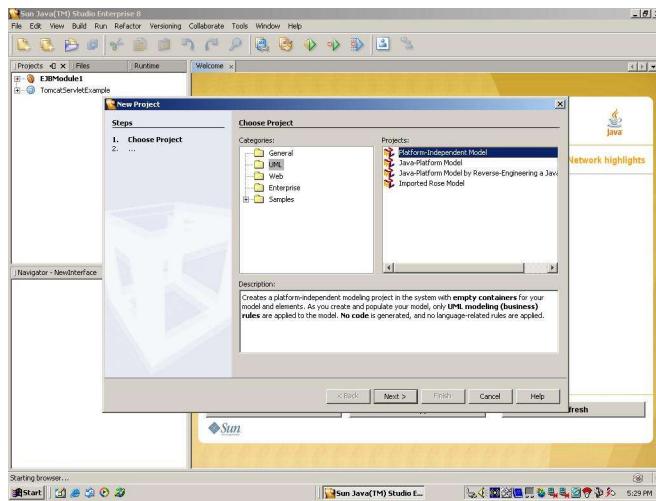


Figure 2.18: New Project Dialog Box

3. Click the **Next** button. The **New Platform-Independent Model** Dialog Box will appear as shown in Figure 2.19. At the **Project Name** text field, enter the name of the project. In the example, the project name is "**Ang Bulilit Liga**". Next, at the **Project Location** text field, enter the directory to which the project files will reside. In the example, it is in the **AngBulilitLiga** directory under the home directory.

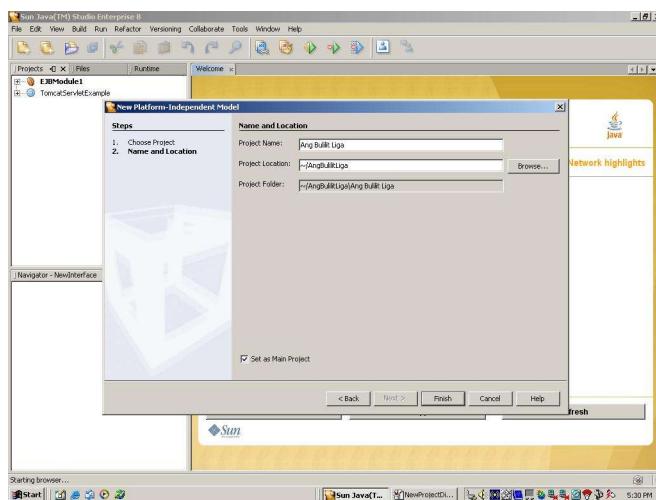


Figure 2.19: New Platform-Independent Dialog Box

4. Click the **Finish** Button. The **New Wizard** Dialog Box will appear which is used to create the first diagram of the project. It is shown in Figure 2.20. At the **Diagram Types** list, select the type of diagram you will create. In the example, it is a **Use Case Diagram**. At the **Diagram Name** text field, enter the name of the diagram. In the example, it is the **Club Membership Maintenance**.

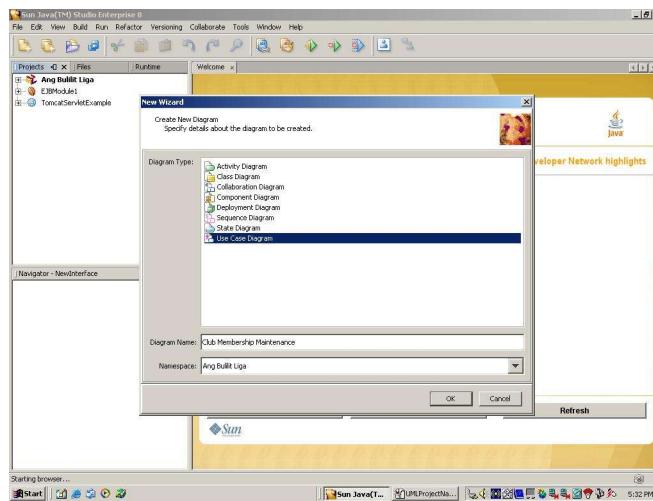


Figure 2.20: New Wizard Dialog Box

5. Click the **OK** button. The window shown in Figure 2.21 is displayed. At this point, you can now create your diagram. The modeling elements are presented in the **Modeling Palette** found at the right side of the window. The **Project Tab** found at the left side presents the organization of the models or files of the project. The files and models are under the "Ang Bulilit Liga" Project Space.

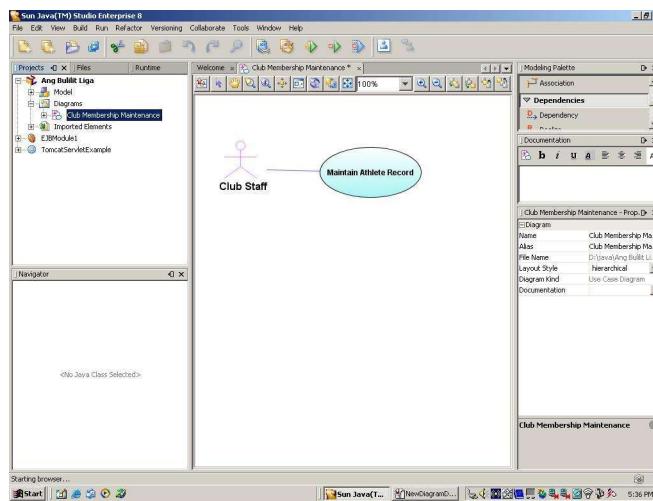


Figure 2.21: Model Design Window

3 Requirements Engineering

Designing and building a computer system is challenging, creative and just plain fun. However, developing a good software that solves the wrong problem serves no one. It is important to understand user's needs and requirements so that we solve the right problem and build the right system.

In this chapter, we will be discussing the concepts and dynamics of requirements engineering. It is a software development phase that consists of seven distinct tasks or activities which has a goal of understanding and documenting stakeholder's requirements. Two important models will be built, namely, **requirements model** which is the system or problem domain model, and the **analysis model** which serves as the base model of the software (solution model). The Requirements Traceability Matrix (RTM) will be introduced to help software engineers manage requirements, and the requirements metrics and its significance will also be discussed.

3.1 Requirements Engineering Concepts

Requirements Engineering allows software developers to understand the problem they are solving. It encompasses a set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end-user will interact with the software. It provides an appropriate mechanism for understanding stakeholder's needs, analyzing requirements, determining feasibility, negotiating a reasonable solution, specifying the solutions clearly, validating specification and managing requirements as they are transformed into a software system.

Significance to the Customer, End-users, Software Development Team and Other Stakeholders

Requirements Engineering provides the basic agreement between end-users and developers on what the software should do. It is a clear statement on the scope and boundaries of the system to be analyzed and studied. It gives stakeholders an opportunity to define their requirements understandable to the development team. This can be achieved through different documents, artifacts or work products such as use case models, analysis models, features and functions list, user scenarios etc.

Designing and building an elegant computer program that solves the wrong problem is a waste. This is the reason why it is important to understand what customer wants before one begins to design and build a computer-based system. Requirements Engineering builds a bridge to design and construction. It allows the software development team to examine:

- the context of the software work to be performed
- the specific needs that design and construction must address
- the priorities that guide the order in which work is to be completed
- the data, functions and behaviors that will have a profound impact on the resultant design

Requirements Engineering, like all other software engineering activities, must be adapted

to the needs of the process, projects, products and the people doing the work. It is an activity that starts at inception until a base model of the software can be used at the design and construction phase.

3.2 Requirements Engineering Tasks

There are seven distinct tasks to requirements engineering, namely, ***inception, elicitation, elaboration, negotiation, specification, validation*** and ***management***. It is important to keep in mind that some of these tasks occurs in parallel and all are adapted to the needs of the project. All strive to define what customer wants, and all serve to establish a solid foundation for the design and construction of what the customer needs.

3.2.1 Inception

In general, most software projects begin when there is a problem to be solved, or an opportunity identified. As an example, consider a business that discovered a need, or a potential new market or service. At ***inception***, the problem scope and its nature is defined. Software engineer asks a set of context free questions with the intent of establishing a basic understanding of the problem, people who want the solution, the nature of the solution, and the effectiveness of the preliminary communication and collaboration between end-users and developers.

Initiating Requirements Engineering

Since this is a preliminary investigation of the problem, a Q&A (Question and Answer) Approach or Interview is an appropriate technique in understanding the problem and its nature. Enumerated below are the recommend steps in initiating the requirements engineering phase.

STEP 1: Identify stakeholders.

A **stakeholder** is anyone who benefits in a direct or indirect way from the system which is being developed. The business operations managers, product managers, marketing people, internal and external customers, end-users, and others are the common people to interview. It is important at this step to create a list of people who will contribute input as requirements are elicited. The list of users will grow as more and more people get involved in elicitation.

STEP 2: Recognize multiple viewpoints.

It is important to remember that different stakeholder would have a different view of the system. Each would gain different benefits when the system is a success; each should have different risks if the development fails. At this step, categorize all stakeholder information and requirements. Also, identify requirements that are inconsistent and in conflict with one another. It should be organized in such a way that stakeholders can decide on a consistent set of requirements for the system.

STEP 3: Work toward collaboration.

The success of most projects would rely on collaboration. To achieve this, find areas within the requirements that are common to stakeholders. However, the challenge here is addressing inconsistencies and conflicts. Collaboration does not mean that a committee decides on the requirements of the system. In many cases, to resolve conflicts a project champion, normally a business manager or senior technologist, decides which requirements are included when the software is developed.

STEP 4: Ask the First Question.

To define the scope and nature of the problem, questions are asked to the customers and stakeholders. These questions may be categorized. As an example, consider the following questions:

Stakeholder's or Customer's Motivation:

1. Who is behind the request for this work?
2. Why are they requesting such a work?
3. Who are the end-users of the system?
4. What are the benefits when the system has been developed successfully?
5. Are there any other ways in providing the solution to the problem? What are the alternatives.

Customer's and Stakeholder's Perception:

1. How can one characterize a "good" output of the software?
2. What are the problems that will be addressed by the software?
3. What is the business environment to which the system will be built?
4. Are there any special performance issues or constraints that will affect the way the solution is approached?

Effectiveness of the Communication:

1. Are we asking the right people the right questions?
2. Are the answers they are providing "official"?
3. Are the questions relevant to the problem?
4. Am I asking too many questions?
5. Can anyone else provide additional information?
6. Is there anything else that I need to know?

Inception Work Product

The main output or work product of inception task is a one- or two- page(s) of **product request** which is a paragraph summary of the problem and its nature.

3.2.2 Elicitation

After inception, one moves onward to **elicitation**. Elicitation is a task that helps the customer define what is required. However, this is not an easy task. Among the problems encountered in elicitation are discussed below:

1. *Problems of Scope.* It is important that the boundaries of the system be clearly and properly defined. It is important to avoid using too much technical detail because it may confuse rather than clarify the system's objectives.
2. *Problems of Understanding.* It is sometimes very difficult for the customers or users to completely define what they needed. Sometimes they have a poor understanding of the capabilities and limitations of their computing environment, or they don't have a full understanding of the problem domain. They sometimes may even omit information believing that it is obvious.
3. *Problems of Volatility.* It is inevitable that requirements change overtime.

To help overcome these problems, software engineers must approach the requirements gathering activity in an organized and systematic manner.

Collaborative Requirements Gathering

Unlike inception where Q&A (Question and Answer) approach is used, elicitation makes use of a requirements elicitation format that combines the elements of problem solving, elaboration, negotiation, and specification. It requires the cooperation of a group of end-users and developers to elicit requirements. They work together to:

- identify the problem
- propose elements of the solution
- negotiate different approaches
- specify a preliminary set of solution requirements

Joint Application Development is one collaborative requirement gathering technique that is popularly used to elicit requirements.

The tasks involved in elicitation may be categorized into three groups, namely, pre-joint meeting tasks, joint meeting tasks and post-joint meeting tasks.

Pre-Joint Meeting Tasks

1. If there is no product request, one stakeholder should write one.
2. Set the place, time and date of the meeting.
3. Select a facilitator.
4. Invite the members of the team which may be the software team, customers, and other stakeholders).
5. Distribute the product request to all attendees before the meeting.
6. Each attendee is asked to make the following:
 - a list of objects that are part of the environment that surrounds the system

- a list of other objects that are produced by the system
- a list of objects that are used by the system to perform its functions
- a list of services (processes or functions) that manipulate or interact with the objects
- a list of constraints such as cost, size and business rules
- a list of performance criteria such as speed, accuracy etc.

Note that the lists are not expected to be exhaustive but are expected to reflect the person's perception of the system.

Joint Meeting Tasks

1. The first topic that needs to be resolved is the need and justification of the new product. Everyone should agree that the product is justified.
2. Each participant presents his list to the group.
3. After all participants present, a combined list is created. It eliminates redundant entries, adds new ideas that come up during the discussion, but does not delete anything.
4. The consensus list in each topic area (objects, services, constraints and performance) is defined. The combined list from the previous task is shortened, lengthened, or reworded to reflect the product or system to be developed.
5. Once the consensus list is defined, the team is divided into sub-teams. Each will be working to develop the **mini-specifications** for one or more entries on the consensus list. The mini-specification is simply an elaboration of the item in the list using words and phrases.
6. Each sub-team, then, presents its mini-specification to all attendees. Additions, deletions and further elaboration are made. In some cases, it will uncover new objects, services, constraints, or performance requirements that will be added to the original lists.
7. In some cases, issues arise that cannot be resolved during the meeting. An issue list is maintained and they will be acted on later.
8. After each mini-specification is completed, each attendee makes a list of validation criteria for the product or system and presents the list to the team. A consensus list of validation criteria is created.
9. One or more participants are assigned the task of writing a complete draft specification using all inputs from the meeting.

Post-Joint Meeting Tasks

1. Compile the complete draft specifications of the items discussed in the meeting.
2. Prioritize the requirements. One can use the Quality Function Technique or MoSCoW Technique.

Quality Function Deployment

Quality Function Deployment is a technique that emphasizes an understanding of what is valuable to the customer. Then, deploy these values throughout the engineering process. It identifies three types of requirements:

1. Normal Requirements

These requirements directly reflect the objectives and goals stated for a product or system during meetings with the customer. It means that if the requirements are present, the customer is satisfied.

2. Expected Requirements

These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. The absence of these requirement may cause for significant dissatisfaction. Examples of expected requirements are ease of human or machine interaction, overall operation correctness and reliability, and ease of software installation.

3. Exciting Requirements

These requirements reflect features that go beyond the customer's expectations and prove to be very satisfying when present.

With the succeeding meetings with the team, **value analysis** is conducted to determine the relative priority of requirement based on three deployments, namely, **function deployment**, **information deployment**, and **task deployment**. Function deployment is used to determine the value of each function that is required for the system. Information deployment identifies both data objects and events that the system must consume and produce. This is related to a function. Task deployment examines the behavior of the product or system within the context of its environment. From the value analysis, each requirements are categorized based on the three types of requirements.

MoSCoW Technique

Each requirement can be evaluated against classes of priority as specified in Table 4. During the software engineering process, a short meeting should be conducted to review and probably, reassign priorities.

Classification	Meaning
Must Have	This requirement will be included in the delivered product.
Should Have	The current project plan indicates that this requirement will be included. If circumstances change, it may be traded out.
Could Have	The current project plan does not indicate that this requirement will be included. If circumstances change, it may be traded in.
Won't Have	This requirement will not be included in the delivered product.

Table 4 Classification of Priorities

Elicitation Work Product

The output of the elicitation task can vary depending on size of the system or product to be built. For most systems, the output or work products include:

- A statement of need and feasibility
- A bounded statement of scope for the system or product
- A list of customer, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment
- A priority list of requirements, preferably, in terms of functions, objects and domain constraints that apply to each

3.2.3 Elaboration

The information obtained from the team during inception and elicitation is expanded and refined during ***elaboration***. This requirement engineering task focuses on defining, redefining and refining of models, namely, the requirements model (system or problem domain) and analysis model (solution domain). It tries to model the "WHAT" rather than the "HOW".

The ***requirements model*** is created using methodologies that capitalizes on user scenarios which define the way the system is used. It describes how the end-users and actors interact with the system.

The ***analysis model*** is derived from the requirements model where each scenario is analyzed to get the analysis classes, i.e., business domain entities that are visible to the end-user. The attributes of each analysis class are defined and responsibilities that are required by each class are identified. The relationships and collaboration between classes are identified and a variety of supplementary UML diagrams are produced. The end-result of elaboration is an analysis model that defines the informational, functional and behavioral domain of the problem. The development of these models will be discussed in the *Requirements Analysis and Model*, and *Requirements Specifications* section of this chapter.

Elaboration Work Product

The requirements model and the analysis model are the main work product of this task.

3.2.4 Negotiation

In ***negotiation***, customers, stakeholders and software development team reconcile conflicts. Conflicts arise when customers are asking for more than what the software development can achieve given limited system resources. To resolve these conflicts, requirements are ranked, risk associated with each requirements are identified and analyzed, estimates on development effort and costs are made, and delivery time set. The purpose of negotiation is to develop a project plan that meets the requirements of the user while reflecting real-world constraints such as time, people and budget. It means that customer gets the system or product that satisfies majority of the needs, and the software team is able to realistically work towards meeting deadlines and budgets.

The Art of Negotiation

Negotiation is a means of establishing collaboration. For a successful software development project, collaboration is a must. Below are some guidelines in negotiating with stakeholders.

1. *Remember that negotiation is not competition.* Everybody should compromise. At some level, everybody should feel that their concerns have been addressed, or that they have achieved something.
2. *Have a strategy.* Listen to what the parties want to achieve. Decide on how we are going to make everything happen.
3. *Listen effectively.* Listening shows that you are concerned. Try not to formulate your response or reaction while the other is speaking. You might get something that can help you negotiate later on.
4. *Focus on the other party's interest.* Don't take hard positions if you want to avoid conflict.
5. *Don't make it personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box.
7. *Be ready to commit.* Once an agreement has been reached, commit to the agreement and move on to other matters.

3.2.5 Specification

A **specification** is the final artifact or work product produced by the software engineer during requirements engineering. It serves as the foundation for subsequent software engineering activities, particularly, the design and construction of the software. It shows the informational, functional and behavioral aspects of the system. It can be written down as a document, a set of graphical models, a formal mathematical model, a prototype or any combination of these. The models serve as your specifications.

3.2.6 Validation

The work products produced as a consequence of requirements engineering are assessed for quality during **validation** step. It examines the specification to ensure that all software requirements have been stated clearly and that inconsistencies, omissions, and errors have been detected and corrected. It checks the conformance work products to the standards established in the software project.

The **review team** that validates the requirements consists of software engineers, customers, users, and other stakeholders. They look for errors in content or interpretation, areas where clarification is required, missing information, inconsistencies, conflicting and unrealistic requirements.

Requirements Validation Checklist

As the models are built, they are examined for consistency, omission, and ambiguity. The requirements are prioritized and grouped within packages that will be implemented as software increments and delivered to the customer. Questions as suggested by Pressman are listed below to serve as a guideline for validating the work products of the requirement engineering phase.

1. Is each requirement consistent with the overall objective for the system or product?
2. Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is not appropriate at the stage?
3. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
4. Is each requirement bounded and clear?
5. Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
6. Do any of the requirements conflict with other requirements?
7. Is each requirement achievable in the technical environment that will house the system or product?
8. Is each requirement testable, once implemented?
9. Does the requirement model properly reflect the information, function and behavior of the system to be built?
10. Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
11. Have the requirements pattern been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that all work products reflect the customer's needs so that it provides a solid foundation for design and construction.

3.2.7 Management

It is a set of activities that help the project team identify, control, and track requirements and their changes at any time as the project progresses. Requirements management starts once they are identified. Each requirement is assigned a unique identifier. Once requirements have been identified, traceability tables are developed.

The **Requirements Traceability Matrix (RTM)** is discussed in the *Requirements Traceability Matrix* section of this chapter which will help software engineers manage requirements as the development process progresses.

3.3 Requirements Analysis and Model

During elaboration, information obtained during inception and elicitation is expanded and refined to produce two important models- requirements model and analysis model. The requirements model provides a model of the system or problem domain. In this section, we will be discussing the requirements model and how it is built.

3.3.1 The Requirements Model

Rational Rose defines the Requirements Model as illustrated in Figure 3.1². It consists of three elements, specifically, **Use Case Model**, **Supplementary Specifications**, and **Glossary**.

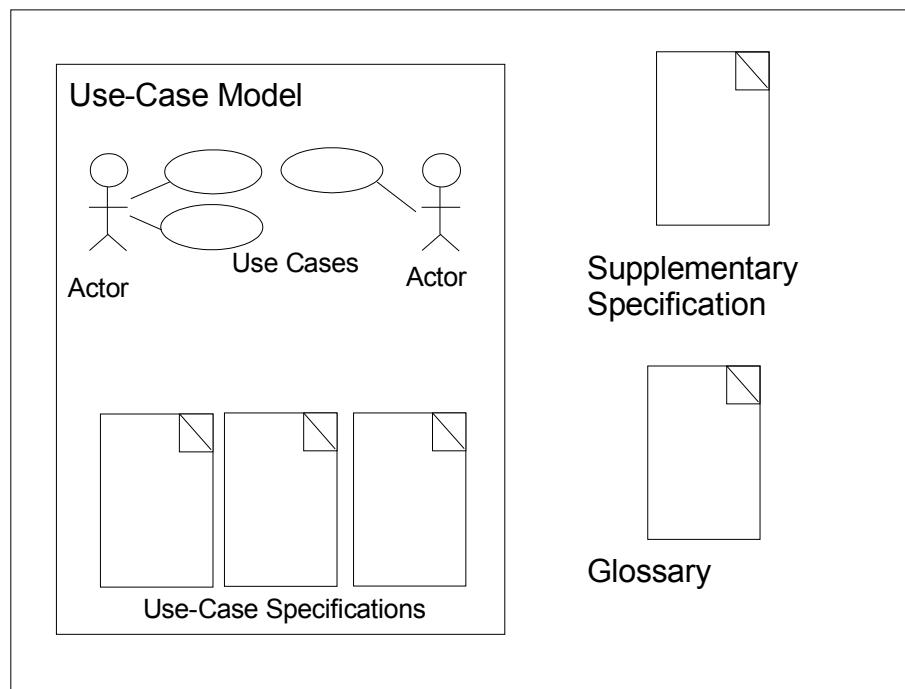


Figure 3.1 Requirements Model

Use Case Model

It is used to describe what the system will do. It serves as a contract between the customers, end-users and system developers. For customers and end-users, it is used to validate that the system will become what they expect it to be. For the developers, it is used to ensure that what they build is what is expected. The Use Case Model consists of two parts, namely, the use case diagrams and use case specifications.

1. The **use case diagram** consists of actors and use cases. It shows the functionality that the system provides and which users will communicate with the system. Each use case in the model is described in detail using the use case specifications. The **use case specifications** are textual documents that specify properties of the use cases such as flow of events, pre-conditions, post-conditions etc. The UML diagramming

² Object-oriented Analysis and Design Using the UML, Rational Rose Corporation, 2000, page 3-5

tool used to define the Use case Model is the Use case Diagram.

Supplementary Specifications

It contains those requirements that don't map to a specific use case. They may be ***non-functional requirements*** such as maintainability of the source codes, usability, reliability, performance, and supportability, or ***system constraints*** that restricts our choices for constructing a solution to the problem such as the system should be developed on Solaris and Java. It is an important complement to the Use Case Model because with it we are able to specify a complete system requirements.

Glossary

It defines a common terminology for all models. This is used by the developers to establish a common dialect with customers and end-users. There should only be one glossary per system.

3.3.2 Scenario Modeling

The Use Case Model is a mechanism for capturing the desired behavior of the system without specifying how the behavior is to be implemented. The point of view of the actors interacting with the system are used in describing the scenarios. ***Scenarios*** are instances of the functionality that the system provides. It captures the specific interactions that occur between producers and consumers of data, and the system itself. It is important to note that building the model is an iterative process of refinement.

Use Case Diagram of UML

As was mentioned in the previous section, the Use Case Diagram of UML is used as the modeling tool for the Use case Model. Table 5 shows the basic notation.

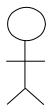
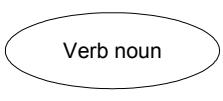
<i>Basic Notation</i>	<i>Name</i>	<i>Meaning</i>
 Actor-name	Actor	It represents a coherent set of roles that users of the system play when interacting with use cases. It calls on the system to deliver one of its services. It can be a person, device or another system. Actors are named using nouns.
	Use Case	It describes the functions that the system performs when interacting with actors. It is a sequence actions that yields an observable result to actors. It is described using a verb-noun phrase.
or 	Association	It shows the relationship or association between an actor and use cases, and/or between use cases.

Table 5 Use Case Diagram Basic Notation

The Use Case Model can be refined to include ***stereotypes on associations*** and ***generalization of actors and use cases***. Stereotypes in UML is a special use of model elements that is constrained to behave in a particular way. They are shown by using a keyword in matched guillemets (<>) such as <<extend>> and <<include>>. Generalization or specialization follows the same concepts that was discussed in object-oriented concepts. The Table 6 shows the enhanced notation of the Use Case Diagram in UML.

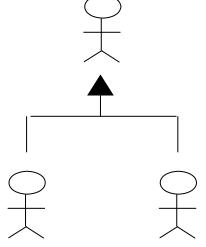
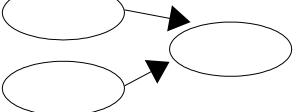
Enhanced Notation	Name	Meaning
<<extend>> 	Extend	It is used to show that a use case provides additional functionality that may be required in another use case.
<<include>> 	Include	It is used to show that when one use case is used, the other (the included) will also be used.
	Actor Generalization	There may be cases that it would be best to show a super-actor communicating with a use case rather than all actors communicating with the same use case, particularly, when all of them will be interacting with the system using the same functionality.
	Use Case Generalization	There may be similar use cases where the common functionality is best represented by generalizing out that functionality into a super-use case.

Table 6 Use Case Diagram Expanded Notation

Developing the Use Case Model

STEP 1: Identify actors.

Identify the external actors that will interact with the system. It may be a person, device or another system. As an example, Figure 3.2 identifies the actors for the **Club Membership Maintenance** of the case study.



Figure 3.2 Club Membership Maintenance Actors

Two actors were identified, namely, **club staff** and **coach**.

STEP 2: Identify use cases.

Identify use cases that the system needs to perform. Remember that use cases are sequence of actions that yields an observable result to actors. As an example, Figure 3.3 identifies the initial use cases.

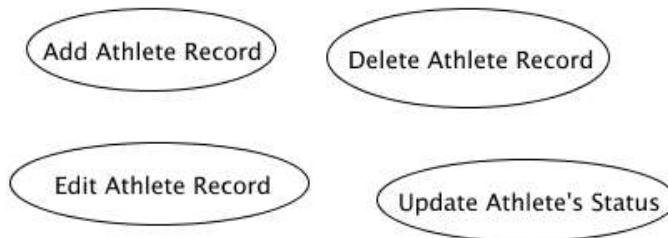


Figure 3.3 Club Membership Maintenance Use cases

The following use cases were identified.

- **Add Athlete Record**
- **Edit Athlete Record**
- **Delete Athlete Record**
- **Update Athlete's Status**

STEP 3: Associate use cases with actors.

Figure 3.4 shows the first iteration of the use case model. The identified use cases are distributed to the two actors.

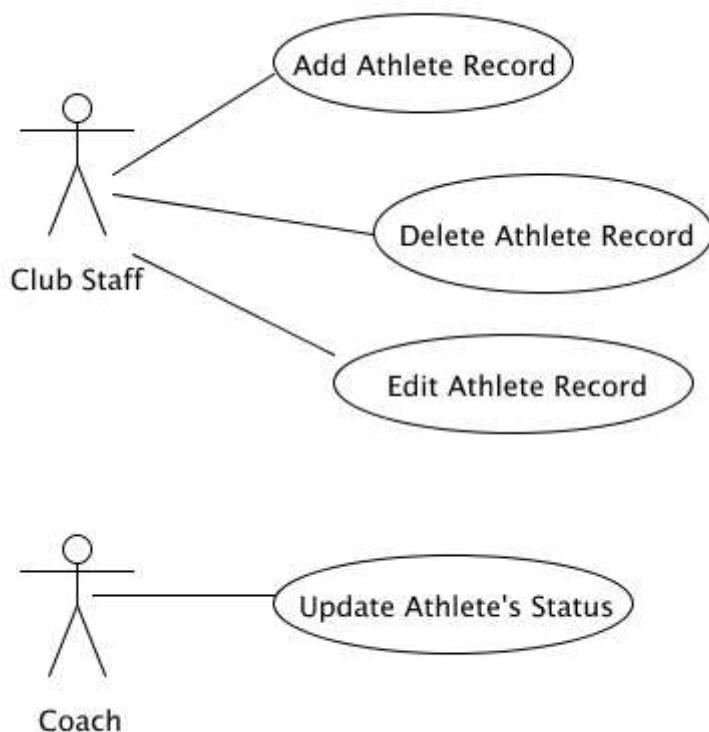


Figure 3.4 First Iteration of Club Membership Use Case Model

STEP 4: Refine and re-define the model.

This use case can be refined by using the enhanced notation. The second iteration of the use case model is seen in Figure 3.5. Optionally, number the use cases.

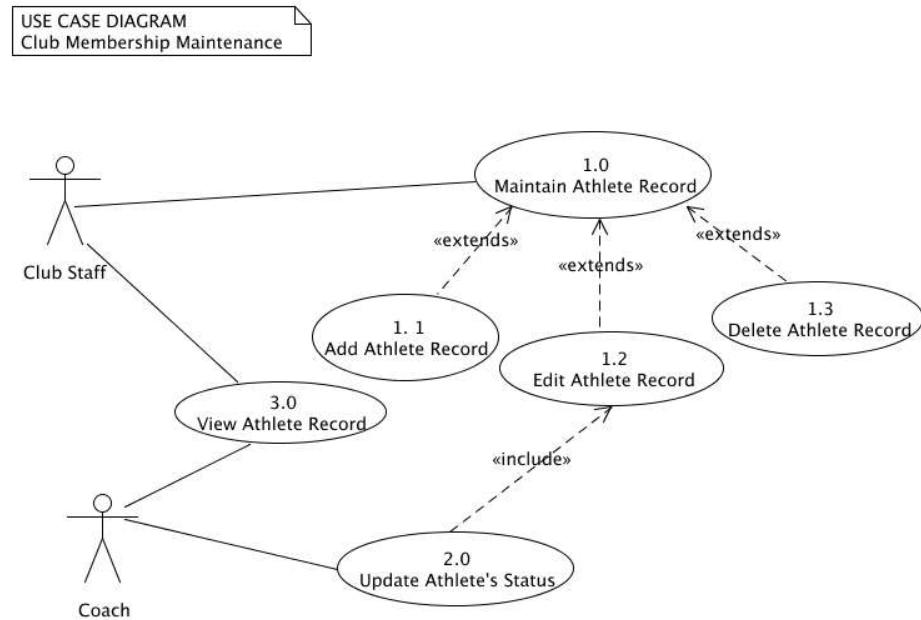


Figure 3.5 Second Iteration Club Membership Use Case Model

In the second iteration, elaboration is done by introducing the **Maintain Athlete Record** Use Case. This is being managed by the **club staff**. Stereotypes on associates are used to include or extend this use case.

The **Maintain Athlete Record** Use Case is extended to have the following option:

- 1.1 **Add Athlete Record**
- 1.2 **Edit Athlete Record**
- 1.3 **Remove Athlete Record**

Every time **Update Athlete's Record** Use Case is performed, an **Edit Athlete Record** is also performed.

Also, the another use case was added to specify that certain actors can view the athlete's record (**View Athlete Record** Use Case).

Modeling is an iterative process. The decision when to stop iterating is a subjective one and is done by the one modeling. To help in the decision-making, consider answering questions found in the *Requirements Model Validation Checklist*.

STEP 5: For each use case, write the use case specification.

For each use case found in the Use Case Model, the use case specification must be defined. The **use case specification** is a document where the internal details of the use case is specified. It can contain any of the listed sections below. However, the first five are the recommended sections.

1. **Name.** This is the name of the use case. It should be the same as the name found in the Use Case Model.
2. **Brief Description.** It describes the role and purpose of the use case using a

- few lines of sentences.
3. *Pre-conditions.* It specifies the conditions that exist before the use case starts.
 4. *Flow of Events.* This is the most important part of the requirements analysis part. These are events that describes what the use case is doing. Here, scenarios are identified.
 5. *Post-conditions.* It specifies the conditions that exist after the use case ends.
 6. *Relationships.* In this section, other use case that are associated with the current use case are specified here. Normally, they are the extend or include use cases.
 7. *Special Requirements.* It contains other requirements that cannot be specified using the diagram which is similar to the non-functional requirements.
 8. *Other Diagrams.* Additional diagrams to help in clarifying requirements are placed in this section such as screen prototypes, sketches of dialog of users with the system etc.

The **flow of events** contains the most important information derived from use case modeling work. It describes essentially what the use case is specifically doing; NOT how the system is design to perform. It provides a description of a sequence of actions that shows what the system needs to do in order to provide the service that an actor is expecting.

An instance of the functionality of the system is a scenario. It is also describe as one flow through a use case. The flow of events of a use case consists of a **basic flow** and several **alternative flows**. The basic flow is the common work flow that the use case follows; normally, we have one basic flow while the alternative flow addresses regular variants, odd cases and exceptional flows handling error situations.

There are two ways in expressing the flow of events- **textual** and **graphical**. To graphically illustrate the flow of events, one uses the Activity Diagram of UML. It is used similar to the Flowchart. However, it is used to model each scenario of the use case. It uses the following notation indicated in Table 4.

Notation	Name	Meaning
verb-noun	Activity State	It represents the performance of an activity or step within the work flow. Activities can be expressed using verb-noun phrases.
→	Transition	It shows what activity state follows after another. An arrow head indicates the direction of the state change.
◇	Decision	It is used to evaluate conditions similar to the decision symbol of the flow chart. It uses guarded conditions which determines the alternative transitions what will be made.
—	Synchronization Bars	It is used to show parallel sub-flows. It illustrates concurrent threads in the work flow.

Table 7 Activity Diagram Notation

Figure 3.6 is an example of an activity diagram. It illustrates how athletes are initially assigned to a squad. The **club staff** gets the filled up application form from the **athletes** and they prepare the mock try-out area. During the try-outs, the **selection committee** evaluates the performance of the **athletes**. If the athlete plays excellently, they are immediately assigned to the **Competing Squad**. Otherwise, they are assigned to the **Training Squad**. The **club staff** will keep the **athlete** records on the appropriate files.

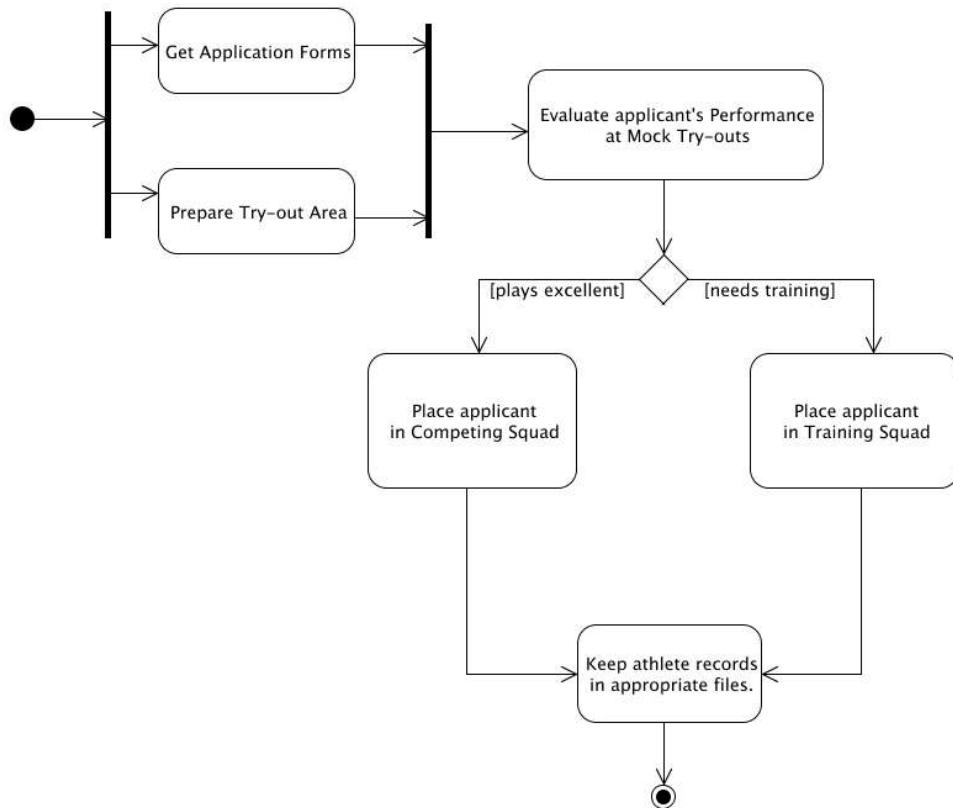


Figure 3.6 Initial Athlete Squad Assignment

The activity diagram can be modified as a the **swimlane diagram**. In this diagram, activities are aligned based on the actors who have responsibility for the specified activity. Figure 3.7 is the modified activity diagram where the actors responsible for the activity is drawn above the activity symbols.

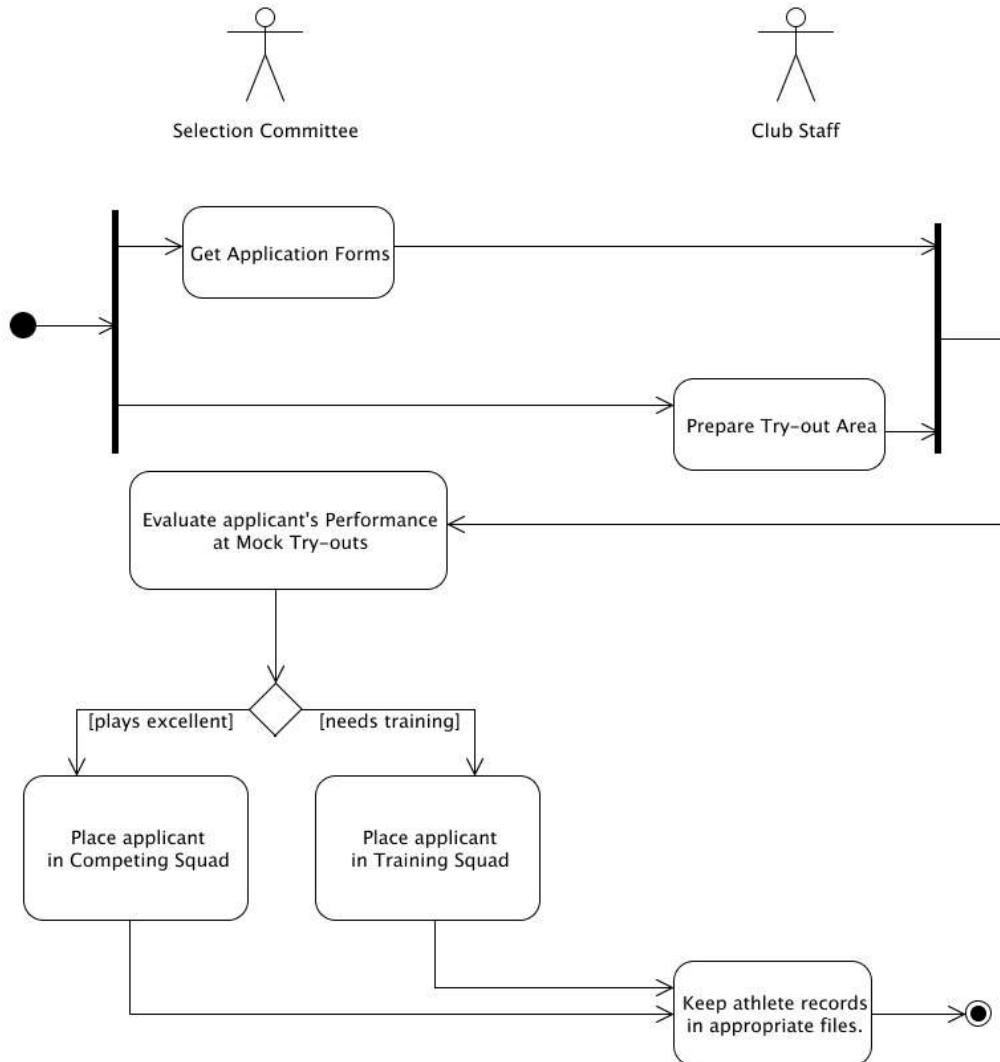


Figure 3.7 Initial Athlete Squad Assignment using Swimlane Diagram

To help software engineers in defining the flow of events, a list of guidelines is presented below.

1. To reinforce actor responsibility, start the description with "When the actor...".
1. Describe the data exchange between the actor and the use case.
2. Try not to describe the details of the user interface unless needed.
3. Answer ALL "what" questions. Test designers will use this text to identify test cases.
4. Avoid terminologies such as "For example,...", "process" and "information".
5. Describes when the use case starts and ends.

STEP 6: Refine and re-define the use case specifications.

Similar with developing the Use Case Model, one can refine and re-define the use case specification. It is also done iteratively. When to stop depends on the one modeling. One can answer the questions presented in the *Requirements Model Validation Checklist* to determine when to stop refining and redefining the use case specifications.

3.3.3 Requirements Model Validation Checklist

Like any work product being produced at any phase, validation is required. Listed below are the questions that guides software engineers to validate the requirements model. It is important to note at this time that the checklist serves as a guide. The software engineer may add or remove questions depending on the circumstances and needs of software development project.

Use Case Model Validation Checklist

1. Can we understand the Use Case Model?
2. Can we form a clear idea of the system's over-all functionality?
3. Can we see the relationship among the functions that the system needs to perform?
4. Did we address all functional requirements?
5. Does the use case model contain inconsistent behavior?
6. Can the use case model be divided into use case packages? Are they divided appropriately?

Actor Validation Checklist

1. Did we identify all actors?
2. Are all actors associated with at least one use case?
3. Does an actor specify a role? Should we merge or split actors?
4. Do the actors have intuitive and descriptive names?
5. Can both users and customers understand the names?

Use Case Validation Checklist

1. Are all use case associated with actors or other use cases?
2. Are use cases independent of one another?
3. Are there any use cases that exhibit similar behavior or flow of events
4. Are use cases given a unique, intuitive or explanatory names?
5. Do customers and users alike understand the names and descriptions of the use cases?

Use Case Specification Validation Checklist

1. Can we clearly see who wishes to perform a use case?
2. Is the purpose of the use case also clear?
3. Is the use case description properly and clearly defined? Can we understand it? Does it encapsulate what the use case is supposed to do?
4. Is it clear when the flow of events starts? When it ends?
5. Is it clear how the flow of events starts? How it ends?
6. Can we clearly understand the actor interactions and exchanges of information?
7. Are there any complex use cases?

Glossary Validation Checklist

1. Is the term clear or concise?
2. Are the terms used within the use case specification?
3. Are the terms used consistently within the system? Are there any synonyms?

3.4 Requirements Specifications

In the previous section, the requirements model was introduced and discussed. In this section, we will be focusing on the analysis model which provides a base model of what the software functions, features and constraints are. It is the most important work product to be developed in the requirements engineering phase because it serves as a basis for design and construction of the software. In this section, we will learn how the analysis model can be derived from the requirements model.

3.4.1 The Analysis Model

The Analysis Model is illustrated in Figure 3.8. It consists of two elements, particularly, **Object Model** and **Behavioral Model**.

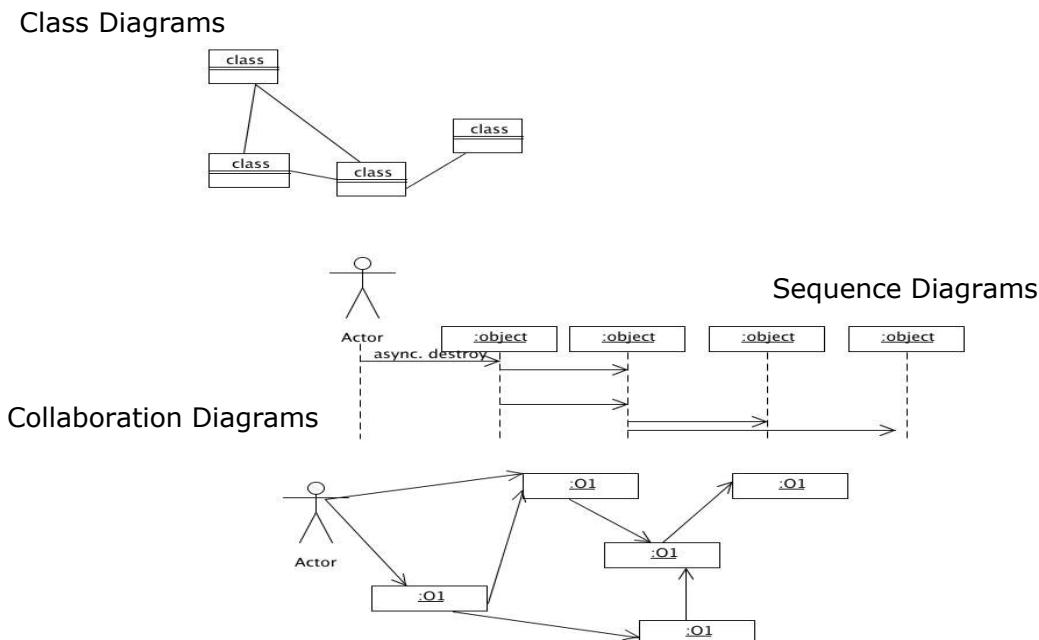


Figure 3.8 Analysis Model

To create the analysis model, the following input documents are needed.

- Requirements Model
- Software Architecture Document (Optional)

Object-Model

It is created using the Class Diagram of UML. It is the most important diagram to be developed because it serves as the major input for the design engineering phase. It consists of the **Analysis classes** which represent an early conceptual model for things in the system which have responsibilities and behavior. It is used to capture the first draft

of objects we want the system to support without making a decision on how much of them to support in hardware and how much in software. It rarely survives into the design engineering phase unchanged.

Behavioral Model

It is created using the Events or Interaction Diagrams of UML which consist of the Sequence and Collaboration Diagrams. It represents the dynamic aspects of the system. It shows the interaction and collaboration among analysis classes.

3.4.2 The Use Case Analysis Technique

It is a technique used in deriving the analysis model from the requirements model. Here, the Use Case Model is parsed to extract analysis classes. The structure and behavior of the analysis classes are then analyzed and documented. It should be able to capture the informational, functional and behavioral aspect of the system.

Developing the Analysis Model

Step 1: Validate the Use case Model.

The Use case model is validated to check the requirements. It sometimes captures additional information that may be needed to understand the internal behavior of the system. Also, one may find that some requirements are incorrect or not well-understood. In such a case, the original flow of events should be updated, i.e., one needs to go back in the requirements model analysis.

One may use a **whitebox approach** in describing and understanding the internal behavior of the system that needs to be performed. As an example, consider the following sentences. Which one is clear?

1. The selection committee gets application forms.
2. The selection committee retrieves the application forms of the applicants who are scheduled for the mock try-outs from the filing cabinet drawer with the title "Mock Try-out Applicants".

Step 2: For each use case, find the analysis classes.

Once the Use Case Model is validated, the candidate analysis classes are identified and described in a few sentences.

A **class** is a description of set of objects that share the same attributes, operations, relationships and semantics. It is an abstraction that emphasizes relevant characteristics and suppresses other characteristics. We say, an object is an instance of a class. A class is represented by a rectangular box with three compartments. The first compartment contains the name of the class. The second compartment contains a list of attributes. The last compartment contains a list of operations. Figure 3.9 shows an example of a class. In this example, the name of the class is **Athlete**. The attributes are shown in the second compartment which includes **id**, **lastname**, **firstname**, **mi**, **address**, **postalCode**, **telephone**, **bday**, **gender** and **status**. The operations are listed in the third compartment which includes **addAthlete()**, **editAthlete()**, **deleteAthlete()** and **updateStatus()**.

«entity»
Athlete
private int id
private String lastname
private String firstname
private String mi
private String address
private String postalCode
private String telephone
private Date bday
private char gender {M or F}
private String status
public void addAthlete(Athlete a)
public void editAthlete(Athlete a)
public void deleteAthlete(Athlete a)
public void updateStatus(Athlete a, String status)

Figure 3.9 Class Representation Sample

Three perspectives are used in identifying analysis classes. They are the **boundary** between the system and its actor, the **information** the system uses, and the **control logic** of the system. They offer a more robust model because they isolate those things most likely to change in a system. In UML, they are given stereotypes and are described in Table 7.

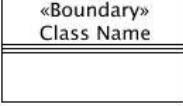
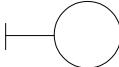
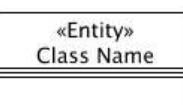
Stereotypes	Representation	Meaning
Boundary	 or 	<p>It is used to model interaction between a system's surroundings and its inner workings. It clarifies system boundaries and aid design by providing a good point of departure for identifying related services. It insulates external forces from internal mechanism and vice versa. It intermediates between the interface and something outside the system. There are three types: user-interface, system-interface and device-interface.</p>
Control	 or 	<p>Most of the functionality of the system is done by this class. It provides behavior that:</p> <ul style="list-style-type: none"> • Is surroundings independent. • Defines control logic and transactions within a use case. • Changes little if the internal structure or behavior of the entity classes changes. • Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes. • Is not performed in the same way every time it is activated. <p>It provides coordinating behavior of the system. It decouples boundary and entity classes from one another.</p>
Entity	 or 	<p>It represents stores of information in the system. It is used to hold and update information about some phenomenon, such as an event, a person or some real-life object. Its main responsibilities are to store and manage information in the system. It represents the key concepts of the system being developed.</p>

Table 8 Three Stereotypes in UML

Identifying Boundary Classes

Using the use case diagram, one boundary class exists for every actor/use case pair. An example is shown in Figure 3.10. The **AthleteRecordUI** is considered the boundary class between the **Club Staff** actor and the **Maintain Athlete Record** Use Case.

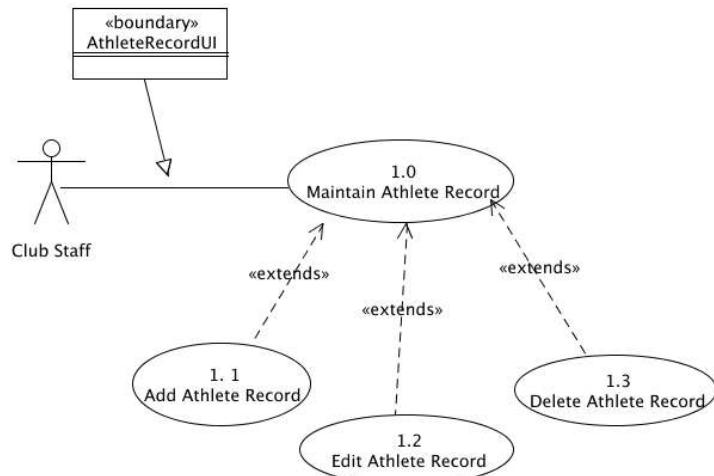


Figure 3.10 Club Membership Sample Boundary Classes

Identifying Control Classes

Using the use case diagram, one control class exists for every use case. An example is shown in Figure 3.11. Four control classes were identified, namely, **AthleteRecordMaintenance**, **AddAthlete**, **EditAthlete** and **DeleteAthlete**.

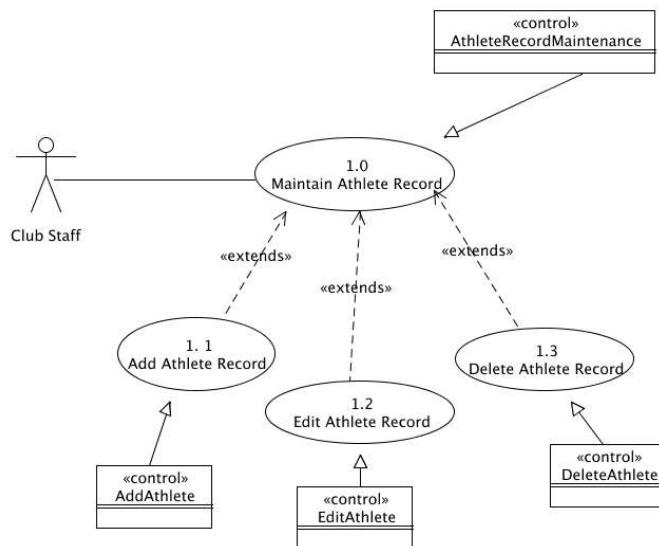


Figure 3.11 Club Membership Sample Control Classes

Identifying Entity Classes

Using the use case specifications, entity classes can be found by:

- using use case flow of events as input.
- getting key abstractions from the use cases.
- filtering nouns.

One can actually employ techniques used to discover entities in Entity-relationship Diagrams. An example of entity classes derived from the use cases *Add Athlete Record* are shown in Figure 3.12. They are **Athlete**, **Guardian** and **AthleteStatus** entity classes.

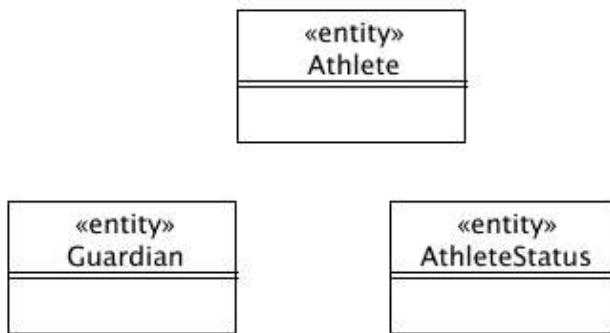


Figure 3.12 Club Membership Sample Entity Classes

Step 3: Model the behavior of the analysis classes.

To model the behavior of the analysis classes, one models the dynamic nature among the analysis classes. It uses the event or interaction diagrams of UML, specifically, Sequence and Collaboration Diagrams. When modeling the behavior of the system, object-level is used rather than class-level to allow for scenarios and events that use more than one instance of a class. Two basic modeling elements are used, namely, **objects** and **messages**.

Objects are instances of a class. They may be **Named Objects** or **Anonymous Objects**. Figure 3.13 shows examples of objects.

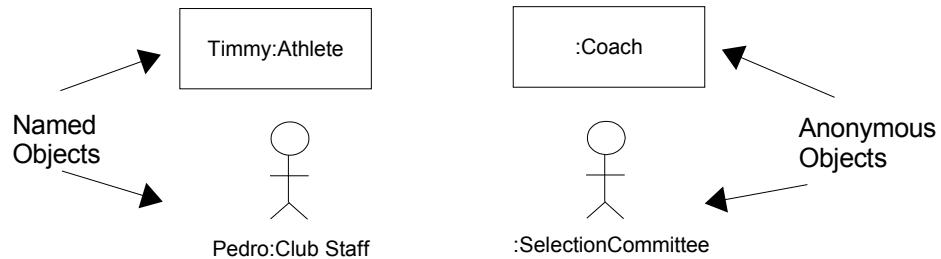


Figure 3.13 Sample Objects

Named Objects are identified by placing their names before the class name; it is separated by a colon (:). In the diagram, **Timmy:Athlete** is considered a named object. Anonymous objects do not have names; to represent them, we place the colon before the class name. **:Coach** is an example of an anonymous object.

Messages are a form of communication between objects. It has the following format:

* [condition] : action(list of parameters) : returnType

Table 8 gives the description of each of the components of the message. Only the action component is required.

Message Component	Meaning
*	It represents any number of repetition of the action.
[condition]	It represents a condition that must hold true before an action can be performed.
action(list of parameters)	It represents an action that must be performed by an object. It can optionally contain a list of parameters.
returnValueType	If an action returns a value, one needs to specify the type of the value.

Table 9 Message Components

Examples of messages are the following.

1. submitApplication()

This message means, an application is submitted.

2. [athleteIsHere = yes] playBasketball(athlete) : void

This message means, if athlete is here at the mock try-outs, he plays basketball.

3. * [for each athlete] isAthleteHere(athlete) : boolean

This message means, for each athlete, check if the athlete is here. Return YES if he is; otherwise, return NO.

Describing the Sequence Diagrams

The **Sequence Diagram** is used to model interactions between objects which map sequential interactions to object interactions. It shows explicitly the sequence of the messages being passed from one object to another. It is used to specify real-time interaction and complex scenarios. Similar with Use case Diagrams, it has a basic notation and enhanced notation. Figure 3.14 shows the basic notation of the sequence diagram.

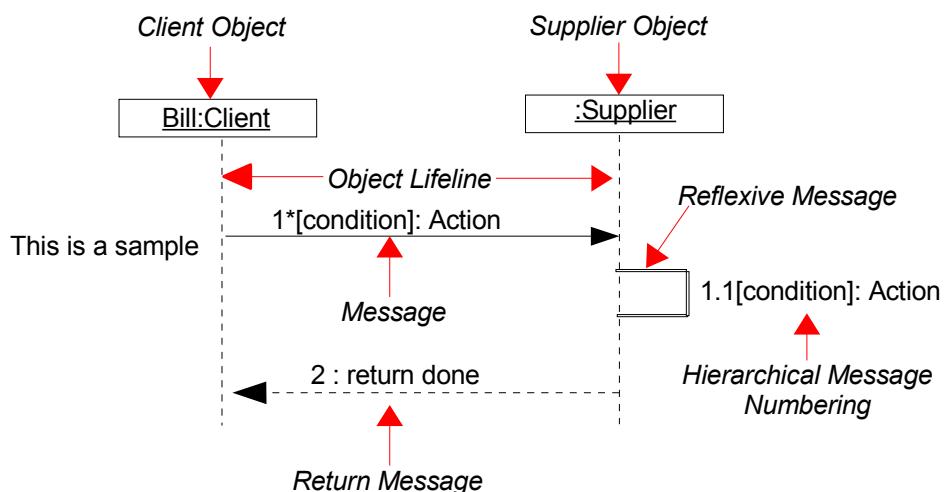


Figure 3.14 Sequence Diagram Basic Notation

Client objects are differentiated with Supplier Objects. **Client objects** are the objects sending messages. They are the ones asking for a service to perform the action specified in the message. **Supplier objects** are the recipients of the messages; they are requested to perform the action specified by the message. Objects are arranged

horizontally. Each object would have a **lifeline**. It represents the existence of the object at a particular time. A **message** is shown as a horizontal line with an arrow from the lifeline of one object to the lifeline of another object. A **reflexive message** shows an arrow that starts and ends on the same lifeline. Each arrow is labeled with a sequence number and the message. A **return message** is shown as a broken horizontal line with the message named *return* optionally followed by a return value.

Figure 3.15 shows the enhanced notation of the Sequence Diagram. It uses the concepts of object creation and termination, object activation and deactivation, and customized messages.

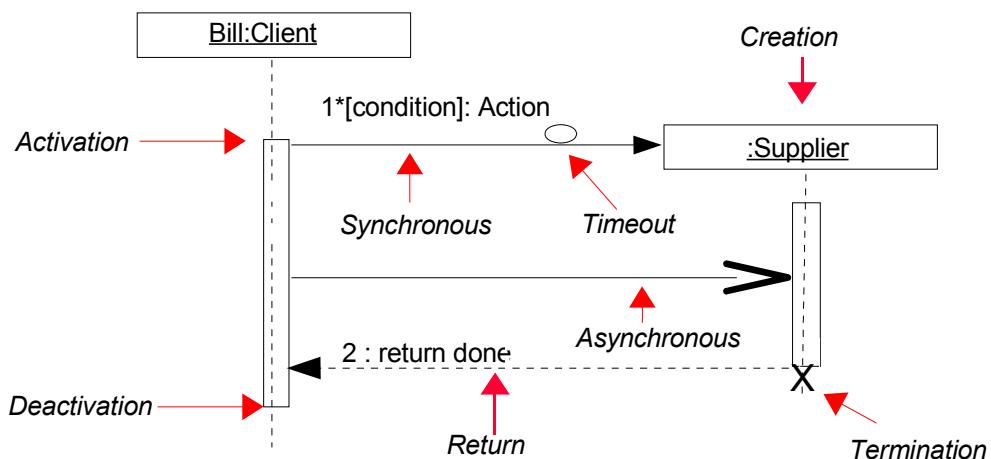


Figure 3.15 Sequence Diagram Enhanced Notation

Placing an object at the top of the diagram implies that the object exists before the scenario starts. If an object is created during the execution of the scenario, place the object relative to the point when it is created. To show that an object has been terminated, place an **X** at the point when the termination occurs.

To show the activation and deactivation of an object, we use the **focus of control**. It represents the relative time that the flow of control is focused in an object, thereby representing the time an object is directing messages. It is shown as a wide rectangle over the object's lifeline. The top of the rectangle indicates when an object becomes active; the bottom of the rectangle indicates when an object has become inactive.

Messages can be customized to be synchronous messages or asynchronous messages. **Synchronous messages** are used when the client object waits for the response of the supplier object before resuming execution. **Asynchronous messages** are used when client object does not need to wait for the supplier object's response. To differentiate between synchronous and asynchronous messages, take a look at the arrow head. If the arrow head is a blocked arrow head, it is synchronous. If the arrow head is a lined arrow head, it is asynchronous.

Timeout is used when the client abandons the message if the supplier object does not respond within a given period of time.

Developing the Sequence Diagram

We use the flow of events defined in a use case specification to help us define the **sequence diagrams**. As was mentioned, modeling of the behavior is done at the object-level rather than the class-level. This means that for every scenario defined in the flow of events we will have one sequence diagram. In the following example, The *successful adding of athlete record* of the **Add Athlete Record** Use Case will be used. In this case, the basic flow is considered.

1. Line up the participating analysis objects at the top of the diagram. As a rule, always place the control object between the boundary objects and entity objects. Remember that the control object serves as an intermediary between these two types of objects. Preferably, all actors are placed either at the beginning or end of the line. An example is shown in Figure 3.16. In this example, three analysis objects and one actor were identified. Specifically, they are **:AthleteRecordUI**, **:AddAthlete**, **:Athlete** and **:Club Staff**. Noticed that the **:AddAthlete**, which is a control class, is placed in between **:AthleteRecordUI** and **:Athlete** objects. It serves as the intermediary object between the boundary and entity. The **Club Staff** is the main actor of this scenario.



Figure 3.16 Line-up of Classes and Actors

2. Draw a vertical dashed line below each object to represent the object's lifeline. Figure 3.17 illustrates an example.

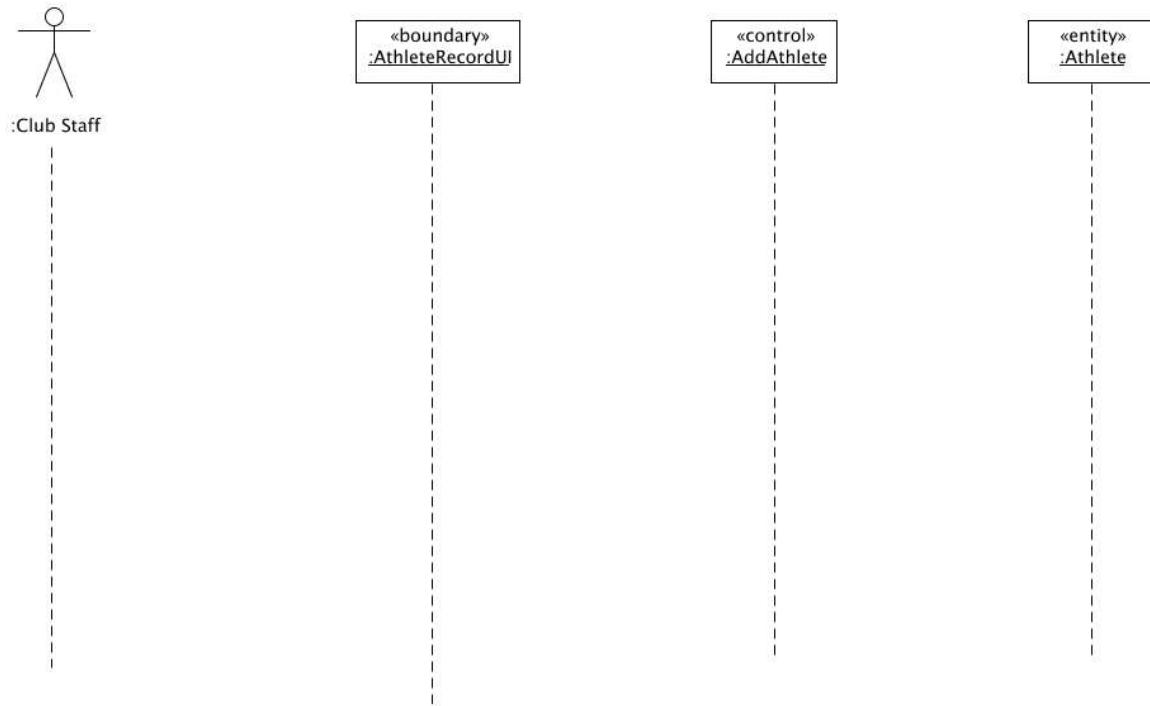


Figure 3.17 With Object's Lifeline

3. Draw the synchronous message from one object to another. As a rule, no messages are sent between the boundary object and entity object. They communicate indirectly through the control object. An example is shown in Figure 3.18.

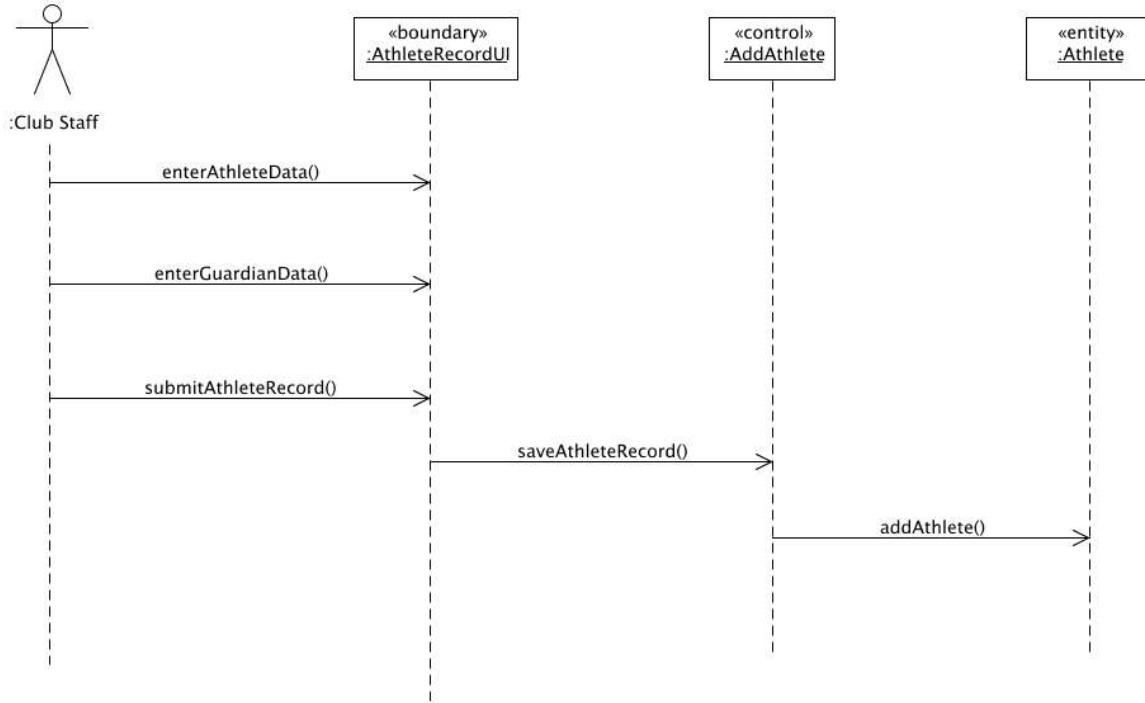


Figure 3.18 With Messages Sent to Objects

Here, the **:Club Staff** enters the athlete and guardian information through the **:AthleteRecordUI** boundary object by sending the messages `enterAthleteData()` and `enterGuardianData()`. To signify that the athlete records is to be saved, the club staff sends a message `submitAthleteRecord()` to the **:AddAthlete** control object. This class, in turn, sends a message, `addAthlete()`, to the **:Athlete** entity object to add the record.

4. Draw the return messages. An example is shown in Figure 3.19. The **:Athlete** entity object returns a status of **done** to the **:AddAthlete** control object. Similarly, the **:AddAthlete** control object returns a status of **done** to the **:AthleteRecordUI** boundary object.

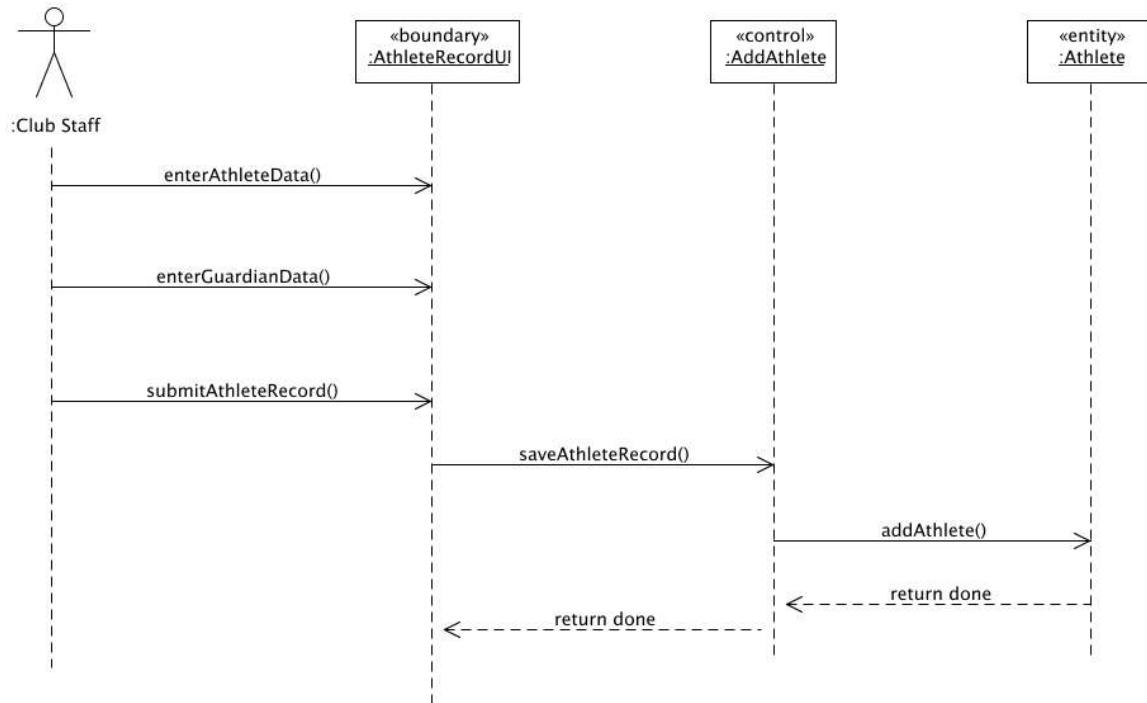


Figure 3.19 With Return Messages

5. Number the messages according to the chronological order of their invocation. Figure 3.20 shows an example.

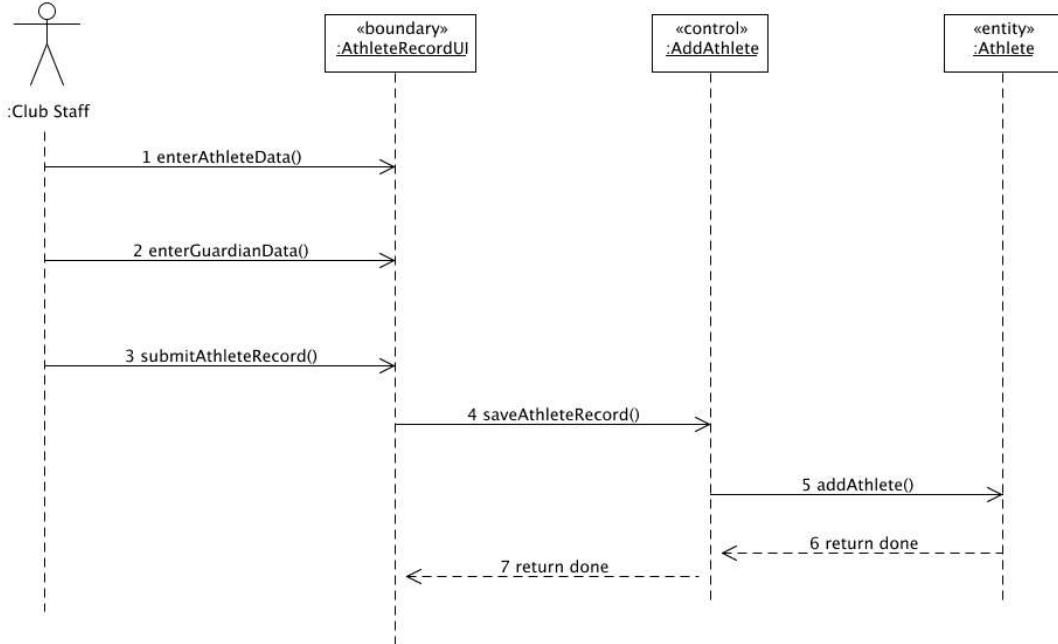


Figure 3.20 With Message Numbers

Numbering of the messages are as follows:

- 1. `enterAthleteData()`
- 2. `enterGuardianData()`
- 3. `submitAthleteRecord()`
- 4. `saveAthleteRecord()`
- 5. `addAthlete()`
- 6. `return done`
- 7. `return done`

6. Elaborate on the messages by specifying any of the following: repetition, conditions, list of parameters, and return type. Figure 3.21 shows an example.

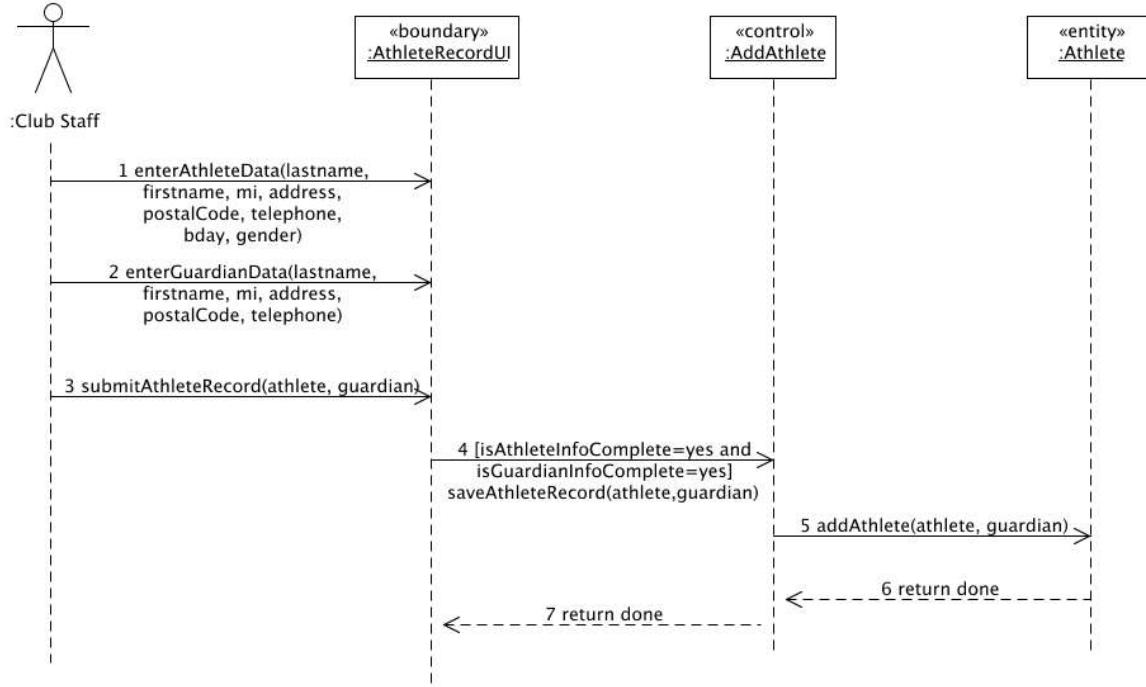


Figure 3.21 With Elaborated Messages

The elaborated messages are as follows:

1. The **enterAthleteData()** message was elaborated to include the following parameters: **lastname**, **firstname**, **mi**, **address**, **postalCode**, **telephone**, **bday**, and **gender**.
2. The **enterGuardianData()** message was elaborated to include the following parameters: **lastname**, **firstname**, **mi**, **address**, **postalCode** and **telephone**.
3. The **submitAthleteRecord()** message was elaborated to include the parameters, **athlete** and **guardian** object which represent the athlete and guardian information respectively.
4. The **saveAthleteRecord()** message was elaborated to include the parameters, **athlete** and **guardian** object which represent the athlete and guardian information respectively. Also, two conditions were identified that must be true in order for the message to be executed. They are **[isAthleteInfoComplete = yes]** and **[isGuardianInfoComplete = yes]**, which means that the athlete record is saved only if the athlete and guardian information are complete.
5. The **addAthlete()** message was elaborated to include the parameters, **athlete** and **guardian** object which represents the athlete and guardian information respectively.

7. Optionally, we can enhance the sequence diagram using the enhanced notation. Figure 3.22 shows an example.

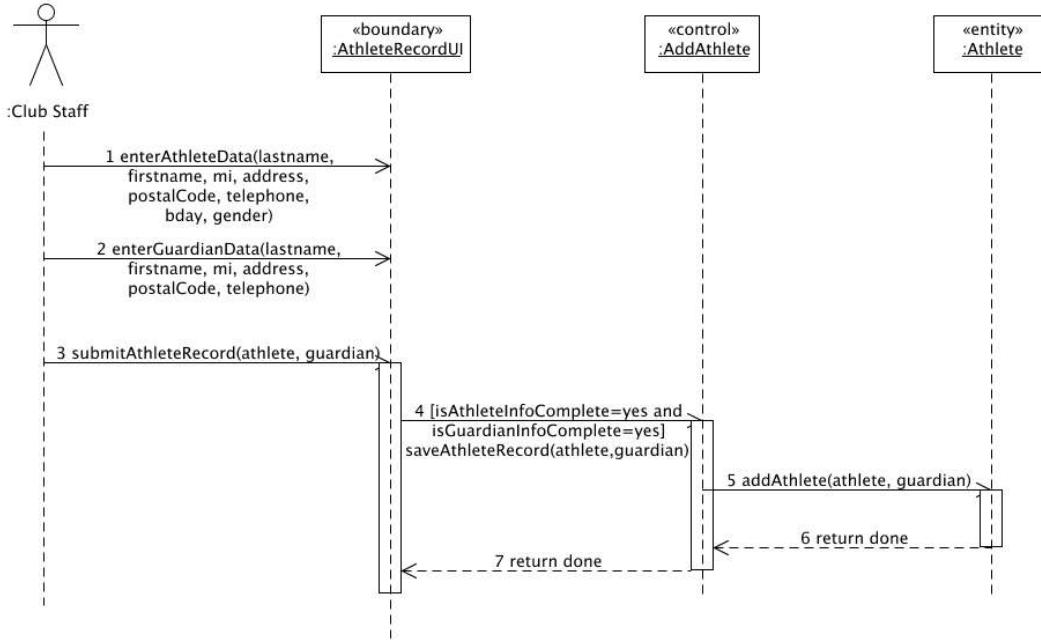


Figure 3.22 Enhanced Sequence Diagram of Adding Athlete Record

This may not be the only sequence diagram for the **Add Athlete Record** use case. Other scenario for this Use Case are *incomplete athlete information*, *incomplete guardian information*, *athlete record already exists* etc. For every scenario identified, a corresponding sequence diagram is created.

Describing the Collaboration Diagram

The **Collaboration Diagram** is used to model a pattern of interaction among objects. It models the objects participating in the interaction by their links to each other and the messages that they send to each other. It validates the class diagram by showing the need for each association. It models the recipient's message effectively which defines an interface to that object. Figure 3.23 shows the notation of the collaboration diagram.

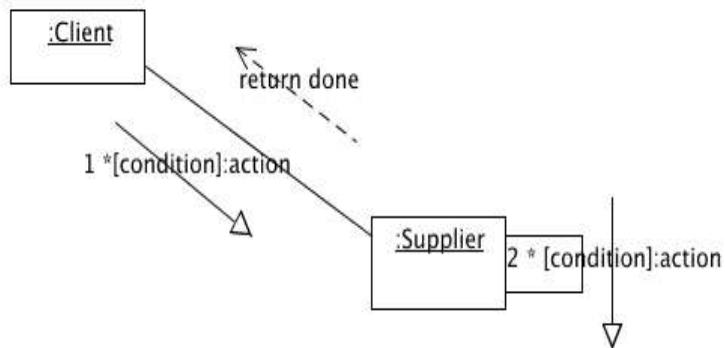


Figure 3.23 Collaboration Diagram

The notation used in the Collaboration Diagram is similar with the Sequence Diagram except that we don't model the messages in chronological order. We only show the message interaction.

Developing the Collaboration Diagrams

Collaboration diagrams can be derived from the sequence diagrams. Similar with sequence diagrams, object-level is used in modeling. Therefore, for every sequence diagram, one collaboration diagram is created. In the following example, the *successful adding of athlete record* scenario of the **Add Athlete** Use Case is used.

1. Draw the participating actors and objects. Figure 3.24 shows an example.



Figure 3.24 Actors and Objects

Again, the analysis objects **:AthleteRecordUI**, **:AddAthlete** and **:Athlete** are used with the **:Club Staff**.

2. If objects exchange messages as shown in their sequence diagram, draw a line connecting them which signifies an association. The lines represent links. For an example, Figure 3.25 depicts one. The **:Club Staff** exchanges messages with **:AthleteRecordUI**. The **:AthleteRecordUI** exchanges messages with **:AddAthlete**. Finally, the **:AddAthlete** exchanges messages with **:Athlete**.

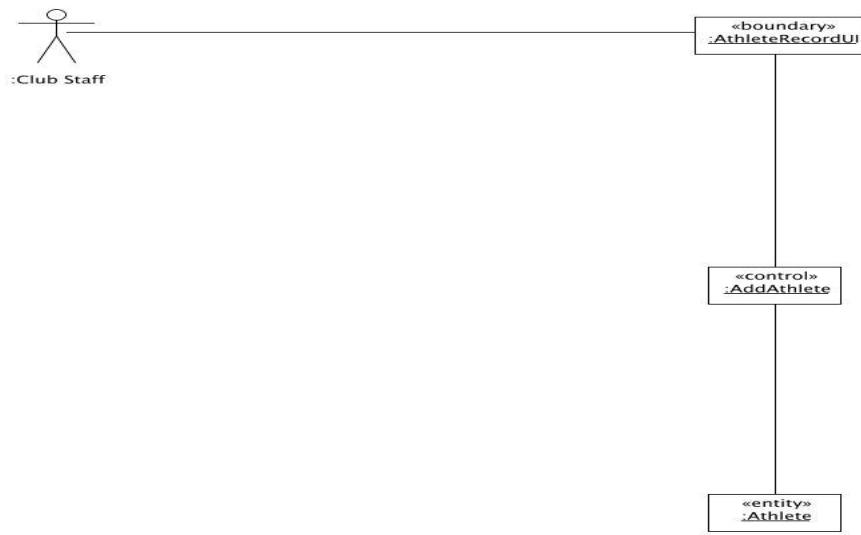


Figure 3.25 Actors and Objects With Links

3. Draw the messages on the association lines. An example is shown in Figure 3.26. Messages are represented as arrow lines with the message written within them. As an example, message number 1 is **enterAthleteData()** message with its corresponding parameters. Complete message definition is written.

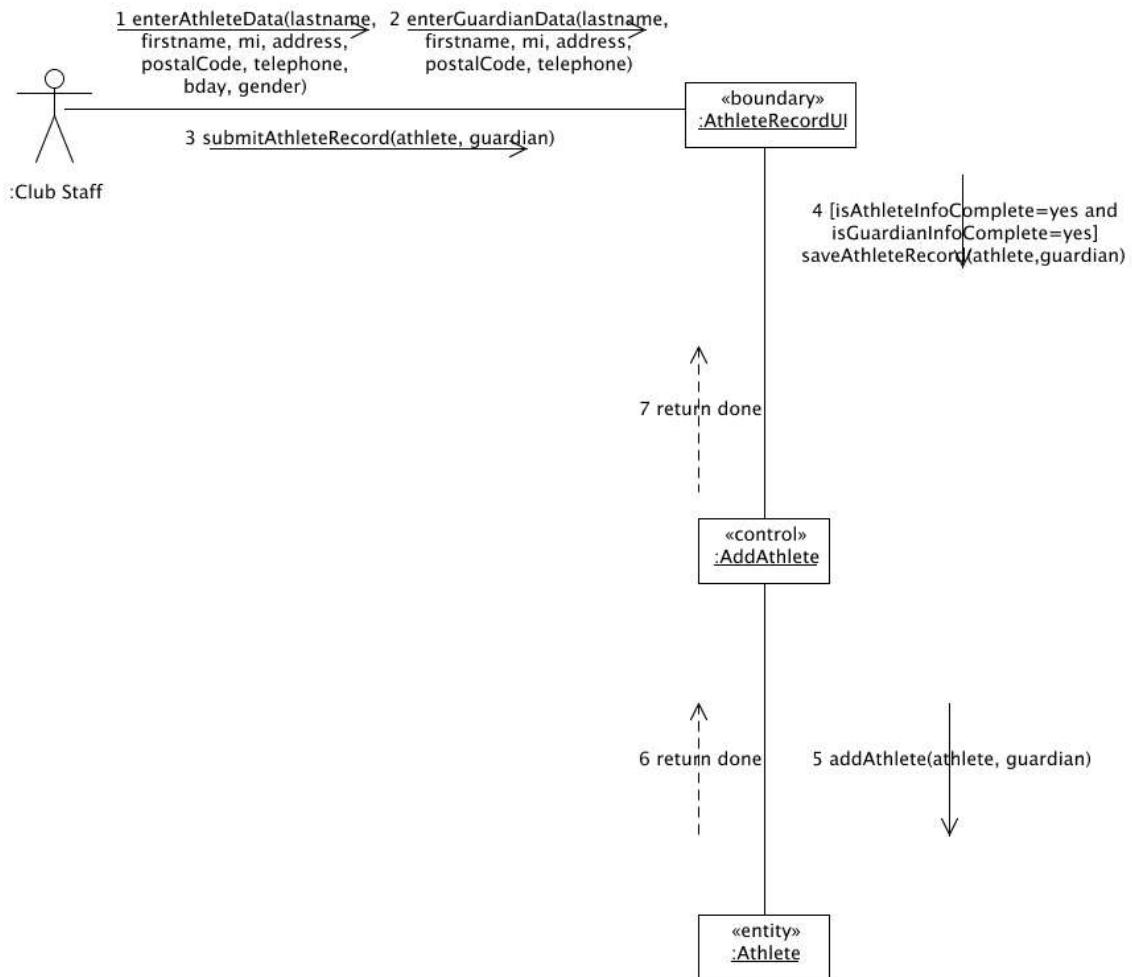


Figure 3.26 Collaboration Diagram of Add Athlete Record

This is the collaboration diagram of a *successful adding of athlete record* scenario of the **Add Athlete Record** Use Case. Similar with sequence diagrams, a collaboration diagram is created for every sequence diagram that was identified.

Step 4: For each resulting analysis classes, identify responsibilities.

A **responsibility** is something that an object provides to other objects or actors. It can be an action that the object can perform or a knowledge that the object maintains and provides to other objects. It can be derived from messages obtained from the events or interaction diagrams.

Using the collaboration diagrams, incoming messages are the responsibility of the class. To get the responsibilities of the analysis classes of the **Add Athlete Record** Use Case, we need to use its collaboration diagrams. Figure 3.27 shows the responsibilities of the classes for the **Add Athlete Record** Use Case using the collaboration diagrams. In this example, **:AthleteRecordUI**'s responsibilities are **enterAthleteData()**, **enterGuardianData()** and **submitAthleteRecord()**. **:AddAthlete**'s responsibility are **saveAthleteRecord()**. Finally, **:Athlete**'s responsibility is to **addAthlete()**, **editAthlete()**, **deleteAthlete()** and **updateStatus()**.

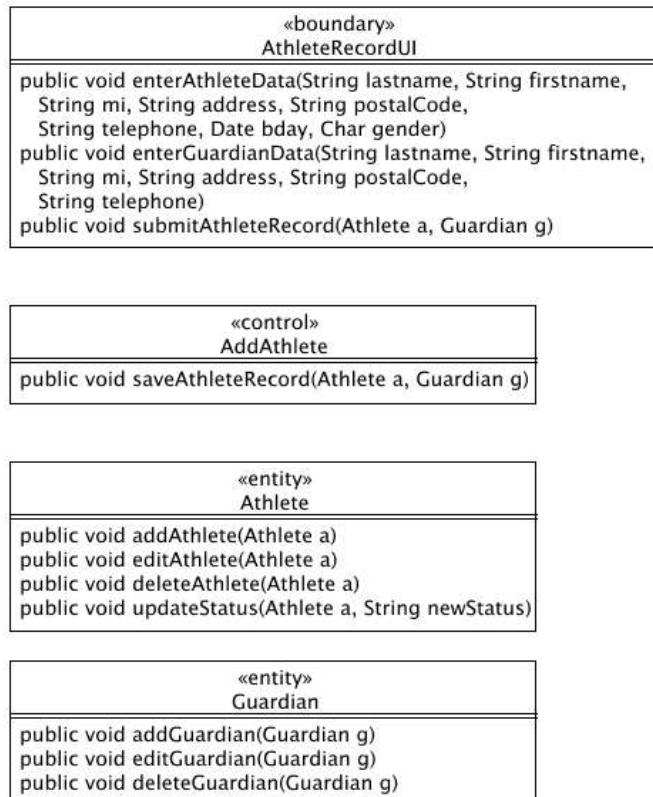


Figure 3.27 Responsibility Identification

To help identify the responsibility of the class, use the following guidelines.

1. Check that classes have consistent responsibility. If it seems that responsibilities are disjoint, split the object into two or more classes. Update the sequence and collaboration diagrams to reflect the changes.
2. Check the classes for similar responsibilities. If it seems that classes have the same responsibilities, combine them. Update the sequence and collaboration diagrams to reflect the changes.

3. Better distribution of responsibilities may be evident while working on another interaction diagram. In such a case, it is better and easier not to modify and update previous diagrams. Take the time to set the diagrams right, but don't get hung up trying to optimize the class interactions.
4. There is no problem if a class has only one responsibility, per se, but it should raise question on why it is needed. It is necessary to justify its existence.

You should compile all responsibilities seen in all collaboration diagrams into a single class definition.

Step 5: For each resulting analysis classes, identify attributes.

An **attribute** is an information that an object owns or known about itself. It is used to store information. Identifying attributes specifies the information that an analysis class is responsible for maintaining. They are specified using the following format:

attribute : data_type = default_value {constraints}

Table 9 gives the description of each component of the attribute specification. The attribute and data type are required.

Attribute Component	Meaning
attribute	It is the name of the attribute. When naming an attribute make sure that it describes the value it holds.
data_type	It specifies the type of the attribute. Preferably, use the data types defined in the Java Programming Language. You can even used the user-defined data types.
default_value	It is the default value assigned to the attribute when an object is instantiated.
constraints	It represents additional constraints applied to the allowable values of the attribute such as numeric ranges.

Table 10 Attribute Specifications

Below are some examples of specifying attributes.

1. InvoiceNumber : Numeric

Here, an attribute named **InvoiceNumber** contains a numeric value.

2. Temperature : Numeric { 4 decimal precisions }

Here, an attribute name **Temperature** contains a numeric value with a format of four decimal precisions.

3. SequenceNumber : Integer = 0 {assigned sequentially by the system}

Here, an attribute name **SequenceNumber** contains an integer with a default value of 0. Later on, it will be assigned a number sequentially assigned by the system.

To help in discovering and specifying attributes, one can use the guideline below.

1. Sources of possible attributes are system domain knowledge, requirements and glossary.
2. Attributes are used instead of classes when:
 - Only the value of the information, not its location, is important.
 - The information is uniquely owned by the object to which it belongs; no other objects refer to the information.
 - The information is accessed by operations which only get, set or perform simple transformations on the information; the information has no real behavior other than providing its value.
3. If the information has complex behavior, or is shared by two or more objects the information should be modeled as a separate class.
4. Attributes are domain dependent.
5. Attributes should support at least one use case.

In our example, the following attributes identified are shown in Figure 3.28. Similar to the previous step, all attributes seen in the collaboration diagrams should be compiled into a single analysis class. As an example, the **:Athlete** analysis class has the following attributes: **id**, **lastname**, **firstname**, **mi**, **address**, **postalCode**, **telephone**, **bday**, **gender**, and **status**.

<pre>«boundary» AthleteRecordUI public void enterAthleteData(String lastname, String firstname, String mi, String address, String postalCode, Date bday, Char gender, String status) public void enterGuardianData(String lastname, String firstname, String mi, String address, String postalCode, String telephone) private void submitAthleteRecord(Athlete a, Guardian g)</pre>	
<pre>«control» AddAthlete public void saveAthleteRecord(Athlete a, Guardian g)</pre>	
<pre>«entity» Athlete private int id private String lastName private String firstName private String mi private String address private String postalCode private String telephone private Date bday private char gender = {M or F} private String status public void addAthlete (Athlete a) public void editAthlete (Athlete a) public void deleteAthlete (Athlete a) public void updateStatus(Athlete a, String newStatus)</pre>	<pre>«entity» AthleteStatus private String code private String name</pre>
<pre>«entity» Guardian private int id private String lastName private String firstName private String mi private String address private String postalCode private String telephone private int athlete public void addGuardian(Guardian g) public void editGuardian(Guardian g) public void removeGuardian(Guardian g)</pre>	

Figure 3.28 Attribute Identification

Step 6: For each resulting analysis classes, identify associations.

An **association** represents the structural relationships between objects of different classes. It connects instances of two or more classes together for some duration. It may be:

1. **Unary Association** which is a relationship of objects of the same class.
2. **Binary association** which is a relationship of two objects of different classes.
3. **Ternary association** which is a relationship of three or more objects of different classes.

Figure 3.29 shows examples of the type of associations. The first example shows a unary association where a manager manages zero or many employee and an employee is being managed by one manager. The second example shows a binary association where an athlete is assigned to zero or one squad while a squad has many athletes.

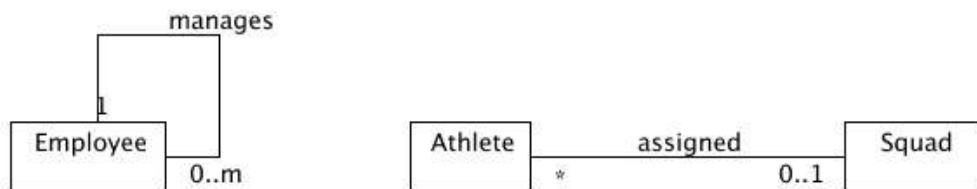


Figure 3.29 Types of Associations

Aggregation is a special form of association that models a whole-part relationship between an aggregate and its part. Some situations where an aggregation is appropriate are:

- An object is physically composed of other objects.
- An object is a logical collection of other objects.
- An object physically contains other objects.

Aggregation is modeled in Figure 3.30 which shows another way of modeling athlete and squad relationship. A squad is an aggregation of zero or many athletes. The end of the link that has a diamond shape symbol which represents “parts” or “components” of the aggregate or whole.



Figure 3.30 Aggregation

Multiplicity is the number of objects of the class that can be associated with one object of the other class. It is indicated by a text expression on the association line. It answers the following questions.

- Is the association optional or mandatory?
- What is the minimum and maximum number of instance that can be linked to one instance of an object?

It is possible that there can be multiple association between the same class. However, they should represent relationship among distinct objects of the same class.

Multiplicity specification are shown in Table 10.

Multiplicity	Description
_____	Unspecified. Use this if you don't know how many instances is associated with one object.
1	Exactly One. Use this if you know that exactly one object is associated with the other object.
0..*	Zero or Many. Use this if you know that zero or many objects are associated with the other object. This is also known as optional association.
*	
0..1	One or Many. Use this if you know that one or many objects are associated with the other object. This is also known as mandatory association
2-4	Specified Range. Use this if the number of objects has a specified range.
2,4..6	Multiple, Disjoint.

Table 11 Multiplicity Specifications

To help in the identification of the associations, the following guidelines can be used.

1. Start studying the links in the collaboration diagrams. They indicate that classes need to communicate.
2. Reflexive links do not need to be instances of unary association. An object can send a message to itself. Unary relationship is needed when two different objects of the same class need to communicate.
3. Focus only on associations needed to realize the use case. Do not add!
4. Give association names- either verb or roles.
5. Write a brief description of the association to indicate how the association is used, or what relationships the association represents.

In our running example, the following associations identified is shown in Figure 3.31.

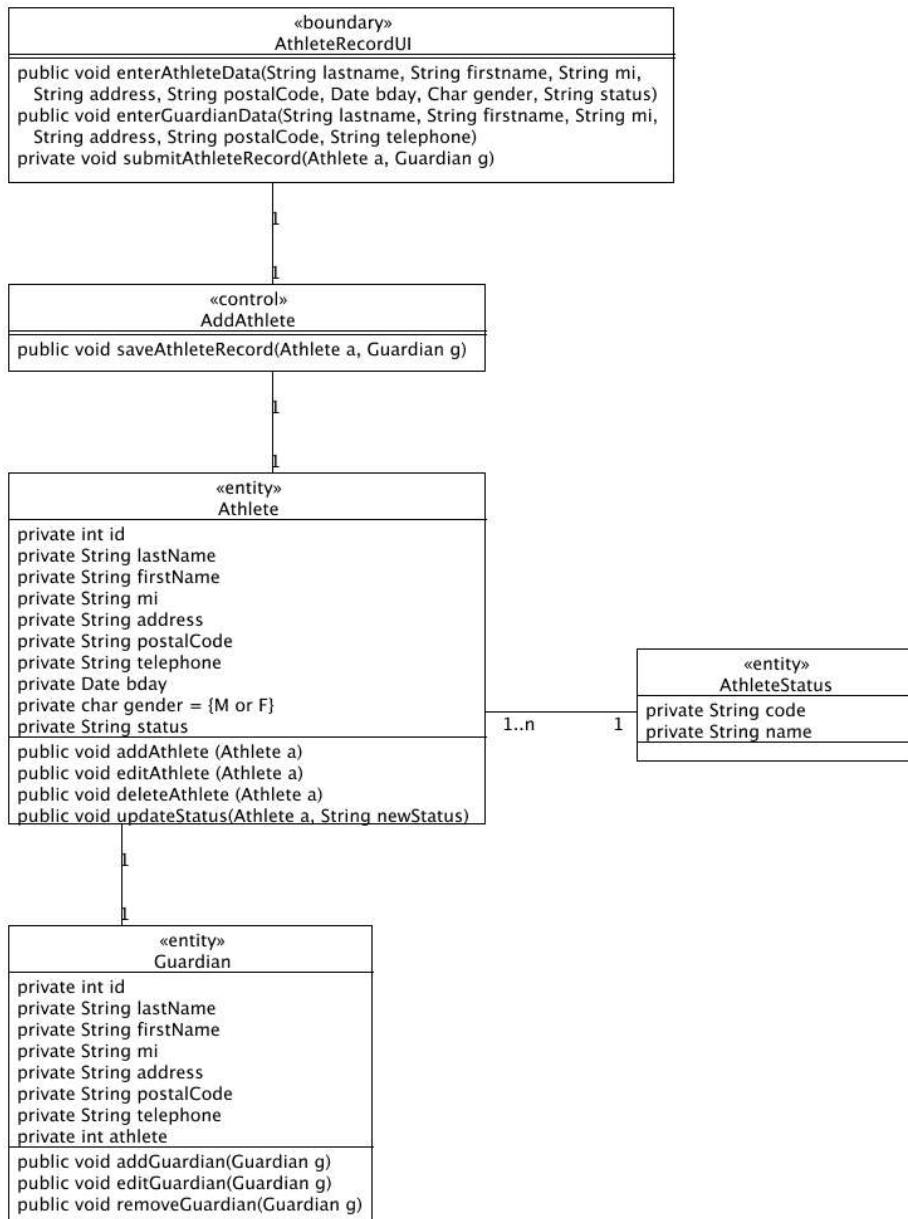


Figure 3.31 Association Identification

An **AthleteRecordUI** boundary class interacts with only one **AddAthlete** control class. An **AddAthlete** control class interacts with only one **Athlete** entity class. An **Athlete** has only one **Guardian** and one **AthleteStatus**. An **AthleteStatus** can be assigned to zero or many **Athletes**.

Step 7: Unify analysis classes in a class diagram.

Unifying analysis classes means ensuring that each analysis class represents a single well-defined concepts with non-overlapping responsibilities. It filters the analysis classes to ensure minimum number of new concepts have been created.

To help in unifying the analysis classes in a single class diagram, the following guidelines can be used.

1. The name of the analysis class should accurately capture the role played by the class in the system. It should be unique, and no two classes should have the same names.
2. Merge classes that have a similar behavior.
3. Merge entity classes that have similar attributes, even if their defined behavior is different; aggregate the behaviors of the merged classes.
4. When one updates a class, you should update any supplemental documentation, where necessary. Sometimes, an update on the original requirements may be required. BUT, this should be controlled as the requirements are the contract with the user/customer, and any changes must be verified and controlled.
5. Evaluate the result. Be sure to:
 - verify that the analysis classes meet the functional requirements made on the system
 - verify that the analysis classes and their relationships are consistent with the collaboration they support

If you noticed the unified class diagram, attribute and responsibilities are refined to give a clear definition of the classes. Remember that the analysis classes rarely survive after the design phase. The important thing at this point is that data (as represented by the attributes) and the operations (as represented by the responsibilities) are represented and distributed to classes. They will be restructured and redefined during the design phase. Use the validation checklist discussed in the succeeding section to check the analysis model. Figure 3.32 shows the unified class diagram of the **Club Membership Maintenance**.

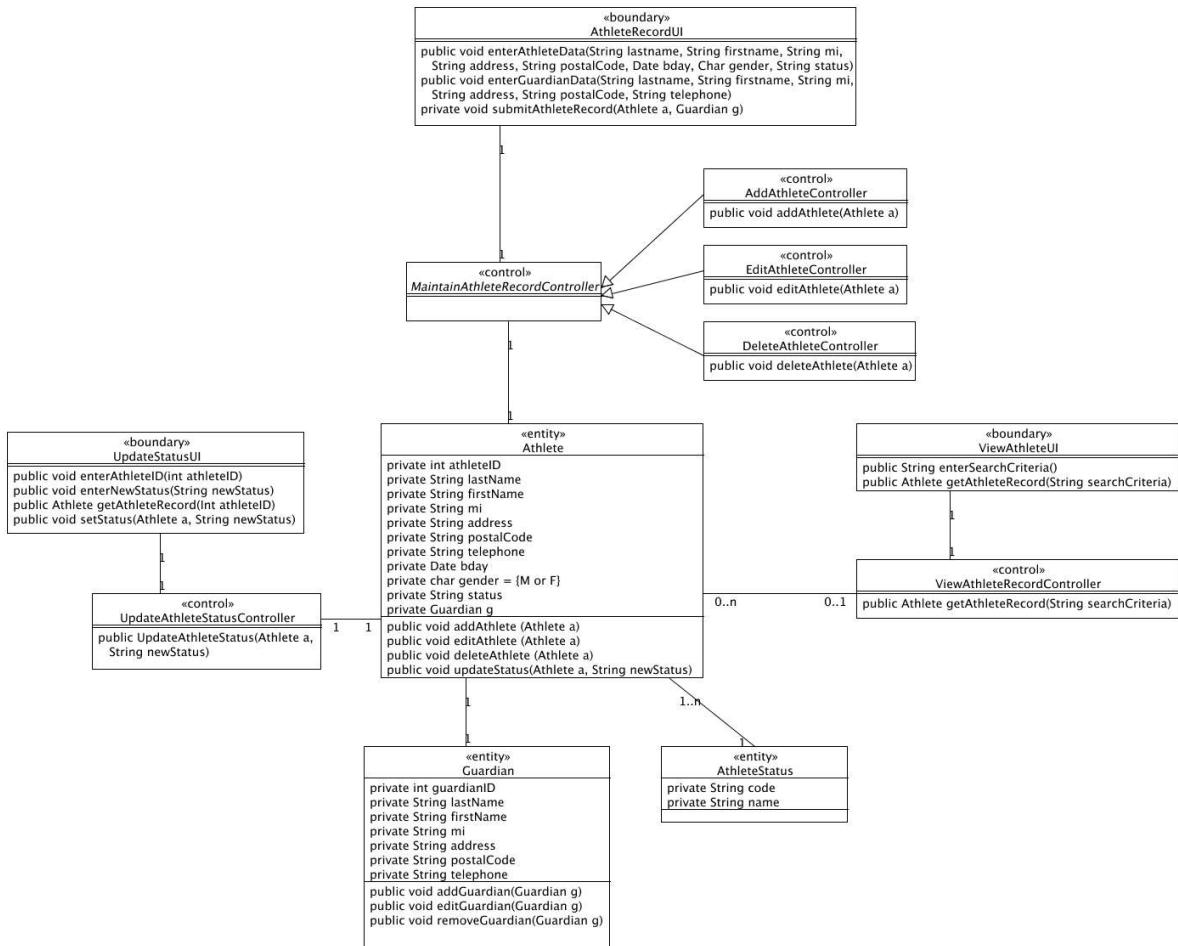


Figure 3.32 Club Membership Maintenance Class Diagram

3.4.3 Analysis Model Validation Checklist

Similar with the Requirements Model, a validation is required. The following guide questions can be used.

Object Model

1. Are all analysis classes identified, defined, and documented?
2. Is the number of classes reasonable?
3. Does the class have a name indicative of its role?
4. Is the class a well-defined abstraction?
5. Are all attributes and responsibilities defined for a class? Are they functionally coupled?
6. Is the class traceable to a requirement?

Behavioral Model

1. Are all scenarios been handled and modeled? (Including exceptional cases)
2. Are the interaction diagrams clear?
3. Are the relationships clear and consistent?
4. Are messages well-defined?
5. Are all diagrams traceable to the requirements?

3.5 Requirements Traceability Matrix (RTM)

The **Requirements Traceability Matrix (RTM)** is a tool for managing requirements that can be used not only in requirements engineering but throughout the software engineering process. It attempts to whitebox the development approach.

The **whitebox approach** refers to the development of system requirements without considering the technical implementation of those requirements. Similar with UML, this approach uses the concept of perspectives. There are three, namely, **conceptual**, **specification** and **implementation**. But in terms of requirements, we deal with the conceptual and specification.

The Conceptual Perspective deals with the “concepts in the domain”. The Specification Perspective deals with interfaces. As an example, concepts within the domain of the case study are athletes, coaches, squad and teams. Interfaces deal with how the concepts (system elements) interact with one another such as promoting or demoting an athlete.

Components of Requirements Traceability Matrix

There are many variations on the components that make up the RTM. Those listed in Table 11 are the recommended initial elements of the RTM.

RTM Component	Description
RTM ID	This is a unique identification number for a specific requirement. Choose a code that is easily identifiable.
Requirements	This is a structured sentence describing the requirements in a “shall” format, ie, the “the system shall...”, “the user shall...” or “selecting an item shall...”
Notes	These are additional notes of the requirements.
Requestor	This is the person who requested the requirement.
Date	This is the date and time the requirement was requested by the requestor.
Priority	This is the priority given to the requirements. One can use a numbered priority system, the Quality Function Deployment, or MoSCoW Technique.

Table 12 Initial RTM Components

While the above elements are the initial RTM, one can customize the matrix to tie the requirements with other work products or documents. This is really where the power of the RTM is reflected. One may add other elements. An example is shown in Table 12.

<i>Additional RTM Component</i>	<i>Description</i>
Relative to RTM ID	This is the RTM ID that specifies if the current RTM is directly or indirectly related to other requirements.
Statement of Work (SOW)	This is a description that relates the requirements directly back into a specific paragraph within the Statement of Work (SOW). A SOW is a document that specify the nature of the engagement of the development team with their clients. It lists the work products and criteria that will define the quality of the work products. It helps in guiding the development team in their approach in delivering the requirements.

Table 13 Additional RTM Components

The components and the use of the RTM should be adapted to the needs of the process, project and product. It grows as the project moves along. As software engineers, you can use all components, remove components or add components as you seem fit.

Consider an example of the RTM for the development of the system for the Ang Bulilit Liga: Club Membership Maintenance as shown in Table 13. The organization is use case-driven.

RTM ID	Requirement	Notes	Requestor	Date	Priority
Use Case 1.0	The system should be able to maintain an athlete's personal information.		RJCS	06/12/05	Should Have
Use Case 2.0	The system should be able to allow the coach to change the status of an athlete.		RJCS	06/12/05	Should Have
Use Case 3.0	The system should be able to view the athlete's personal information.		RJCS	06/12/05	Should Have
Use Case 1.1	The system should be able to add an athlete's record.		RJCS	06/13/05	Should Have
Use Case 1.2	The system should be able to edit an athlete's record.		RJCS	06/13/05	Should Have
Use Case 1.3	The system should be able to remove an athlete's record		RJCS	06/13/05	Should Have

Table 14 Sample Initial RTM for Club Membership Maintenance

3.6 Requirements Metrics

Measuring requirements usually focuses on three areas: process, product and resources. The requirements work products can be evaluated by looking first at the number of requirements. As the number grows, it gives an indication how large the software project is. Requirement size can be a good input for estimating software development effort. Also, as the software development progresses, the requirement size can be tracked down. As design and development occur, we would have a deeper understanding of the problem and solution which could lead to uncovering requirements that were not apparent during the requirements engineering process.

Similarly, one can measure the number of times that the requirements have changed. A large number of changes indicates some instability and uncertainty in our understanding of what the system should do or how it should behave. In such a case, a thorough understanding of the requirements should be done; possibly, iterate back to the requirements engineering phase.

Requirements can be used by the designers and testers. It can be used to determine if the requirements are ready to be turned over to them. The **Requirements Profile Test** is a technique employed to determine the readiness to turn over requirements to the designers or testers³.

For the designers, they are asked to rate each requirement on a scale of 1 to 5 based on a system as specified in Table 14.

³ Pfleeger, Software Engineering Theory and Practice, pp. 178-179

System Designer Rate	Description
1	It means that the designer understands this requirement completely, and that he has designed a requirement similar to this one from previous projects. He won't have any problems designing this one.
2	It means that there are aspects of the requirement that are new to the designer; however, they are not radically different from requirements that he has successfully designed in previous projects.
3	It means that there are aspects of the requirement that are very different from requirement the designer has designed before; however, he understands it and is confident that he can develop a good design.
4	It means that there are parts of this requirement that he does not understand; he is not confident that he can develop a good design.
5	It means he does not understand this requirement at all and he cannot develop a design for it.

Table 15 System Designer Scale Description

For testers, they are asked to rate each requirement on a scale of 1 to 5 based on the system as specified in Table 15.

System Tester Rate	Description
1	It means that he understands the requirement completely, and that he has tested similar requirements in previous projects; he is confident that he will have no trouble testing the code against the requirement.
2	It means that there are aspects of the requirement that is new to him; however, they are not radically different from requirements that he has successfully tested in previous projects.
3	It means that there are aspects of the requirement that are very different from requirements he has tested before; however, he understands it and is confident that he can test it.
4	It means that there are aspects of the requirement that he does not understand; he is not confident that he can devise a test to address this requirement.
5	It means that he does not understand this requirement at all, and he cannot develop a test to address it.

Table 16 System Tester Scale Description

If the result of the requirement's profile resulted with 1's and 2's, the requirements can be passed on the designers and testers. Otherwise, requirements need to be reassessed and rewritten. You need to iterate back to the requirement engineering phase.

Assessment is subjective. However, the scores can provide useful information that encourages you to improve the quality of the requirements before design proceeds.

3.7 Exercises

3.7.1 Creating the Requirements Model

1. Create the Requirements Model of the Coach Information System.
 - Develop the Use Case Diagram.
 - For each use case, develop the Use Case Specifications.
 - Create the Glossary.
2. Create the Requirements Model of the Squad & Team Maintenance System.
 - Develop the Use Case Diagram.
 - For each use case, develop the Use Case Specifications.
 - Create the Glossary.

3.7.2 Creating Analysis Model

1. Create the Analysis Model of the Coach Information System.
 - Develop the Sequence Diagram of every scenario defined in the Use Case Specification.
 - For each Sequence Diagram, create the collaboration diagrams.
 - Create the class diagram of the analysis classes.
2. Create the Analysis Model of the Squad & Team Maintenance System.
 - Develop the Sequence Diagram of every scenario defined in the Use Case Specification.
 - For each Sequence Diagram, create the collaboration diagrams.
 - Create the class diagram of the analysis classes.

3.8 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in this chapter. Particularly, they are:

1. Developing the Requirements Model
2. Developing the Analysis Model
3. Developing the Requirements Traceability Matrix

MAJOR WORK PRODUCTS:

1. The Requirements Model
 - Use Case Diagram
 - Use Case Specifications
2. The Analysis Model

- Class Diagram
- Sequence Diagram
- Collaboration Diagram

3. Action List

4. Requirements Traceability Matrix

4 Design Engineering

Unlike Requirements Engineering which creates model that focuses on the description of data, function and behavior, **design engineering** focuses on the creation of a representation or model that are concentrated on architecture of the software, data structures, interfaces and components that are necessary to implement the software. In this chapter, we will learn the design concepts and principles, data design, interface design, component-level design. We will also learn what elements of the RTM are modified and added to trace design work products with the requirements. The design metrics will also be discussed.

4.1 Design Engineering Concepts

Design Engineering is the most challenging phase of the software development project that every software engineer experience. Creativity is required for the formulation of a product or system where requirements (end-users and developers) and technical aspects are joined together. It is the last software engineering phase where models are created. It sets a stage where construction and testing of software is done. It uses various techniques and principles for the purpose of defining a device, process or system in sufficient detail to permit its physical realization. It produces a design model that translates the analysis model into a blueprint for constructing and testing the software.

The design process is an iterative process of refinement, i.e., from a higher level of abstraction to lower levels of abstraction. Each representation must be traced back to a specific requirement as documented in the RTM to ensure quality of design. Specifically, the design should:

- implement all explicitly stated requirements as modeled in the analysis model, at the same time, all implicit requirements desired by the end-user and developers;
- be readable and understandable by those who will generate the code and test the software; and
- provide an over-all illustration of the software from the perspective of the data, function and behavior from an implementation perspective.

Pressman suggests a list of quality guidelines as enumerated below⁴.

1. The design should have a recognizable architecture that has been created using known architectural styles or patterns, that consists of good design components, and that can be created in an evolutionary manner.
2. The design should be modular that are logically partitioned into subsystems and elements.
3. The design should consists of unique data representations, architecture, interfaces, and components.
4. The design of the data structure should lead to the design of the appropriate classes that are derived from known data patterns.
5. The design of the components should have independent functional characteristics.

⁴ Pressman, Software Engineering A Practitioner's Approach, page262-263

6. The design should have interfaces that reduces the complexity of the links between components and the environment.
7. The design is derived from using a method repetitively to obtain information during the requirements engineering phase.
8. The design should use a notation that convey its meaning.

The above guidelines encourages good design through the application of fundamental design principles, systematic methodology and thorough review.

4.1.1 Design Concepts

Design concepts provides the software engineer a foundation from which design methods can be applied. It provides a necessary framework of creating the design work products **right**.

Abstraction

When designing a modular system, many level of abstractions are used. As software engineers, we define different levels of abstractions as we design the blueprint of the software. At the higher level of abstraction, we state the solution using broad terms. When we iterate to much lower level of abstractions, a detailed description of the solution is defined. Two types of abstractions are created, namely, data abstractions and procedural abstractions.

Data abstractions refer to the named collection of data that describes the information required by the system.

Procedural abstractions refer to the sequences of commands or instructions that have a specific limited actions.

Modularity

Modularity is the characteristic of a software that allows its development and maintenance to be manageable. The software is decomposed into pieces called **modules**. They are named and addressable components when linked and working together satisfy a requirement. Design is modularized so that we can easily develop a plan for software increments, accommodate changes easily, test and debug effectively, and maintain the system with little side-effects. In object-oriented approach, they are called **classes**.

Modularity leads to information hiding. **Information hiding** means hiding the details (attributes and operations) of the module or class from all others that have no need for such information. Modules and classes communicate through interfaces, thus, enforces access constraints on data and procedural details. This limits or controls the propagation of changes and errors when modifications are done to the modules or classes.

Modularity also encourages functional independence. **Functional Independence** is the characteristic of a module or class to address a specific function as defined by the requirements. They are achieved by defining modules that do a single task or function and have just enough interaction with other modules. Good design uses two important criteria: coupling and cohesion.

Coupling is the degree of interconnectedness between design objects as represented by

the number of links an object has and by the degree of interaction it has with other objects. For object-oriented design, two types of coupling are used.

1. **Interaction Coupling** is the measure of the number of message types an object sends to another object, and the number of parameters passed with these message types. Good interaction coupling is kept to a minimum to avoid possible change ripples through the interface.
2. **Inheritance Coupling** is the degree to which a subclass actually needs the features (attributes and operations) it inherits from its base class. One minimizes the number of attributes and operations that are unnecessarily inherited.

Cohesion is the measure to which an element (attribute, operation, or class within a package) contributes to a single purpose. For object-oriented design, three types of cohesion are used.

1. **Operation Cohesion** is the degree to which an operation focuses on a single functional requirement. Good design produces highly cohesive operations.
2. **Class Cohesion** is the degree to which a class is focused on a single requirement.
3. **Specialization Cohesion** address the semantic cohesion of inheritance. Inheritance definition should reflect true inheritance rather than sharing syntactic structure.

Refinement

Refinement is also known as the **process of elaboration**. Abstraction complements refinement as they enable a software engineer to specify the behavior and data of a class or module yet suppressing low levels of detail. It helps the software engineer in creating a complete design model as the design evolves. Refinement helps the software engineer to uncover the details as the development progresses.

Refactoring

Refactoring is a technique that simplifies the design of the component without changing its function and behavior. It is a process of changing the software so that the external behavior remains the same and the internal structures are improved. During refactoring, the design model is checked for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures or any other design failures. These are corrected to produce a better design.

4.1.2 The Design Model

The work product of the design engineering phase is the design model which consists of the architectural design, data design, interface design and component-level design.

Architectural Design

This refers to the overall structure of the software. It includes the ways in which it provides conceptual integrity for a system. It represents layers, subsystems and components. It is modeled using the package diagram of UML.

Data Design

This refers to the design and organization of data. Entity classes that are defined in the requirements engineering phase are refined to create the logical database design. Persistent classes are developed to access data from a database server. It is modeled using the class diagram.

Interface Design

This refers to the design of the interaction of the system with its environment, particularly, the human-interaction aspects. It includes the dialog and screen designs. Report and form layouts are included. It uses the class diagram and state transition diagrams.

Component-level Design

This refers to the design of the internal behavior of each classes. Of particular interest are the control classes. This is the most important design because the functional requirements are represented by these classes. It uses the class diagrams and component diagrams.

Deployment-level Design

This refers to the design of the how the software will be deployed for operational use. Software functionality, subsystems and components are distributed to the physical environment that will support the software. The deployment diagram will be used to represent this model.

4.2 Software Architecture

There is no general agreed definition for the term software architecture. It is interpreted differently depending upon the context. Some would describe it in terms of class structures and the ways in which they are grouped together. Others used it to describe the overall organization of a system into subsystem. Buschmann et al.⁵ defined it as follows:

A **software architecture** is a description of the sub-systems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of software design activity.

It is this definition that is being followed by this section. Software architecture is the layered structure of software components and the manner in which these components interact, and the structure of the data that are used by these components. It involves making decisions on how the software is built, and normally, controls the iterative and incremental development of the software.

Defining the architecture is important for several reasons. First, representation of software architecture enables communication between stakeholders (end-users and developers). Second, it enables design decisions that will have a significant effect on all software engineering work products for the software's interoperability. Lastly, it gives an intellectual view of how the system is structured and how its components work together.

It is modeled using the Package Diagram of UML. A **package** is a model element that can contain other elements. It is a mechanism used to organize design elements and allows modularization of the software components.

4.2.1 Describing the Package Diagram

The **Package Diagram** shows the breakdown of larger systems into logical groupings of smaller subsystems. It shows groupings of classes and dependencies among them. A dependency exists between two elements if changes to the definition of one element may cause changes to other elements. Figure 4.1 shows the basic notation of the package diagram.

⁵ Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. and Stal, M., *Pattern Oriented Software Architecture Volume 1*, Chichester:John Wiley, 1996.

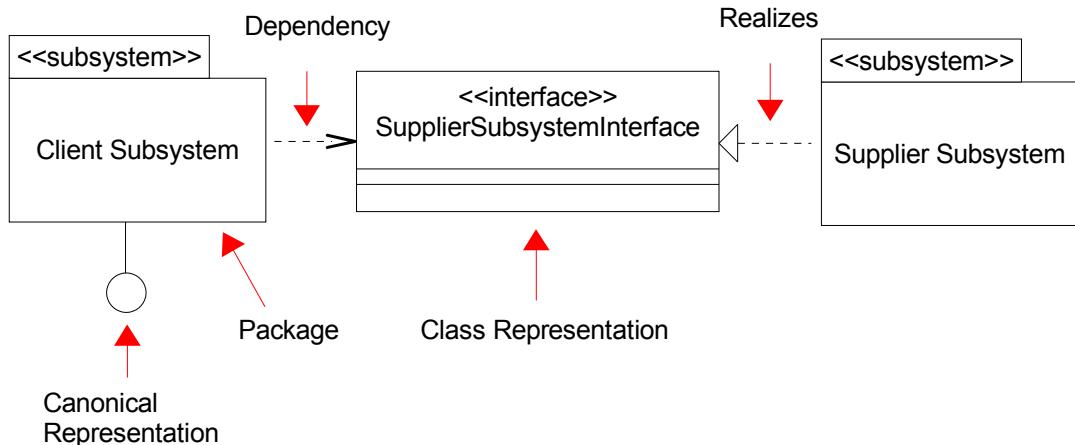


Figure 4.1 Package Diagram Basic Notation

Packages are represented as **folders**. If a package is represented as a subsystem, the subsystem stereotype is indicated (<<subsystem>>). The **dependencies** are represented as broken arrow lines with the arrow head as a line. As in the example, Client Subsystem is dependent on the Supplier Subsystem. The packages are realized by **interfaces**. The interface can be canonical as represented by the circle as depicted by the Client Subsystem Interface, or it can be a class definition as depicted by the Supplier Subsystem Interface (stereotype is <<interface>>). Noticed that the relationship of the package is illustrated with a broken line with a transparent blocked arrow head.

4.2.2 Subsystems and Interfaces

A **subsystem** is a combination of a package (it can contain other model elements) and a class (it has behavior that interacts with other model elements). It can be used to partition the system into parts which can be independently ordered, configured and delivered. It allows components to be developed independently as long as the interface does not change. It can be deployed across a set of distributed computational nodes and changed without breaking other parts of the system. It also provides restricted security control over key resources.

It typically groups together elements of the system that share common properties. It encapsulates an understandable set of responsibilities in order to ensure that it has integrity and can be maintained. As an example, one subsystem may contain human-computer interfaces that deals with how people interact with the system; another subsystem may deal with data management.

The advantages of defining subsystems are as follows:

- It allows for the development of smaller units.
- It maximizes the reusability of components.
- It allows developers handle complexity.

- It supports and eases maintainability.
- It supports portability.

Each subsystem should have a clear boundary and fully defined interfaces with other subsystems.

Interfaces define a set of operations which are realized by a subsystem. It allows the separation of the declaration of behavior from the realization of the behavior. It serves as a contract to help in the independent development of the components by the development team, and ensures that the components can work together. The specification of the interface defines the precise nature of the subsystem's interaction with the rest of the system but does not describe its internal structure.

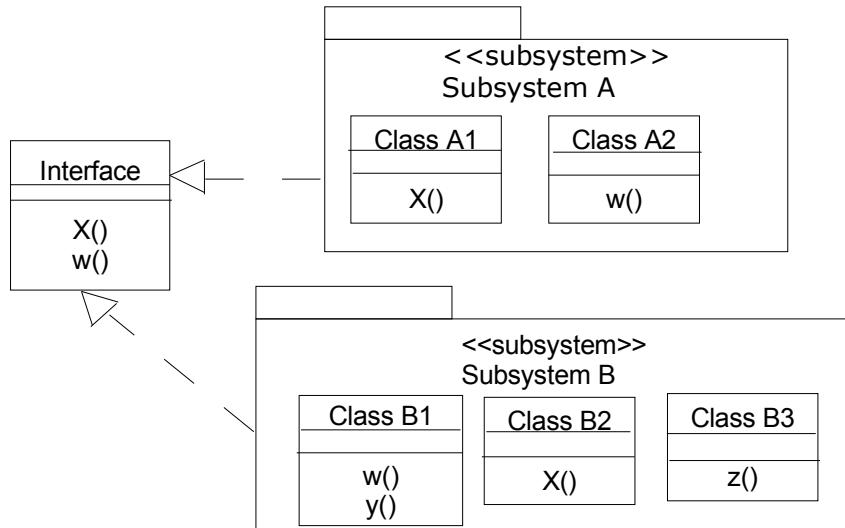
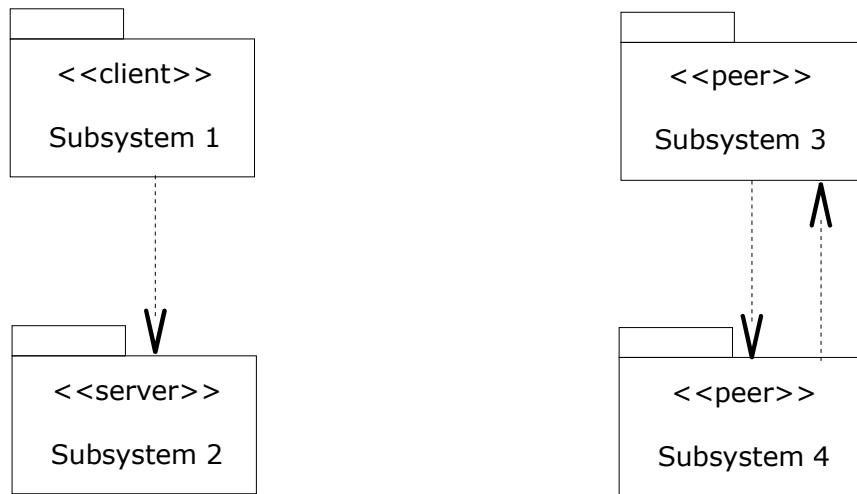


Figure 4.2 Subsystems and Interfaces

As shown in Figure 4.2, an interface is realized by one or more subsystems. This is the secret of having plug-and-play components. The implementation of the subsystem can change without drastically changing other subsystems as long as the interface definition does not change.

Each subsystem provides services for other subsystems. There are two styles of communication subsystems use; they are known as **client-server** and **peer-to-peer** communication. They are shown in Figure 4.3.



The server subsystem does not depend on the client subsystem. Changes to the client interface do not affect it.

Each peer subsystem depends on the other. They are affected by changes in each other's interface.

Figure 4.3 Communication Styles of Subsystems

In a client-server communication, the client-subsystem needs to know the interface of the server subsystems. The communication is only in one direction. The client-subsystem request services from the server-subsystem; not vice versa. In a peer-to-peer communication, each subsystem knows the interface of the other subsystem. In this case, coupling is tighter. The communication is two way since either peer subsystem may request services from others.

In general, client-server communication is simpler to implement and maintain since they are less tightly coupled than the peer-to-peer communication.

There are two approaches in dividing software into subsystems. They are known as layering and partitioning. **Layering** focuses on subsystems as represented by different levels of abstraction or layers of services while **partitioning** focuses on the different aspects of the functionality of the system as a whole. In practice, both approaches are used such that some subsystems are defined in layers others as partitions.

Layered architectures are commonly used for high-level structures for a system. The general structure is depicted in Figure 4.4.

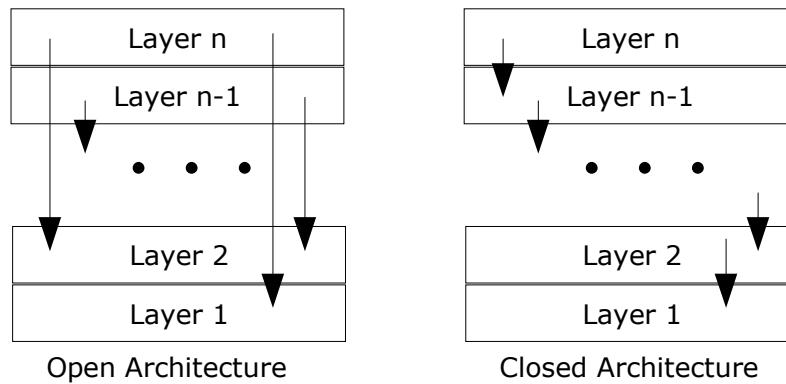


Figure 4.4 General Structures of Layered Architecture

Each layer of the architecture represents one or more subsystems which may be differentiated from one another by differing levels of abstraction or by a different focus of their functionality. The top layer request services from the layer below it. They, in turn, may use services of the next layer. In an **open layered architecture**, subsystems may request services from any layers below them. For closed layered architecture, layers request services from the layer directly below them. They cannot skip layers.

Closed layered architecture minimizes dependencies between layers and reduces the impact of change to the interface of any one layer. Open layered architectures allows for the development of more compact codes since services of all lower level layers can be accessed directly by any layer above them without the need for extra program codes to pass messages through each intervening layer. However, it breaks encapsulation of the layers, increases the dependencies between layers and increases the difficulty caused when a layer needs to be modified.

Some layers within a layered architecture may have to be decomposed because of their complexity. Partitioning is required to define these decomposed components. As an illustration, consider the Figure 4.5.

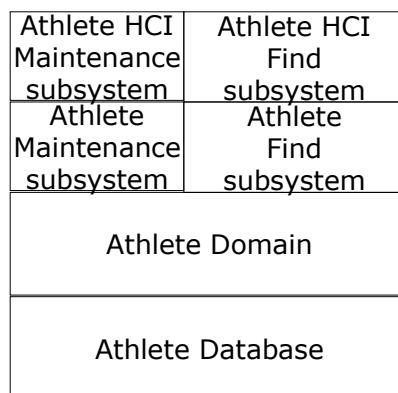


Figure 4.5 Layering and Partitioning Example

This figure uses the four layer architecture. It consists of the following:

1. *The Database Layer.* This layer is responsible for the storage and retrieval of information from a repository. In the example, it is the Athlete Database.
2. *The Domain Layer.* This layer is responsible for services or objects that are shared by different applications. It is particularly defined when systems are distributed. In the example, it is the Athlete Domain.
3. *The Application Layer.* This layer is responsible for executing applications that are representing business logic. In the example, they are the Athlete Maintenance Subsystem and Athlete Find Subsystem.
4. *The Presentation Layer.* This layer is responsible for presenting data to users, devices or other systems. In the example, they are the Athlete HCI Maintenance Subsystem and the Athlete HCI Find Subsystem.

A Sample of Layered Architecture

Layered architectures are used widely in practice. Java 2 Enterprise Edition (J2EE™) adopts a multi-tiered approach and an associated patterns catalog has been developed. This section provides an overview of the J2EE Platform.

The J2EE platform represents a standard for implementing and deploying enterprise applications. It is designed to provide client-side and server-side support for developing distributed and multi-layered applications. Applications are configured as:

1. **Client-tier** which provides the user interface that supports, one or more client types both outside and inside a corporate firewall.
2. **Middle-tier** modules which provide client services through web containers in the Web-tier and business logic component services through the Enterprise JavaBean (EJB) containers in the EJB-tier.
3. **Back-end** provides the enterprise information systems for data management is supported through the Enterprise Information Systems (EIS) which has a standard APIs.

Figure 4.6 depicts the various components and services that make up a typical J2EE environment.

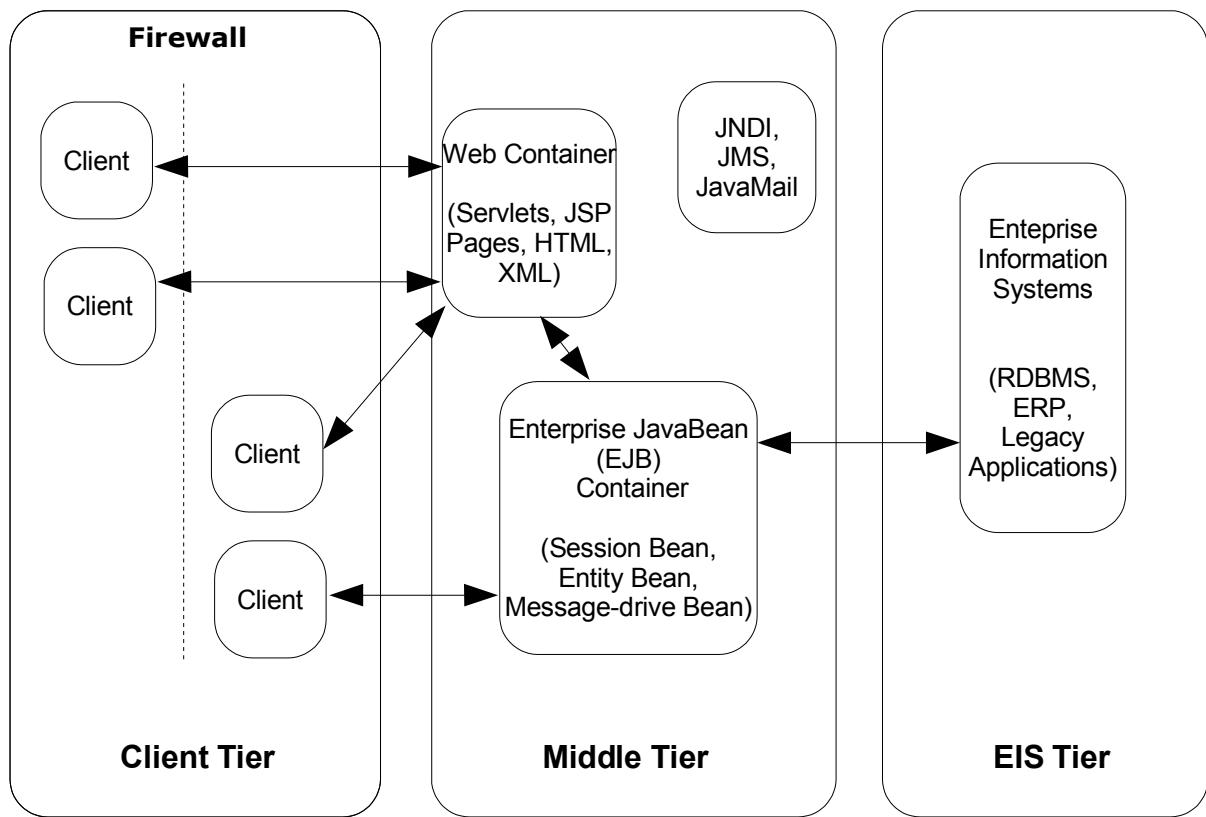


Figure 4.6 Java 2 Enterprise Edition (J2EE) Architecture

For more details on the J2EE Platform, check out the site. http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e to learn how to design enterprise applications.

4.2.3 Developing the Architectural Design

In constructing the software architecture, the design elements are grouped together in packages and subsystems. Interfaces are defined. The relationship among them are illustrated by defining the dependencies. As an example, the **Club Membership Maintenance System**'s analysis model will be used which was developed in the previous chapter.

STEP 1: If the analysis model has not been validated yet, validate the analysis model.

We must ensure that attributes and responsibilities are distributed to the classes, and ensure that a class defines a single logical abstraction. If there are problems with the analysis model, iterate back to the requirements engineering phase. One can use the Analysis Model Validation Checklist.

STEP 2: Package related analysis classes together.

As was mentioned previously, packages are mechanism that allows us to group together model elements. Packaging decisions are based on packaging criteria on a number of different factors which includes configuration units, allocation of resources, user type

groupings, and representation of the existing product and services the system uses. The following provides guidelines on how to group together classes.

Packaging Classes Guidelines

1. Packaging Boundary Classes

- If the system interfaces are likely to be changed or replaced, the boundary classes should be separated from the rest of the design.
- If no major interface changes are planned, the boundary classes should be placed together with the entity and control classes with which they are functionally related.
- Mandatory boundary classes that are not functionally related on any entity or control classes are grouped together separately.
- If the boundary class is related on an optional service, group it in a separate subsystem with the classes that collaborate to provide the said service. In this way, when the subsystem is mapped onto an optional component interface, it can provide that service using the boundary class.

2. Packaging Functionally Related Classes

- If a change in one class affects a change to another class, they are functionally related.
- If the class is removed from the package and has a great impact to a group of classes, the class is functionally related to the impacted classes.
- Two classes are functionally related if they have a large number of messages that they can send to one another.
- Two classes are functionally related if they interact with the same actor, or are affected by the changes made by the same actor.
- Two classes are functionally related if they have a relationship with one another.
- A class is functionally related to the class that creates instances of it.
- Two class that are related to different actors should not be placed on the same package.
- Optional and mandatory classes should not be placed in the same classes.

After making a decision on how to group the analysis classes, the package dependencies are defined. **Package Dependencies** are also known as **visibility**. It defines what is and is not accessible within the package. Table 17 shows the visibility symbol that are placed beside the name of the class within the package.

Visibility Symbols	Descriptions
+	It represents a public model element which can be accessed outside of the package.
-	It represents a private model element which cannot be accessed outside of the package.
#	It represents protected model elements which can be accessed by owning the model element or by inheritance.

Table 17: Package Visibility Symbols

An example of package visibility is depicted in Figure 4.7. Classes within package can collaborate with one another. Thus, they are visible. As an example, classes in Package A, such as **ClassA1**, **ClassA2** and **ClassA3** can collaborate with one another since they reside on the same package. However, when classes resides on different packages, they are inaccessible unless the class has been defined as public. **ClassB1** is visible since it is defined as public. **ClassB2**, on the other hand, is not.

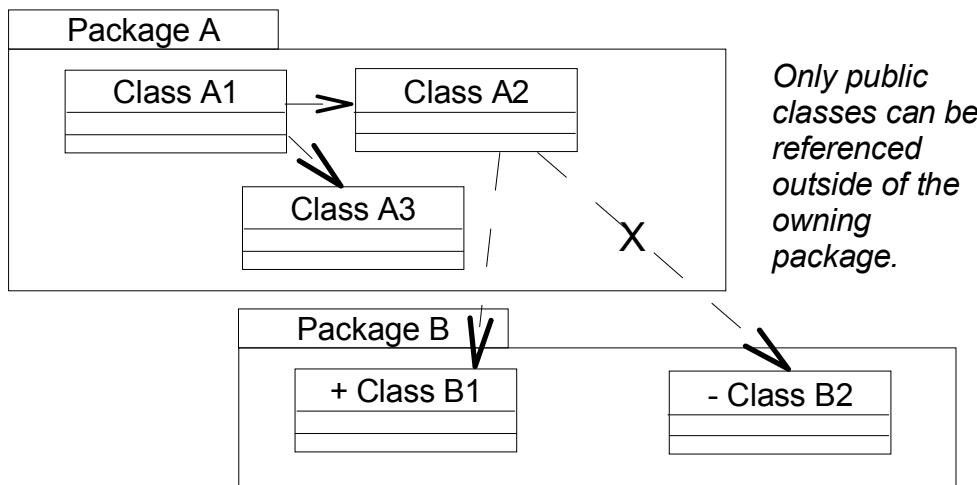


Figure 4.7 Package Visibility

Package Coupling defines how dependencies are defined between packages. It should adhere to some rules.

1. Packages should not be cross-coupled.
2. Packages in lower layers should not be dependent on packages on upper layers.

Layers will be discussed in one of the steps.

3. In general, packages should not skip layers. However, exceptions can be made and should be documented.
4. Packages should not be dependent on subsystems. They should be dependent on other packages or on subsystem interfaces.

The packaging decision for the **Club Membership Maintenance System** is illustrated in Figure 4.8.

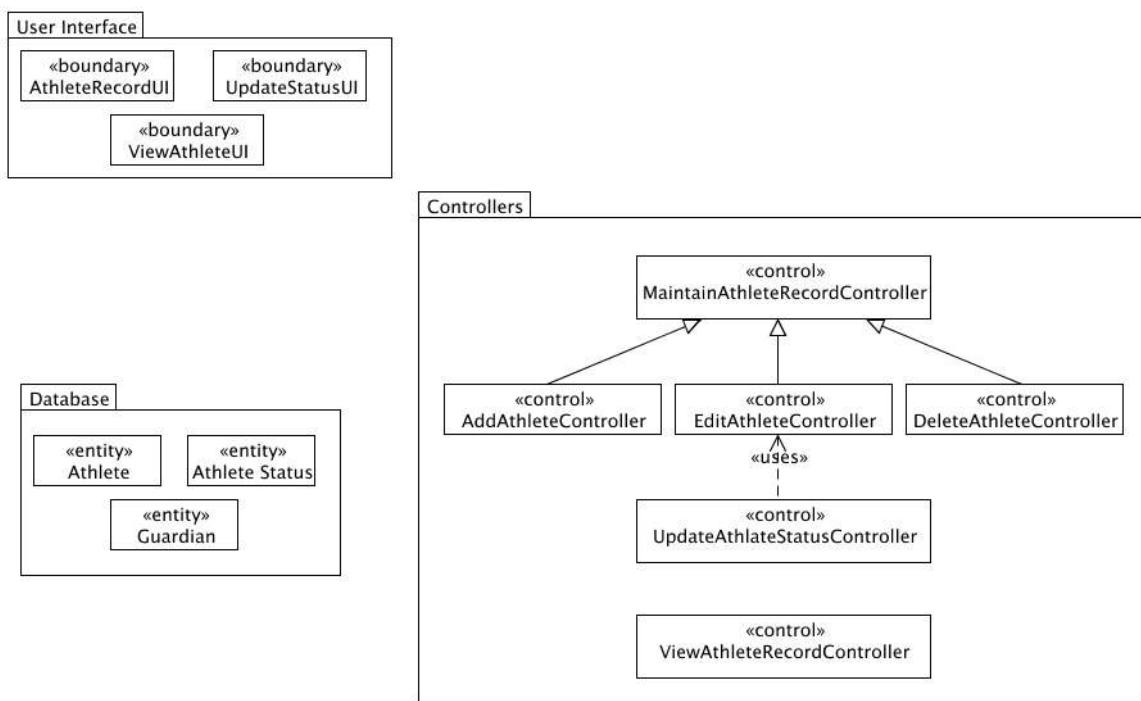


Figure 4.8 Club Membership Maintenance System Packaging Decision

STEP 3: Identify Design Classes and Subsystems.

The analysis classes are analyzed to determine if they can be design classes or subsystems. It is possible that analysis classes are expanded, collapsed, combined or removed from the design. At this point, decisions have to be made on the following:

- Which analysis classes are really classes? Which are not?
- Which analysis classes are subsystems?
- Which components are existing? Which components need to be designed and implemented?

The following serves as a guideline in identifying design classes.

1. An analysis class is a design class if it is a simple class or it represents a single logical abstraction.
2. If the analysis class is complex, it can be refined to become:
 - a part of another class
 - an aggregate class
 - a group of classes that inherits from the same class
 - a package
 - a subsystem
 - an association or relationship between design elements

The following serves as a guideline in identifying subsystems.

1. *Object Collaboration.* If the objects of the classes collaborate only with themselves, and the collaboration produces an observable result, they should be part of the subsystem.
2. *Optionality.* If the associations of the classes are optional, or features which may be removed, upgraded or replaced with alternatives, encapsulate the classes within a subsystem.
3. *User Interface.* Separating together boundary and related entity classes as one subsystem is called **horizontal subsystem** while grouping them together is called **vertical subsystem**. The decision to go for horizontal or vertical subsystem depends on the coupling of the user interface and entity. If the boundary class is used to display entity class information, vertical subsystem is the choice.
4. *Actor.* If two different actors use the same functionality provided by the class, model them as different subsystems since each actor may independently change requirements.
5. *Class coupling and cohesion.* Organize highly coupled classes into subsystems. Separate along the lines of weak coupling.
6. *Substitution.* Represent different service levels for a particular capability as separate subsystem each realizes the same interface.
7. *Distribution.* If the particular functionality must reside on a particular node, ensure that the subsystem functionality maps onto a single node.
8. *Volatility.* Encapsulate into one subsystem all classes that may change over a period of time.

The possible subsystems are:

1. Analysis classes that provide complex services or utilities, and boundary classes.
2. Existing products or external systems in the design such as communication software, database access support, types and data structures, common utilities and application specific products.

In the case of the **Club Membership Maintenance System**, the initial subsystems identified are the boundary classes **AthleteRecordUI** and **UpdateStatusUI**.

STEP 4: Define the interface of the subsystems.

Interfaces are group of externally visible (public) operations. In UML, it contains no internal structure, no attributes, no associations and the operation implementation details are not defined. It is equivalent to an abstract class that has no attributes, no associations and only abstract operations. It allows the separation of the declaration of the behavior from its implementation.

Interfaces define a set of operations which are realized by a subsystem. It serves as a contract to help in the independent development of the components by the development team, and ensures that the components can work together.

Figure 4.9 shows the subsystems defined for the **Club Membership Maintenance System**. The interface is defined by placing a stereotype <<interface>> at the class name compartment. To differentiate it from the boundary class, the name of the interface class begins with the letter 'I'. The Application Subsystem realizes the **IAthleteRecordUI** Interface.

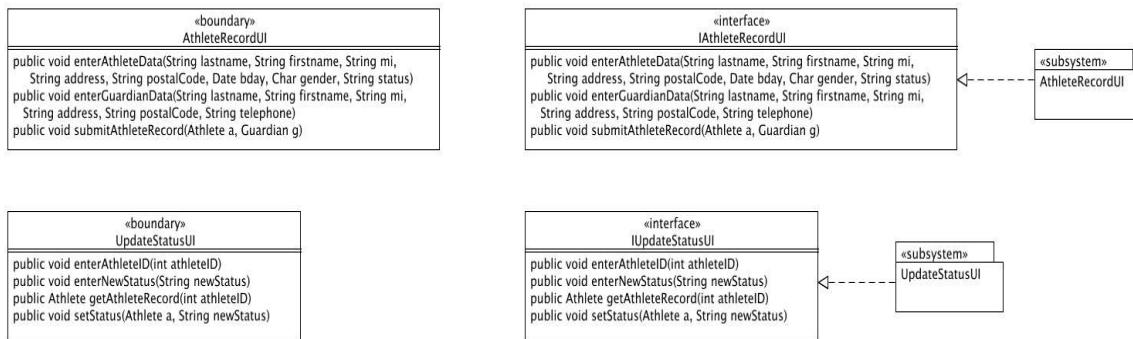


Figure 4.9 Club Membership Maintenance System Subsystem Identification

STEP 5: Layer subsystems and classes.

Defining layers of the software application is necessary to achieve an organized means of designing and developing the software. Design elements (design classes and subsystems) need to be allocated to specific layers in the architecture. In general, most boundary classes tend to appear at the top layer, control classes tend to appear in the middle and entity classes appear below. This type of layering is known as the **three layer architecture** and is shown in Figure 4.10.

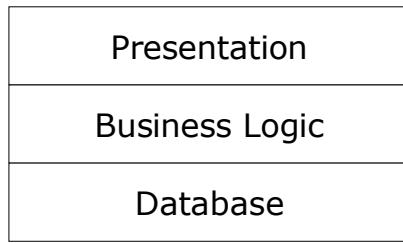


Figure 4.10 Three Layer Architecture

There are many architecture styles and patterns that are more complex and comprehensive than the simple three layer architecture. Normally, they are designed based on technological requirements such as system should distributed, system should support concurrency, system is considered a web-based application. For simplicity purpose, the case study uses the three layer architecture. When classes within layers are elaborated, chances are additional layers and packages are used, defined and refined.

The following serves as a guideline in defining layers of the architecture.

1. *Consider visibility.* Dependencies among classes and subsystem occur only within the current layer and the layer directly below it. If dependency skips, it should be justified and documented.
2. *Consider volatility.* Normally, upper layers are affected by requirements change while lower layers are affected by environment and technology change.
3. *Number of Layers.* Small systems should have 3 – 4 layers while larger systems should have 5 – 7 layers.

For the **Club Membership Maintenance System**, the layers are depicted in Figure 4.11. Notice that the analysis classes that are identified as subsystems are replaced with the subsystem interface definition. For clarification, the realization of the interface is not shown.

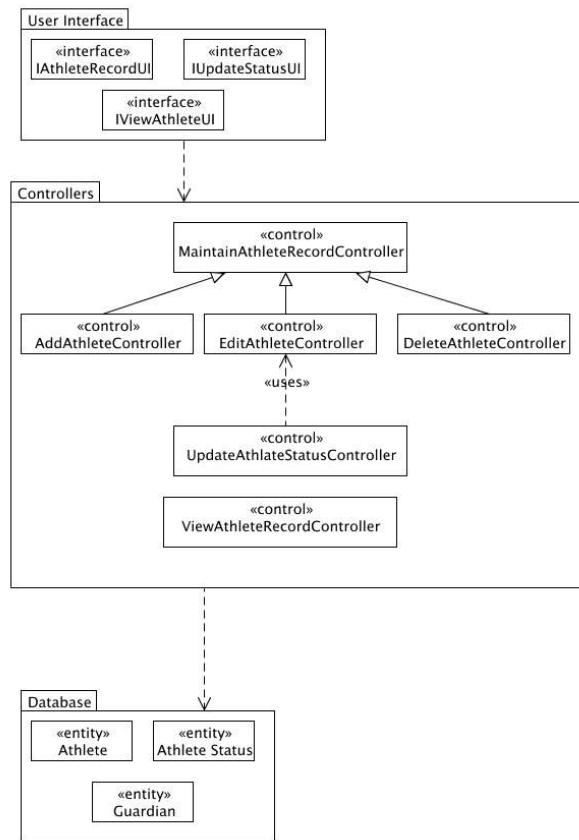


Figure 4.11 Club Membership Maintenance Layering Decision

4.2.4 Software Architecture Validation Checklist

Like any other work product, the software architecture should be validated. Use the following list questions as the checklist.

Layers

1. For smaller systems, are there more than four layers?
2. For larger systems, are there more than seven layers?

Subsystems and Interfaces

1. Are the subsystems partitioning done in a logically consistent way across the architecture?
2. Is the name of the interface depict the role of the subsystem within the entire system?
3. Is the interface description concise and clear?
4. Are all operations that the subsystem needs to perform identified? Are there any missing operations?

Packages

1. Are the name of the packages descriptive?
2. Does the package description match the responsibilities?

4.3 Design Patterns

A **design pattern** describes a proven solution to a problems that keep recurring. It leverages the knowledge and insights of other developers. They are reusable solutions to common problems. It addresses individual problems but can be combined in different ways to achieve an integrated solution for an entire system.

Design patterns are not frameworks. **Frameworks** are partially completed software systems that may be targeted at a specific type of application. An application system can be customized to an organization's needs by completing the unfinished elements and adding application specific elements. It would involve specialization of classes and the implementation of some operations. It is essentially a reusable mini-architecture that provides structure and behavior common to all applications of this type.

Design patterns, on the other hand, are more abstract and general than frameworks. It is a description of the way that a problem can be solved but it is not, by itself, is a solution. It cannot be directly implemented; a successful implementation is an example of a design pattern. It is more primitive than a framework. A framework can use several patterns; a pattern cannot incorporate a framework.

Patterns may be documented using one of several alternative templates. The pattern template determines the style and structure of the pattern description, and these vary in the emphasis they place on different aspects of the pattern. In general, a pattern description includes the following elements:

1. **Name.** The name of the pattern should be meaningful and reflects the knowledge embodied by the pattern. This may be a single word or a short phrase.
2. **Context.** The context of the pattern represents the circumstances or pre-conditions under which it can occur. It should be detailed enough to allow the applicability of the pattern to be determined.
3. **Problem.** It should provide a description of the problem that the pattern addresses. It should identify and describe the objectives to be achieved within a specific context and constraining forces.
4. **Solution.** It is a description of the static and dynamic relationships among the components of the pattern. The structure, the participants and their collaborations are all described. A solution should resolve all the forces in the given context. A solution that does not solve all the forces fails.

The use of design patterns requires careful analysis of the problem that is to be addressed and the context in which it occurs. All members of the development team should receive proper training. When a designer is contemplating the use of design patterns, issues should be considered. The following questions provide a guideline to resolve these issues.

1. Is there a pattern that addresses a similar problem?
2. Does the pattern trigger an alternative solution that may be more acceptable?
3. Is there another simple solution? Patterns should not be used just for the sake of it?
4. Is the context of the pattern consistent with that of the problem?
5. Are the consequences of using the pattern acceptable?
6. Are constraints imposed by the software environment that would conflict with

the use of the pattern?

How do we use the pattern? The following steps provides a guideline in using a selected design pattern.

1. To get a complete overview, read the pattern.
2. Study the structure, participants and collaboration of the pattern in detail.
3. Examine the sample codes to see the pattern in use.
4. Name the pattern participants (i.e. Classes) that are meaningful to the application.
5. Define the classes.
6. Choose application specific names for the operations.
7. Code or implement the operations that perform responsibilities and collaborations in the pattern.

It is important to note at this time that a pattern is not a prescriptive solution to the problem. It should be view as a guide on how to find a suitable solution to a problem.

As an example, there are a set of common design patterns for the J2EE platform. This section briefly discusses some of these patterns. To find out more about J2EE Design Patterns, you can check out this website, <http://java.sun.com/blueprints/patterns/index.html>.

Composite View

Context

The Composite View Pattern allows the development of the view more manageable through the creation of a template to handle common page elements for a view. For a web application, a page contains a combination of dynamic contents and static elements such header, footer, logo, background, and so forth. The template captures the common features.

Problem

Difficulty of modifying and managing the layout of multiple views due to duplication of code. Pages are built by formatting code directly on each view.

Solution

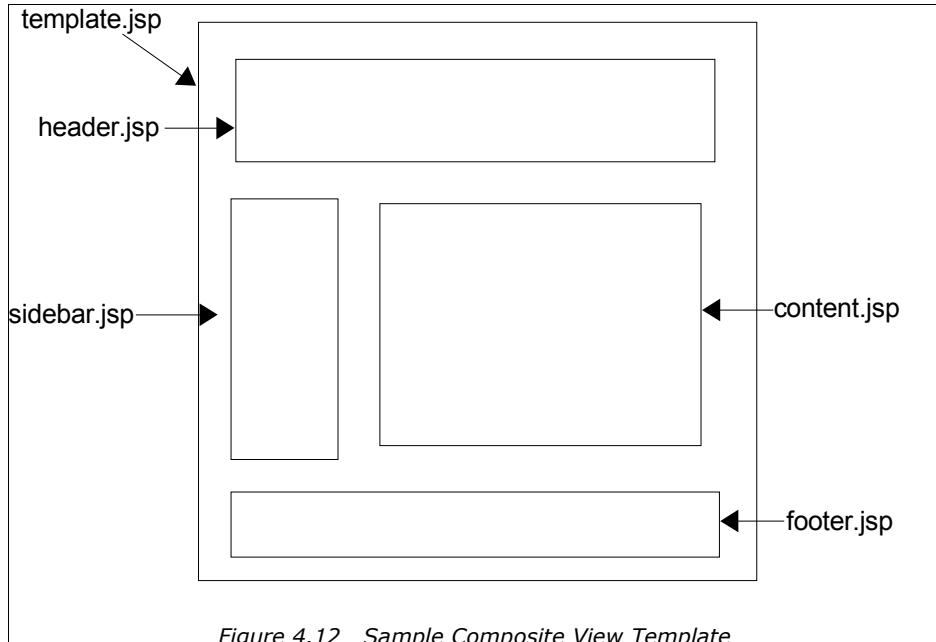
Use this pattern when a view is composed of multiple atomic sub-views. Each component of the template may be included into the whole and the layout of the page may be managed independently of the content. This solution provides the creation of a composite view through the inclusion and substitution of modular dynamic components. It promotes reusability of atomic portions of the view by encouraging modular design. It is used to generate pages containing display components that may be combined in a variety of ways. This scenario occurs for a page that shows different information at the same time such as found in most website's home pages where you might find new feeds, weather information, stock quotes etc.

A benefit of using this pattern is that interface designer can prototype the layout of the page, plugging in static content on each component. As the development of the project

progresses, the actual content is substituted for these placeholders.

This pattern has a drawback. There is a runtime overhead associated with it. It is a trade-off between flexibility and performance.

Figure 4.12 shows an example of a page layout. The page consists of the following views: header, sidebar, footer and content.



This form may be an application website screen and is defined within the file definition as follows:

```
<screen name="main">
    <parameter key="banner" value="/header.jsp" />
    <parameter key="footer" value="/footer.jsp" />
    <parameter key="sidebar" value="/sidebar.jsp" />
    <parameter key="body" value="/content.jsp" />
</screen>
```

Text 1 Sample Implementation of Composite View

Figure 4.13 shows the class diagram that represents this pattern while Figure 4.14 shows the collaboration of the classes.

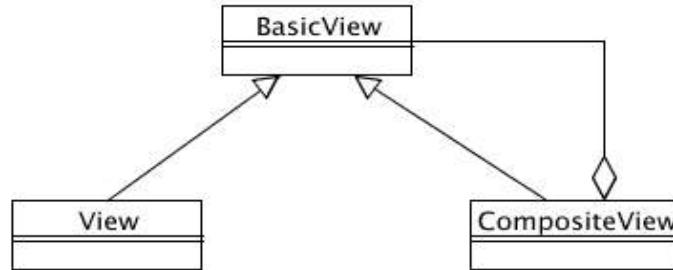


Figure 4.13 Composite View Class Diagram

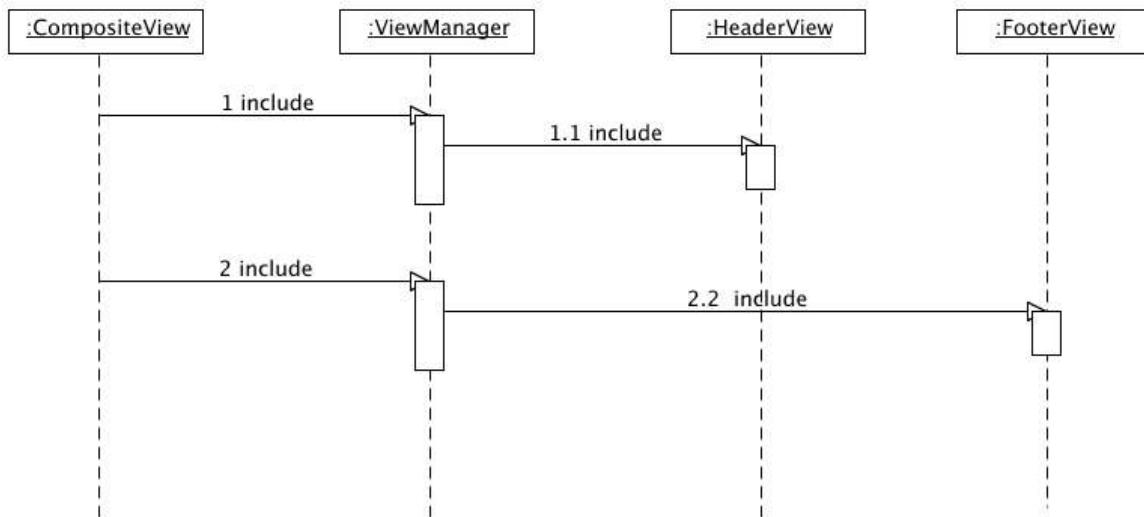


Figure 4.14 Composite View Sequence Diagram

Table 18 describes the responsibilities of each of the classes.

Class	Description
CompositeView	It is an aggregate of multiple views.
ViewManager	It manages the inclusion of portions of template fragments into the composite view.
IncludedView	It is a subview that is one atomic piece of a larger whole view. This included view can also consist of multiple views.

Table 18: Composite View Design Pattern Classes

Front Controller

Context

The Front Controller Pattern provides a centralized controller for managing requests. It receives all incoming client requests, forwards each request to an appropriate request handler, and presents an appropriate response to the client.

Problem

The system needs a centralized access point for presentation-tier request handling to support the integration of system data retrieval, view management, and navigation. When someone accesses the view directly without going through a centralized request handler, problems may occur such as:

- each view may provide its own system service which may result in duplication of code
- navigation of view is responsibility of the view which may result in commingled view content and view navigation.

Additionally, distributed control is more difficult to maintain since changes will often be made at different places in the software.

Solution

A controller is used as an initial point of contact for handling a request. It is used to manage the handling of services such as invoking security services for authentication and authorization, delegating business processing, managing the appropriate view, handling of errors, and managing the selection of content creation strategies. It centralizes decision-making controls. The class diagram and sequence diagram for this pattern are shown in Figure 4.15 and Figure 4.16.

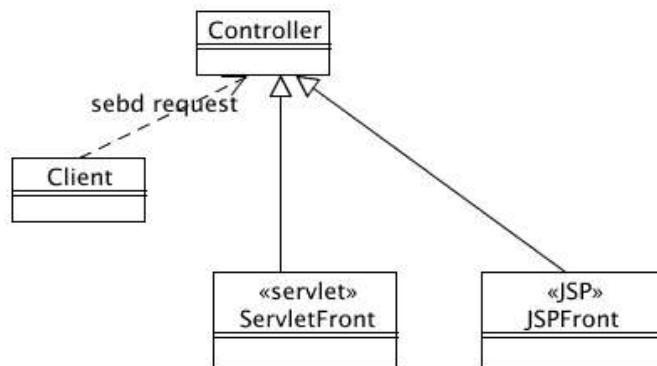


Figure 4.15 Front Controller Class Diagram

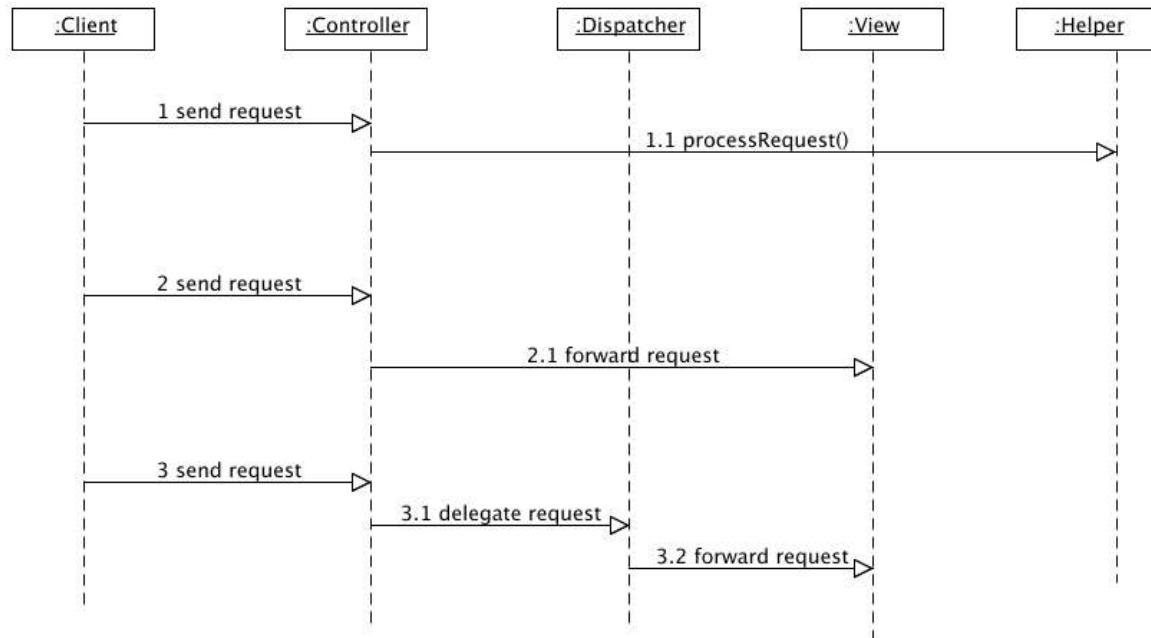


Figure 4.16 Front Controller Sequence Diagram

The roles of the classes are enumerated in Table 19.

Class	Description
Controller	It is the initial contact point for handling all requests in the system. It may delegate to a Helper object.
Dispatcher	It is responsible for view management and navigation, managing the choice of the next view to present to the user and the mechanism for vectoring control to his resource. It can be encapsulated within a controller or a separate component working in coordination with the controller. It uses the <i>RequestDispatcher</i> object and encapsulates some additional methods.
Helper	It is responsible for helping a view or controller complete the processing. It has numerous responsibilities which include gathering data required by the view and storing data in the intermediate model.
View	It represents and displays information to the client.

Table 19: Front Controller Design Pattern Classes

As an example of implementing a front controller, a servlet is used to represent this controller. A fraction of the sample code is represented below. In the **processRequest()** method, a **RequestHelper** (**Helper**) object is used to execute the command which represents the client's request. After the command is executed, a **RequestDispatcher** object is used to get the next view.

```
        catch (Exception e) {
            request.setAttribute(resource.getMessageAttr(),
                "Exception occurred: " + e.getMessage());
            page=resource.getErrorPage(e);
        }
        //Message Sequence 2 example of the
        //Sequence Diagram.
        //Dispatch control to view
        dispatch(request, response, page);
        ...
        ...
        private void dispatch(HttpServletRequest request,
            HttpServletResponse response, String page) throws
            javax.servlet.ServletException, java.io.IOException {
            RequestDispatcher dispatcher =
                getServletContext().getRequestDispatcher(page);
            dispatcher.forward(request, response);
        }
    }
```

Text 2 Sample Implementation of Front Controller

Data Access Object Design Pattern

Context

The Data Access Object (DAO) Design Pattern separates resource's client interface from its data access mechanisms. It adapts a specific data resource's access API to a generic client interface. It allows data access mechanism to change independently of the code that uses the data.

Problem

Data will be coming from different persistent storage mechanism such as relational database, mainframe or legacy systems, B2B external services, LDAP etc. Access mechanism varies based on the type of storage. It allows maintainability of codes by separating access mechanism from the view and processing. When the storage type changes, we only need to change the code that specifies the access mechanism.

Solution

Use the Data Access Object to abstract and encapsulate all access to the data source. It implements the access mechanism required to work with the data source. The data source can be a relational database system, external service such as B2B, a repository similar to LDAP etc.

The class diagram of this pattern is depicted in Figure 4.17.

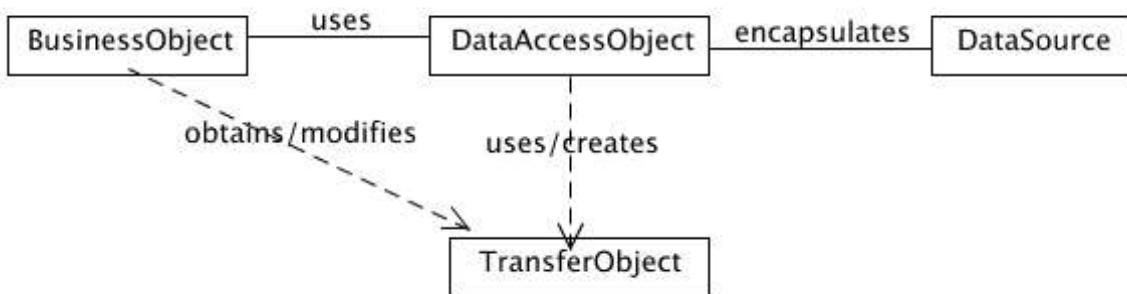


Figure 4.17 Data Access Object (DAO) Class Diagram

Table 20 shows the classes that are used in persistence.

Class	Description
Business Object	It represents the data client. It is the object that requires access to the data source to obtain and store data.
DataAccessObject	It is the primary object of this pattern. It abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source.
DataSource	It represents a data source which can be a RDBMS, OODBMS, XML system or a flat-file. It can also be another system (legacy or mainframe), service from B2B or a kind of repository such as LDAP.
TransferObject	The represents a data carrier. It is used to return data to the client or the client can use it to update data.

Table 20: Data Access Object (DAO) Design Pattern Classes

The sequence diagram of this pattern is illustrated in Figure 4.18.

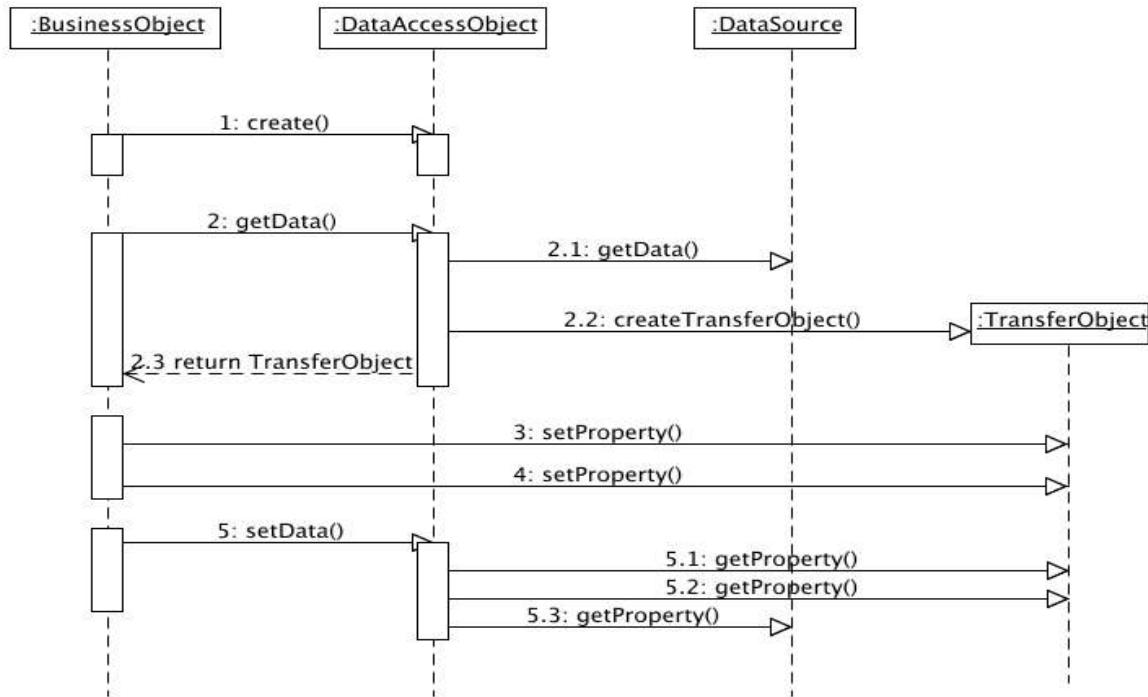


Figure 4.18 Data Access Object (DAO) Sequence Diagram

The Dynamic View shows the interaction among the classes within this pattern. Specifically, four interactions can be noted from the diagram above. They are listed below.

1. A **BusinessObject** needs to create **DataAccessObject** for a particular **DataSource**. This is needed in order to retrieve or store enterprise data. In the diagram, it is the message number 1.
2. A **BusinessObject** may request to retrieve data. This is done by sending a **getData()** message to the **DataAccessObject** which in turn, would request the data source. A **TransferObject** is created to hold the data retrieve from the **DataSource**. In the diagram, it is represented by message number 2 and its sub-messages.
3. A **BusinessObject** may modify the contents of the **TransferObject**. This simulates the modification of the data. However, data are not yet modified in the data source. This is represented by message number 3 and 4.
4. In order to save the data in the data source, a **setData()** message is sent to the **DataAccessObject**. The **DataAccessObject** retrieve the data from the **TransferObject** and prepares it for updating. It will, then, send **setData()**

message to the **DataSource** to update its record.

Data Access Object Strategy

The DAO pattern can be made highly flexible by adopting the *Abstract Factory Pattern* and *Factory Method Pattern*. If the underlying storage is not subject to change from one implementation to another, the Factory Method pattern can be used. The class diagram is presented in Figure 4.19.

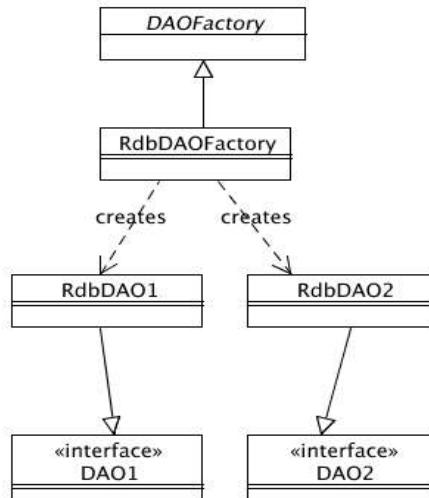


Figure 4.19 Factory Method Strategy

If the underlying storage is subject to change from one implementation to another, the *Abstract Factory Pattern* can be used. It can in turn build on and use the *Factory Method* implementation. In this case, this strategy provides an abstract DAO factory object that can construct various types of concrete DAO factories. Each factory supporting a different type of persistent storage. Once a concrete factory is obtained for a specific implementation, it is used to produce DAOs supported and implemented in that implementation. The class diagram is represented in Figure 4.20.

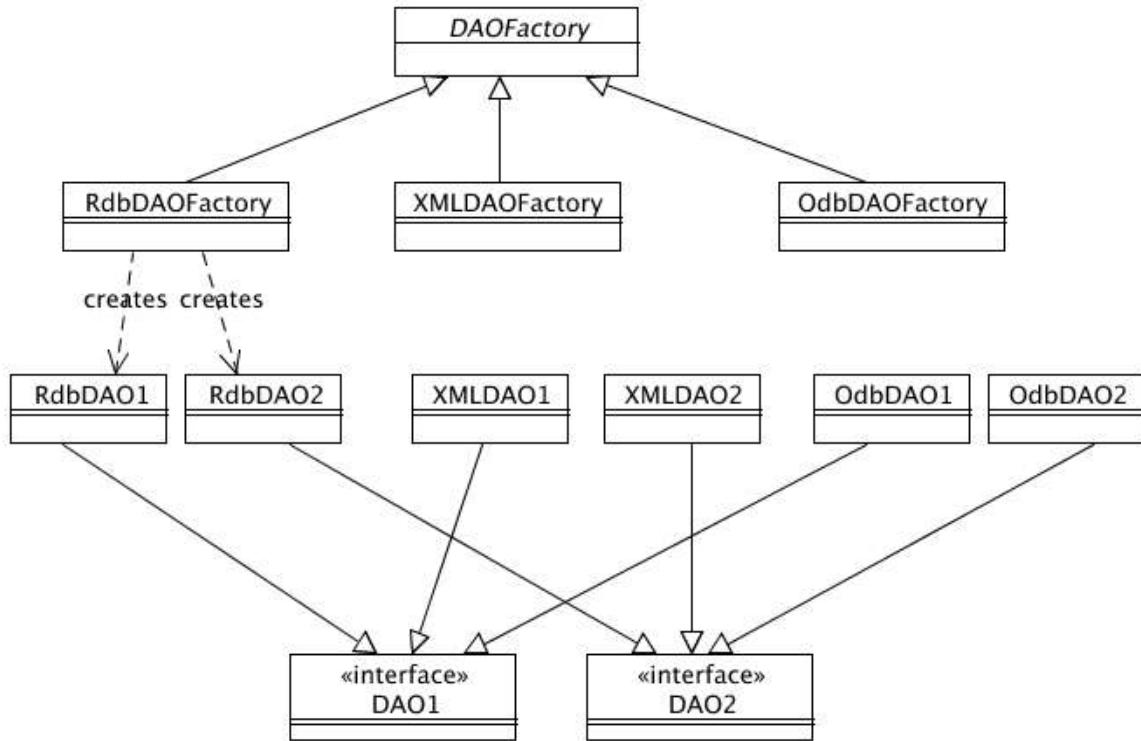


Figure 4.20 Abstract Factory Pattern

- In this class diagram, **DAOFactory** is an abstraction of a *Data Access Object Factory* which is inherited and implemented by different concrete DAO factories to support storage implementation-specific access. The concrete DAO factories create data access objects which implement interfaces of a data access objects.

An example of an implementation of the **DAOFactory** is shown below by the following codes. The first code shows the definition of the abstract class **DAOFactory**.

```
//Abstract class DAOFactory
public abstract class DAOFactory {
    //List of DAO Types
    public static final int DEFAULT = 1;
    public static final int MYSQL = 2;
    public static final int XML = 3
    ...
    ...
    // The concrete factories should implement these methods.
    public abstract AthleteDAO getAthleteDAO();
    public abstract AthleteStatusDAO getAthleteStatusDAO();
    ...
    ...
    public static DAOFactory getDAOFactory(int whichFactory){
        switch(whichFactory) {
            case DEFAULT : return new DefaultDAOFactory();
            case MYSQL : return new MySQLDAOFactory();
            case XML : return new XMLDAOFactory();
            ...
            default : return null;
        }
    }
}
```

Text 3 Definition of DAOFactory Abstract Class

Each of the identified DAO types should be implemented. In this example, the **DefaultDAOFactory** is implemented; the other are similar except for specifics of each implementation such as the drivers, database URL etc.

```
//DefaultDAOFactory Implementation

import java.sql.*;

public class DefaultDAOFactory extends DAOFactory {
    public static final String
        DRIVER = "sun.jdbc.odbc.JdbcOdbcDriver";
    public static final String
        URL = "jdbc:odbc:AngBulilitLiga";
    ...
    ...

    // method for connection
    public static Connection createConnection() {
        // Use DRIVER and URL to create connection.
        ...
    }

    public AthleteDAO getAthleteDAO() {
        // DefaultAthleteDAO implements AthleteDAO
        return new DefaultAthleteDAO();
    }

    public AthleteStatusDAO getAthleteStatusDAO() {
        // DefaultAthleteStatusDAO implements AthleteStatusDAO
        return new DefaultAthleteStatusDAO();
    }
    ...
}
```

Text 4 DefaultDAOFactory Implementation

The **AthleteDAO** is an interface definition and it is coded as follows:

```
//Interface of all AthleteDAO

public interface AthleteDAO {
    public int insertAthlete(Athlete a);
    public int deleteAthlete(Athlete a);
    public int updateAthlete(Athlete a);
    public Athlete findAthleteRecord(String criteria);
    public RowSet selectAthlete(String criteria);
    ...
}
```

Text 5 AthleteDAO Interface Definition

The **DefaultAthleteDAO** implements all of the methods of the interface **AthleteDAO**. The skeleton code of this class is shown below.

```
import java.sql.*

public class DefaultAthleteDAO implements AthleteDAO {
    public DefaultAthleteDAO() {
        //initialization
    }

    public int insertAthlete (Athlete a) {
        // Implement insert athlete record here.
        // return 0 if successful or -1 if not.
    }

    public int updateAthlete (Athlete a) {
        // Implement update athlete record here.
        // return 0 if successful or -1 if not.
    }

    public int deleteAthlete (Athlete a) {
        // Implement delete athlete record here.
        // return 0 if successful or -1 if not.
    }

    ...
    ...
}
```

Text 6 DefaultAthleteDAO Implementation

The **Athlete** Transfer Object is implemented using the following code. It is used by the DAOs to send and receive data from the database.

```
public class Athlete {  
    private int athleteID;  
    private String lastName;  
    private String firstName;  
    // Other attributes of an athlete include Guardian  
  
    // getter and setter methods  
    public void setAthleteID(int id) {  
        athleteID = id;  
    }  
    ...  
  
    public int getAthleteID() {  
        return athleteID;  
    }  
    ...  
}
```

Text 7 Athlete Transfer Object Implementation

A fraction of the a client code that shows the use of the object of the classes is presented below.

```
...
...
// create the needed DAO Factory
DAOFactory dFactory =
    DAOFactory.getDAOFactory(DAOFactory.DEFAULT);

// Create AthleteDAO
AthleteDAO = athDAO = dfactory.getAthleteDAO();

...
// Insert a new athlete
// Assume ath is an instance of
// Athlete Transfer Object
int status = athDAO.insertAthlete(ath);
```

Text 8 Fraction of Client Code Using DAO Objects

4.3.1 Model-View-Controller Design Pattern

Problem

Enterprise applications need to support multiple types of users with multiple types of interfaces. For one application, it may require an HTML form for web customers, a WML form for wireless customers, a Java Swing Interface for administrators, and an XML-based Web services for suppliers. The forces behind this pattern are as follows:

- The same enterprise data needs to be accessed by different views.
- The same enterprise data needs to be updated through different interactions.
- The support of multiple types of views and interactions should not impact the components that provide the core functionality of the enterprise application.

Context

The application presents contents to users in numerous pages containing various data.

Solution

The **Model-View-Controller (MVC)** is a widely used design pattern for interactive applications. It divides functionality among objects involved in maintaining and presenting data to minimize the degree of coupling between the objects. It divides the application into three, namely, model, view and controller.

1. **Model.** It represents enterprise data and the business rules that govern access to and updates to this data. It serves as a software approximation of real-world process, so simple real-world modeling techniques apply when defining the model.
2. **View.** It renders the contents of a model. It accesses enterprise data through the model and specifies how that data is presented to the actors. It is responsible for maintaining the consistency in its presentation when the underlying model changes. It can be achieved through a push model, where the view registers itself with the model for change notifications or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data.
3. **Controller.** It translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, it can be clicks on buttons or menu selections. For Web application, they appear as GET and POST HTTP requests. The actions performed on the model can be activating device, business process or changing the state of a model.

Figure 4.21 shows the Model-View-Controller Design Pattern.

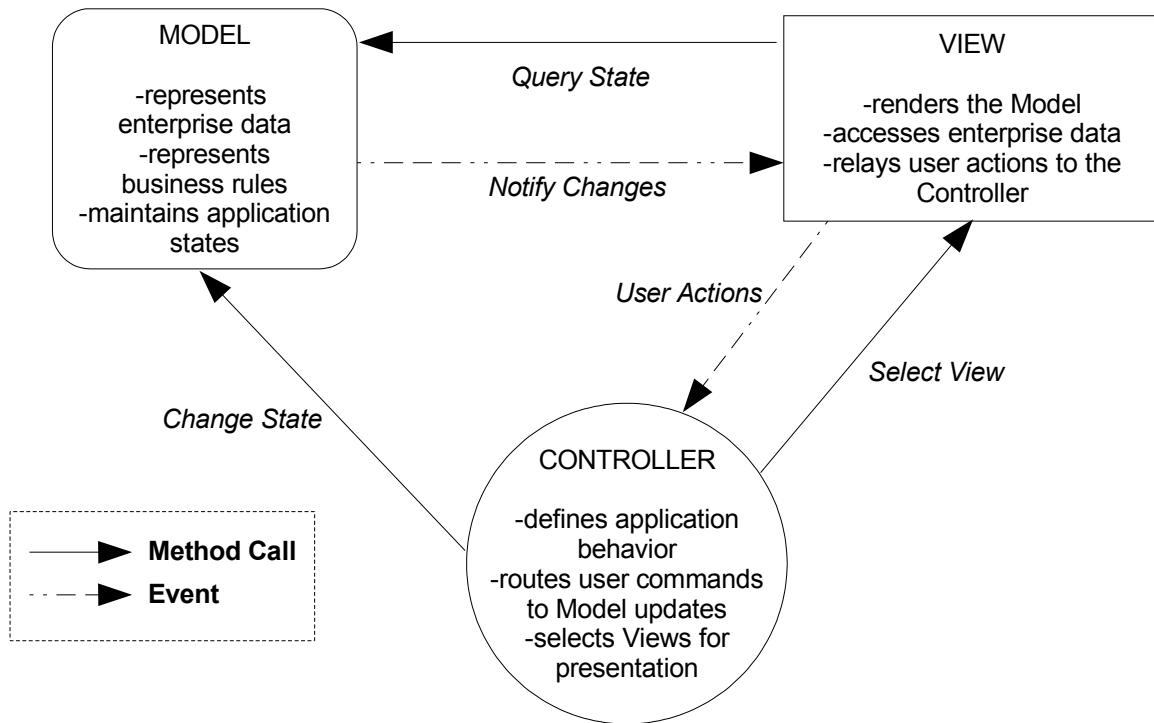


Figure 4.21 Model-View-Controller Design Pattern

The strategies by which MVC can be implemented are as follows:

- For Web-based clients such as browsers, use *Java Server Pages (JSP)* to render the view, *Servlet* as the controller, and *Enterprise JavaBeans (EJB)* components as the model.
- For Centralized controller, instead of having multiple servlets as controller, a main servlet is used to make control more manageable. The Front-Controller pattern can be a useful pattern for this strategy.

4.4 Data Design

Data design, also known as **data architecting**, is a software engineering task that creates a model of the data in a more implementation specific representation. The techniques used here falls under the field of database analysis and design, normally, discussed in a Database Course. We will not discuss how databases are analyzed, designed and implemented. For this course, we assume that the database was implemented using a relational database system. The logical database design is illustrated using a class diagram in Figure 4.22.

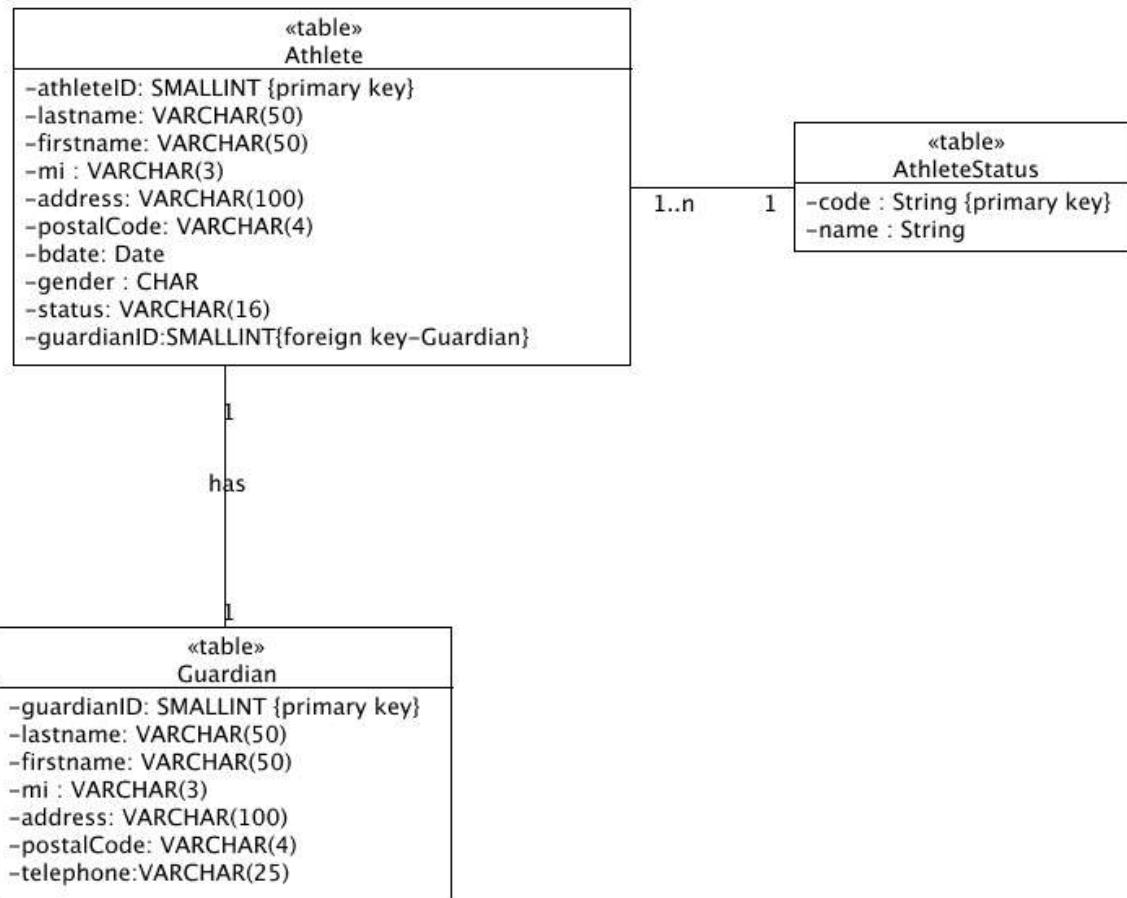


Figure 4.22 Ang Bulilit Liga Logical Database Design

For the **Club Membership Maintenance System**, three tables were defined. The **Athlete** table contains data about the athlete. The **Guardian** table contains data about the guardian of the athlete. To simplify the record-keeping, it is assumed that there is only one guardian per athlete. The **AthleteStatus** table contains the coding scheme for the athlete status.

To access the data in a database system, our programs should be able to connect to the database server. This section discusses the concept of persistence and teaches how to model persistent classes. In terms of the our software architecture, another layer, called

the **persistent layer**, is created on top of the **database layer**.

Persistence means to make an element exists even after the application that created it terminates. For classes that needs to be persistent, we need to identify:

- *Granularity*. It is the size of the persistent object.
- *Volume*. It is the number of objects to keep persistent.
- *Duration*. It defines how long to keep the object persistent.
- *Access Mechanism*. It defines whether more or less the object is constant, i.e., do we allow modifications or updates.
- *Reliability*. It defines how the object will survive if a crash occurs.

There are a lot of design patterns that can be used to model persistence. Two persistence design patterns are discussed in the succeeding sections.

4.4.1 JDBC Design Pattern

This section discusses the pattern of use of the persistent mechanism chosen for the Relational Database Management System (RDBMS) classes which is the Java Database Connectivity (JDBC) Design Pattern. There are two views for this pattern, namely, **static view** and the **dynamic view**.

Static View of Design Pattern of Persistence

The static view is an object model of the pattern. It is illustrated using a class diagram. Figure 4.23 shows the design pattern for the persistent classes.

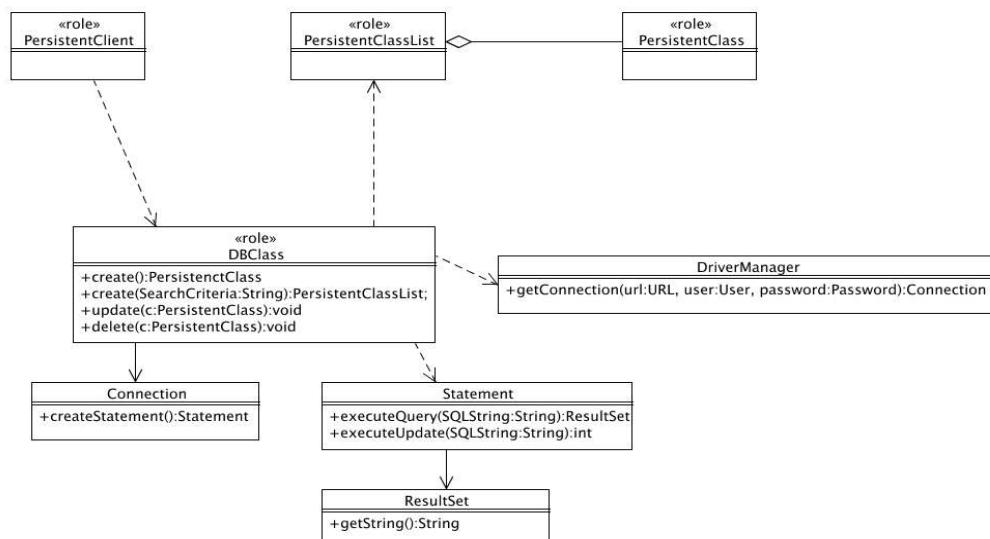


Figure 4.23 JDBC Design Pattern Static View

Table 21 shows the classes that are used in persistence.

Class	Description
PersistentClient	It is a class requesting data from the database. Normally, these are control classes asking something from an entity class. It works with a DBClass .
DBClass	It is a class responsible for reading and writing persistent data. It is also responsible for accessing the JDBC database using the DriverManager class.
DriverManager	It is a class that returns a connection to the DBClass . Once a connection is open the DBClass can create SQL statement that will be sent to the underlying RDBMS and execute using the Statement class.
Statement	It is a class that represents an SQL statement. The SQL statement is the language used to access data from the database. Any result returned by the SQL statement is placed in the class ResultSet .
ResultSet	It is a class that holds the records returned by a query.
PersistentclassList	It is a class that is used to return a set of persistent objects as a result of a database query. One record is equivalent to a PersistentClass in the list.

Table 21: JDBC Design Pattern Classes

The **DBClass** is responsible for making another class instance persistent. It understands the OO-to-RDBMS mapping. It has the behavior to interface with RDBMS. **Every class that needs to be persistent will have a corresponding DBClass!**

Dynamic View of Design Pattern of Persistence

The dynamic view of the design pattern of persistence shows how the classes from the static view interacts with one another. The sequence diagram is used to illustrate this dynamic behavior. There are several dynamic behavior that is seen in this pattern, specifically, they are **JDBC Initialization**, **JDBC Create**, **JDBC Read**, **JDBC Update** and **JDBC Delete**.

1. **JDBC Initialization.** Initialization must occur before any persistent class can be accessed. It means one needs to establish connection with the underlying RDBMS. It involves the following:

- loading the appropriate driver
- connect to the database

The sequence diagram of JDBC Initialization is shown in Figure 4.24.



Figure 4.24 JDBC Initialization

The **DBClass** loads the appropriate driver by calling the `getConnection(url, user, password)`. This method seeks to establish a connection to the given database *URL*. The **DriverManager** attempts to select an appropriate driver from the set of registered JDBC drivers.

2. **JDBC Create.** This behavior creates a record. It executes the INSERT-statement of SQL. It assumes that a connection has been established. The sequence diagram is shown in Figure 4.25.

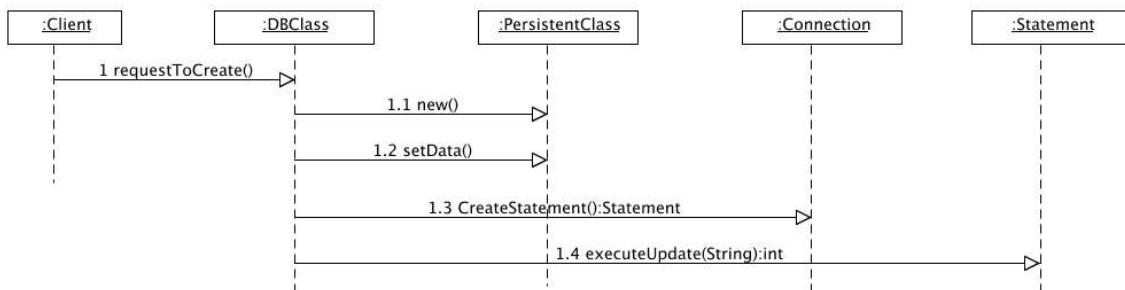


Figure 4.25 JDBC Create

Sequence Diagram Flow Description:

- The **PersistencyClient** asks the **DBClass** to create a new class.
- The **DBClass** creates a new instance of the **PersistentClass** (`new()`) and assigns

values to the **PersistentClass**.

- The **DBClass**, then, creates a new Statement using the **createStatement()** of the **connection**. Normally, this is an INSERT-statement in SQL.
- The **Statement** is executed via **executeUpdate(String):int** method. A record is inserted in the database.

- 3. JDBC Read.** This behavior retrieves records from the database. It executes the SELECT-statement. It also assumes that a connection has been established. The sequence diagram is shown in Figure 4.26.

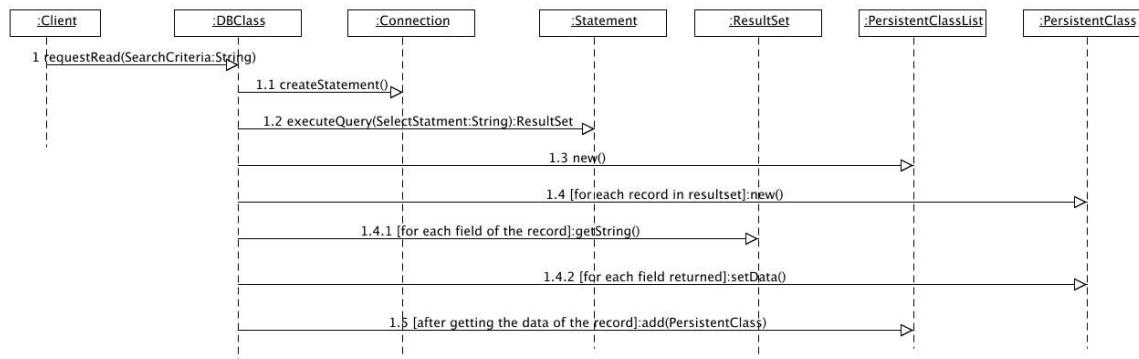


Figure 4.26 JDBC Read

- The **PersistencyClient** asks the **DBClass** to retrieve records from the database. **SearchCriteria** string qualifies what records to return.
- The **DBClass** creates a SELECT-statement **Statement** using the **createStatement()** method of **Connection**.
- The **Statement** is executed via **executeQuery()** and returns a **ResultSet**.
- The **DBClass** instantiates a **PersistentclassList** to hold the records represented in the **ResultSet**.
- For each record in the **ResultSet**, the **DBClass** instantiates a **PersistentClass**.
- For each field of the record, assign the value of that field (**getString()**) to the appropriate attribute in the **PersistentClass** (**setData()**).
- After getting all data of the record and mapping it to the attributes of the **PersistentClass**, add the **PersistentClass** to the **PersistentclassList**.

- 4. JDBC Update.** This executes the UPDATE-statement of SQL. It changes the values of an existing record in the database. It assumes that connection has been established. The sequence diagram is shown in Figure 4.27.

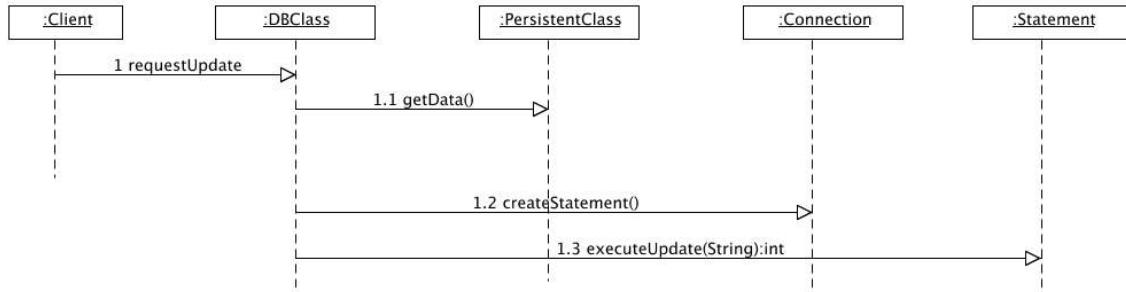


Figure 4.27 JDBC Update

- **PersistencyClient** asks the **DBClass** to update a record.
- The **DBClass** retrieves the appropriate data from the **PersistentClass**. The **PersistentClass** must provide access routine for all persistent data that the **DBClass** needs. This provides external access to certain persistent attributes that would have been otherwise have been private. This is a price that you pay to pull the persistent knowledge out of the class that encapsulates them.
- The **Connection** will create an UPDATE-Statement.
- Once the **Statement** is built, the update is executed and the database is updated with the new data from the class.

5. **JDBC Delete**. This executes the DELETE-statement of SQL. It deletes records in the database. It assumes that connection has been established. The sequence diagram is shown in Figure 4.28.

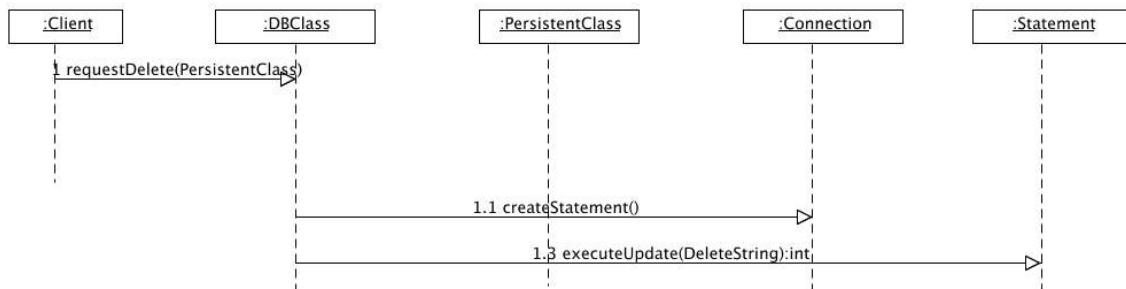


Figure 4.28 JDBC Delete

- The **PersistencyClient** asks the **DBClass** to delete the record.
- The **DBClass** creates a DELETE-Statement and executes it via **executeUpdate()**

method.

4.4.2 Developing the Data Design Model

In developing the Data Design Model, a pattern should be decided on. In the following example, we will use the JDBC Design Pattern to model the **Club Membership Maintenance Data Design**.

STEP 1. Define the static view of the data elements that needs to be persistent.

Using the static view of the design pattern, model the classes that collaborate. Figure 4.29 shows the application of the JDBC Design pattern. The **DBAthlete** class is used to connect and execute SQL statements. It also populates the persistence classes, **PCLAthlete**, **Athlete** and **Guardian**.

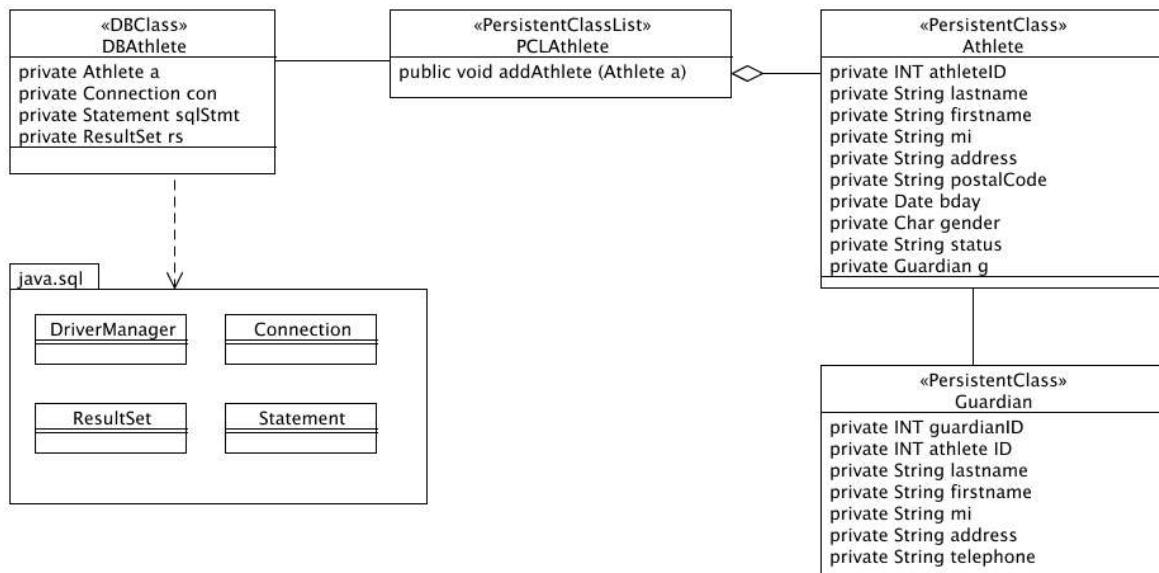


Figure 4.29 DBAthlete Static View

STEP 2: For all behavior, model the behavior of the classes in a sequence diagram.

All the JDBC Dynamic Views should be modeled to determine how the DBClasses and Persistent Classes collaborate using the Sequence Diagrams. The succeeding diagrams illustrates the JDBC Read (Figure 4.30), Create (Figure 4.31), Update (Figure 4.32) and Delete (Figure 4.33).

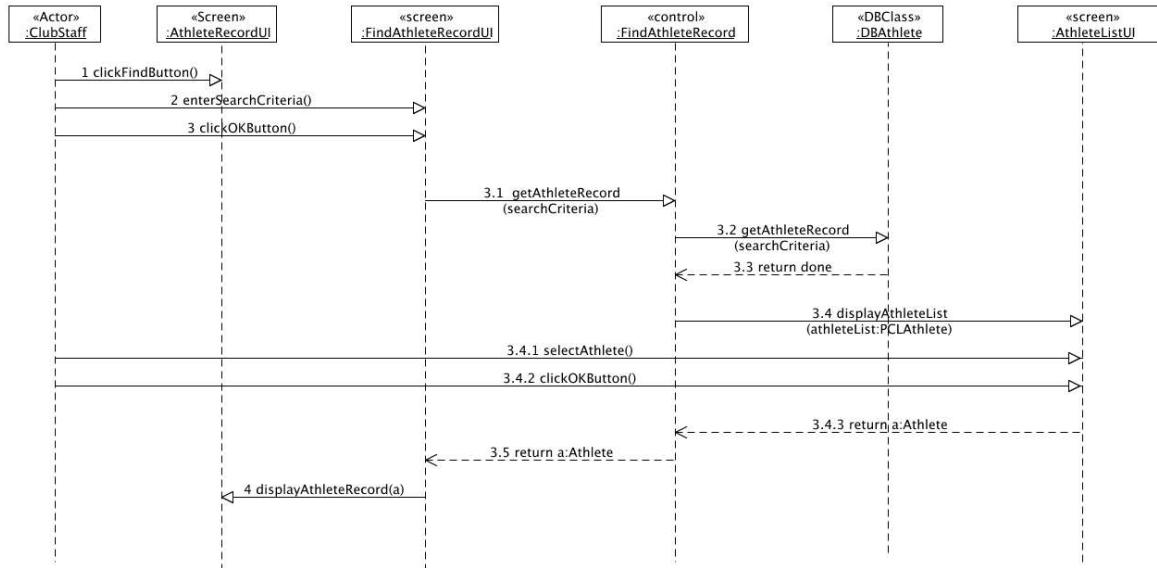


Figure 4.30 DBAthlete JDBC Read

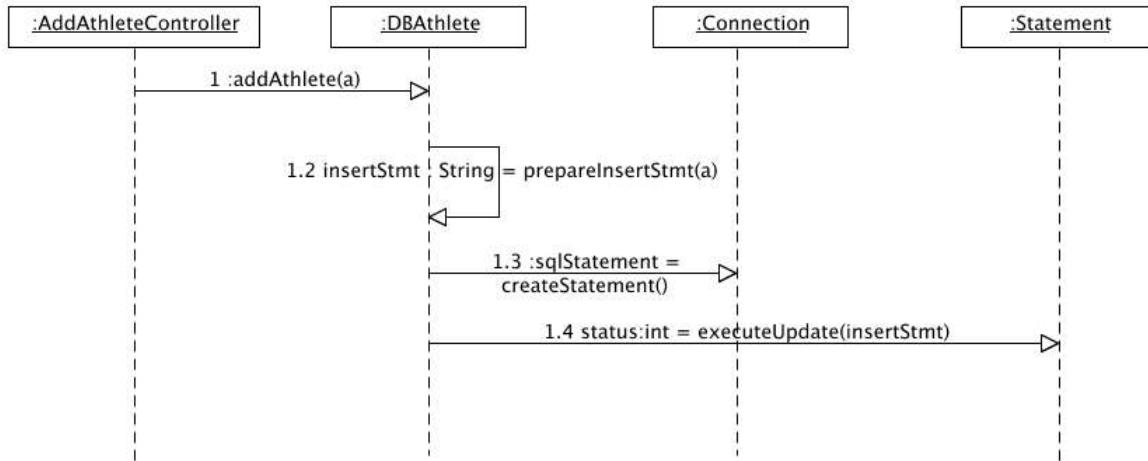


Figure 4.31 DBAthlete JDBC Create

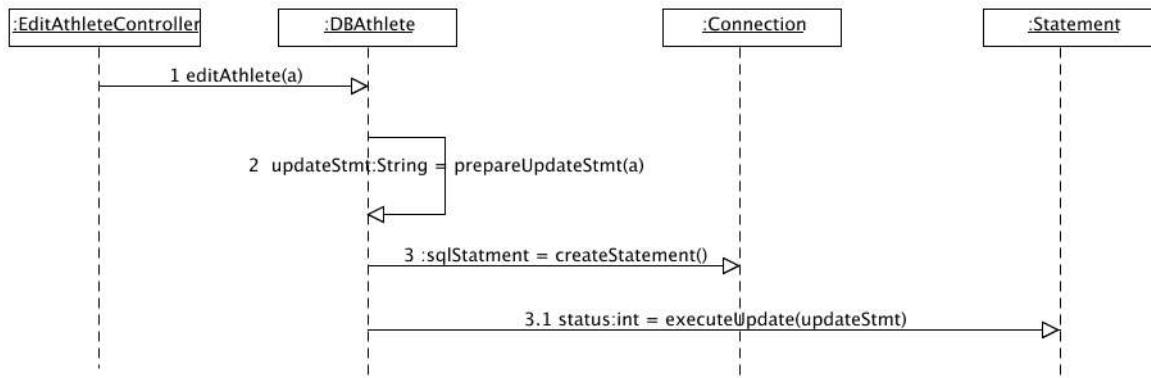


Figure 4.32 DBAthlete JDBC Update

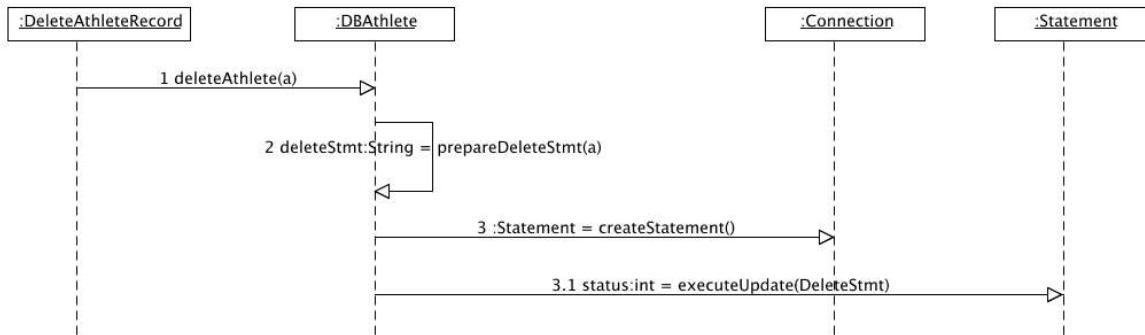


Figure 4.33 DBAthlete JDBC Delete

STEP 3: Document the data design classes.

Identify attributes and operations of each class. Examples are shown in Figure 4.34.

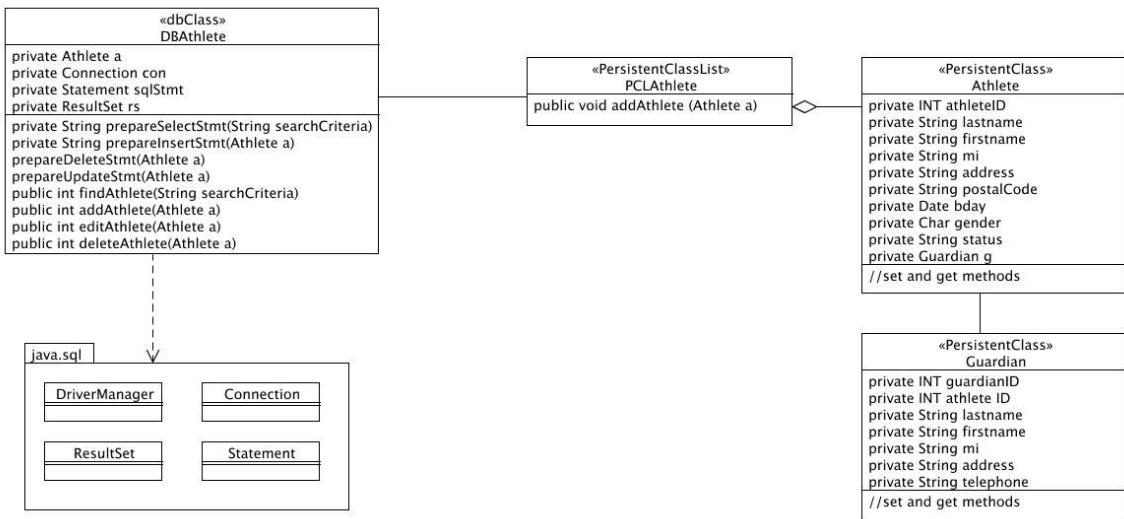


Figure 4.34 Definition of Athlete DBClasses and Persistent Classes

STEP 4: Modify the software architecture to include the data design classes.

It should be on top of the database layer. In simple systems, this becomes part of the business logic layer. You may modify packaging decisions. At this point, entity classes are replaced by the DBClasses, Persistent Classes and Database Design. The modification is shown in Figure 4.35.

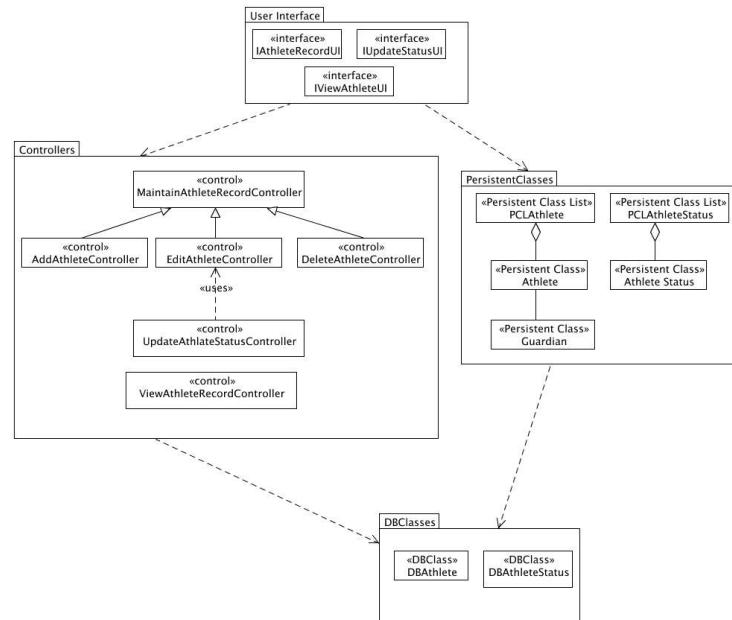


Figure 4.35 Modified Software Architecture from Data Design Model

4.5 Interface Design

The **Interface Design** is concern with designing elements that facilitate how software communicates with humans, devices and other systems that interoperate with it. In this section, we will be concentrating on design elements that interact with people, particularly, pre-printed forms, reports and screens.

Designing good forms, reports and screens are critical in determining how the system will be acceptable to end-users. It will ultimately establish the success of the system. As software engineers, we would want to decrease emphasis on reports and forms and we want to increase emphasis on the screen.

When do we use reports?

- Reports are used for audit trails and controls.
- Reports are used for compliance with external agencies.
- Reports are used if it is to voluminous for on-line browsing.

When do we use forms?

- Forms are used for turnaround documents.
- Forms are used if personnels need to do the transaction but does not have access to workstations.
- Forms are used if they are legally important documents.

When to use Screens?

- Screens are used to query a single record from a database.
- Screens are used for low volume output.
- Screens are used for intermediate steps in long interactive process.

4.5.1 Report Design

Reports can be placed on an ordinary paper, continuous forms, screen-based or microfilm or microfiche. The following gives the steps in designing reports.

Report Design Consideration

1. *Number of Copies and Volume.* There should be a balance between the number of copies generated with the number of pages.

- How many copies of the report will be generated?
- On the average how many pages will be generated?
- Can the user do without the report?
- How often to do the user file a copy? Is it really needed?

2. *Report Generation*

- When do users need the report?
- When is the cut-off period of entering or processing of data before reports are generated?

3. Report Frequency

- How often do the user need the report?

4. Report Figures

- Do the user required reports to generate tables, graphs, charts etc.

5. Media

- Where will the report be produced on CRT Screen, Ordinary Continuous Forms, Preprinted Forms, Bond Paper, Microfilm or microfiche, Storage media, etc.

Report Layout Guidelines

1. Adhere to standard printout data zones of the report layout. The data zones are the headers, detail lines, footers, and summary.

2. Consider the following tips in designing report headers.

- Always include report date and page numbers.
- Clearly describe report and contents.
- Column headers identify the data items listed.
- Align column headers with data item's length.
- Remove corporate name from internal reports for security purposes.

3. Consider the following tips in designing the detail line.

- One line should represent one record of information. Fit details in one line. If more than one line is needed, alphanumeric fields are grouped on the first line and numeric fields are grouped in the next line.
- Important fields are placed on the left side.
- The first five lines and the last five lines are used for the headers and footers. Do not use this space.
- It is a good idea to group the number of details per page by monetary values such as by 10's, 20's and 50's.
- If trends are part of the report, represent them in graphic form such as charts.
- Group related items, such as Current Sales with Year-to-end Sales or Account number, Name and Address.
- For numeric representation such as money, it would be appropriate to include punctuation marks and sign.
- Avoid printing repetitive data.

4. Consider the following tips in designing the report footer.

- Place page totals, grand totals and code-value tables in this area.
- It is also a good place for an indicator that nothing follows.

5. Consider the following tips in designing the summary page.

- Summary pages can be printed at the beginning of the report or at the end.
- Select only one.

Developing the Report Layouts

STEP 1: Define the Report Layout Standards.

As an example, the following is the standard for defining the reports of the *Club Membership Maintenance System*.

- Report heading should contain title of the report, preparation date, club name and page number.
- Body of report should have descriptive data on the left side and numerical data on the right side.
- Bottom of each page provides instructions or explanation of each code while bottom last page contains grand totals and indication of no more pages to follow.
- Important data are positioned on top of the document; then, left to right.
- Codes are always explained when used.
- Dates use MM/DD/YY format.
- Use only logos on all externally distributed documents.

STEP 2: Prepare Report Layouts.

The standards are used as guide for the implementation. For representing data that will appear on the columns of the report use the following format conventions.

Format Convention	Description
9	This means that the data is numeric. The number of 9's indicate the digits. Examples: 9999 – a thousand number 999.99 – a hundred number with two decimal places
A	This is a alphabet string. The number of A's indicates the number of characters.
X	This is an alphanumeric string.
MM/DD/YY	This indicates a date.
HH:MM:SS	This indicates a time.

Table 22: Report Format Conventions

Figure 4.36 is an example of a report layout which follows the data format conventions as specified in Table 22. This is the general report layout for listing athletes based on their status.

Ang Bulilit Liga Club
List Member by Status
Prepared : MM/DD/YYYY
Page 99
 STATUS: XXXXX
MEMBER MEMBER MEMBER
NUMBER LASTNAME FIRSTNAME
 9999 Xxxxxxxxxx Xxxxxxxxxx
9999 Xxxxxxxxxx Xxxxxxxxxx
9999 Xxxxxxxxxx Xxxxxxxxxx
 Total Number XXXXX Members : 9999

Figure 4.36 Sample Report Layout

4.5.2 Forms Design

Forms are usually used for input when a workstation is not available. It is sometimes used as a turnaround document. The following are the steps in designing forms.

Forms Layout Guidelines

1. Instructions should appear on the form except when the same person fills it over and over again. General instructions are placed on top. They should be brief. Specific instructions should be placed just before the corresponding items.
2. Use familiar words. Avoid using "not", "except" and "unless". Use positive, active and short sentences.
3. The answer space should be enough for user to comfortably enter all necessary information. If the form will be used for a typewriter, 6 or 8 lines per inch is a good measurement. For handwritten answer spaces, 3 lines per inch with 5 to 8 characters per inch is a good measurement.

4. Use the following guidelines for typefaces of the fonts.
 - Gothic
 - Simple, squared off, no serifs
 - Easy to read, even when compressed
 - Capital letters are desirable for headings
 - Italic
 - Has serifs and a distinct slant
 - Hard to read in large amounts of long letters
 - Good for printing out a work or phrase
 - Roman
 - Has serifs but does not slant
 - Best for large quantities
 - Good for instructions
5. Lower case print is easier to read than uppercase. However, for small font size, they should not be used. Use upper letters to call attention to certain statements.

Developing the Form Layouts

STEP 1: Define the standards to be used for designing forms.

An example of a standard for defining forms is listed below.

- Paper size should not be bigger than 8 1/2" x 11".
- Colors Schemes
 - White copies goes to the applicant.
 - Yellow copies goes to the club staff.
 - Pink copies goes to the coach.
- Important data should be positioned on left side of document.
- Dates should use DD/MM/YY Format.
- Logos are only used for all externally distributed documents.
- Heading should include Page Number and Title.
- Binding should use 3-hole paper.
- Ruling should follow 3 lines per inch.

STEP 2: Prepare Form Samples.

Using the standard format defined in step 1, design the layout of the forms. Redesign if necessary.

4.5.3 Screen and Dialog Design

The boundary classes are refined to define screens that the users will be using and the interaction (dialog) of the screens with other components of the software. In general, there are two metaphors used in designing user-interface are used.

1. **Dialog Metaphor** is an interaction between the user and the system through the use of menus, function keys or entering of a command through a command line. Such metaphor is widely used in the structured approach of developing user-interfaces that are shown on character or text based terminals.
2. **Direct Manipulation Metaphor** is an interaction between user and the system using graphical user interfaces (GUIs). Such interfaces are event-driven, i.e., when a user, let say clicks a button (event), the system will respond. It gives the user an impression that they are manipulating objects on the screen.

For object-oriented development which uses visual programming integrated environments, we use the direct manipulation metaphor. However, when designing interfaces, several guidelines should be followed.

Dialog and Screen Design Guidelines

1. Stick to the rule: ONE IDEA PER SCREEN. If the main idea changes, another screen must appear.
2. Standardize the screen layout. This would involve defining style guides that should be applied to every type of screen element. Style guides support consistency of interface. Consistency helps users learn the application faster since functionality and appearance are the same across different parts of the application. This applies to commands, the format of the data as it is presented to the user such as date formats, layout of the screens etc. Several software vendors provide style guides that you may use. Common style guides are listed below:
 - Java Look and Feel Design Guidelines (<http://java.sun.com>)
 - The Windows Interface Guidelines for Software Design (<http://msdn.microsoft.com>)
 - Macintosh Human Interface Guidelines (<http://developer.apple.com>)
3. When presenting data items to users, always remember that:
 - They should have a screen label or caption, and they should be the same throughout the entire application. As an example, you have a data item that represents a student identification number. You may decide to use 'Student ID' as its caption. For every screen that displays this item, use the value 'Student ID' as the label; do not change it to 'Student Number' or 'Student No.'.
 - As much as possible, they should be located on the same place in all screens, particularly, for the data entry screens.
 - They should be displayed in the same way such as in the case of presenting dates. If the format of the dates follows 'MM-DD-YYYY' format, then all dates should be displayed in this format. Do not change it.

4. For controllers that require a longer time to process, keep user informed of what is going on. Use "XX% completed" messages or line indicators.
5. If possible, it would be wise to always inform the user of the next step. Wizard-type dialog design is always a good approach.
6. Provide meaningful error messages and help screens. AVOID PROFANITY!

Developing the Screen and Dialog Design

STEP 1: Prototyping the User-interface.

A **prototype** is a model that looks, and to some extent, behaves like the finished product but lacking in certain features. It is like an architect's scale model of a new building. There are several types of prototypes.

1. **Horizontal prototypes** provide a model of the user interface. It deals only with the presentation or view layer of the software architecture. Basically, this type of prototype shows the screen sequencing when the user interacts with the software. No functionality is emulated.
2. **Vertical prototypes** takes one functional subsystem of the whole system and develops it through each layer: user-interface, some logical functionality and database.
3. **Throwaway Prototypes** are prototypes that are discarded later after they have served their purpose. This type of prototype is done during the requirements engineering phase. As much as possible, we avoid creating such prototypes.

The use of **Visual Programming Environment** can support the development prototypes. NetBeans and Java Studio Enterprise 8 provides a means to visually create our prototypes.

To demonstrate step one, assume we are building a screen prototype for maintaining an athlete record. The main screen, which is named **Athlete** (Figure 4.37), allows us to manage a single athlete record. When a user clicks the **Find Button**, the **Find an Athlete** (Figure 4.38) screen appears. This screen is used to specify the search criteria that determines which athlete records that are retrieved from the database. When the criteria has been entered and the **OK Button** is clicked, the **Athlete List** (Figure 4.38) screen will be displayed containing the list of athletes that satisfies the search criteria. The user can select the record by highlighting the name. When the **OK Button** is clicked, it will return to the **Athlete** screen where the attributes of the chosen athlete are displayed in the appropriate screen elements.

Athlete

Membership No:	3456	Status	Active ▾	<input type="button" value="Find..."/>												
Last Name	<input type="text" value="De la Cruz"/>			<input type="button" value="Save"/>												
First Name	<input type="text" value="Johnny"/>			<input type="button" value="Delete"/>												
Middle Initial	<input type="text" value="A."/>			<input type="button" value="Cancel"/>												
Address	<input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/>		Postal Code	<input type="text" value="2619"/>												
Date of Birth	<input type="text" value="9/24/1998"/>															
Gender	<input checked="" type="radio"/> Male <input type="radio"/> Female															
<p>Guardian:</p> <table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;">Last Name</td> <td><input type="text" value="De la Cruz, Sr."/></td> </tr> <tr> <td>First Name</td> <td><input type="text" value="Johnny"/></td> </tr> <tr> <td>Middle Initial</td> <td><input type="text" value="A."/></td> </tr> <tr> <td>Address</td> <td><input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/></td> </tr> <tr> <td>Telephone</td> <td><input type="text" value="555-9895"/></td> </tr> <tr> <td>Postal Code</td> <td><input type="text" value="2619"/></td> </tr> </table>					Last Name	<input type="text" value="De la Cruz, Sr."/>	First Name	<input type="text" value="Johnny"/>	Middle Initial	<input type="text" value="A."/>	Address	<input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/>	Telephone	<input type="text" value="555-9895"/>	Postal Code	<input type="text" value="2619"/>
Last Name	<input type="text" value="De la Cruz, Sr."/>															
First Name	<input type="text" value="Johnny"/>															
Middle Initial	<input type="text" value="A."/>															
Address	<input type="text" value="Lot 24 Block 18, St. Andrewsfield Quezon City"/>															
Telephone	<input type="text" value="555-9895"/>															
Postal Code	<input type="text" value="2619"/>															

Figure 4.37 Screen Prototype of Athlete Record

Find An Athlete

Membership No:		Status	Active ▾	<input style="border: 1px solid #0070C0; border-radius: 15px; padding: 5px; background-color: white; color: #0070C0; font-weight: bold; font-size: 14px; margin-right: 10px;" type="button" value="OK"/> <input style="border: 1px solid #0070C0; border-radius: 15px; padding: 5px; background-color: white; color: #0070C0; font-weight: bold; font-size: 14px;" type="button" value="Cancel"/>
Last Name				
First Name				
Middle Initial				
Date of Birth	9/1/1998	To	9/30/1998	
Gender	<input checked="" type="radio"/> Male	<input type="radio"/> Female		

Athlete List

Athlete ID	Last Name	First Name	MI
3303	Armstrong	Mark	L.
3307	Brown	Lione	G.
3312	Ferran	Julius	R.
3318	Jones	Simon	R.
3323	Napoli	Ray	F.
3326	Robinson	James	S.
3336	Schmidt	John	C.
3343	Smith	Allan	A.
3344	Smith	Jacob	F.
3347	Smith	Kenny	S.
3348	Smith	Leo	A.

Figure 4.38 Screen Prototypes of FindAthleteRecordUI and AthleteListUI

In the following steps, screens will be treated as single objects. In reality, they may well be an instance of a subclass such as Dialog. As an example, take a look at Figure 4.39.

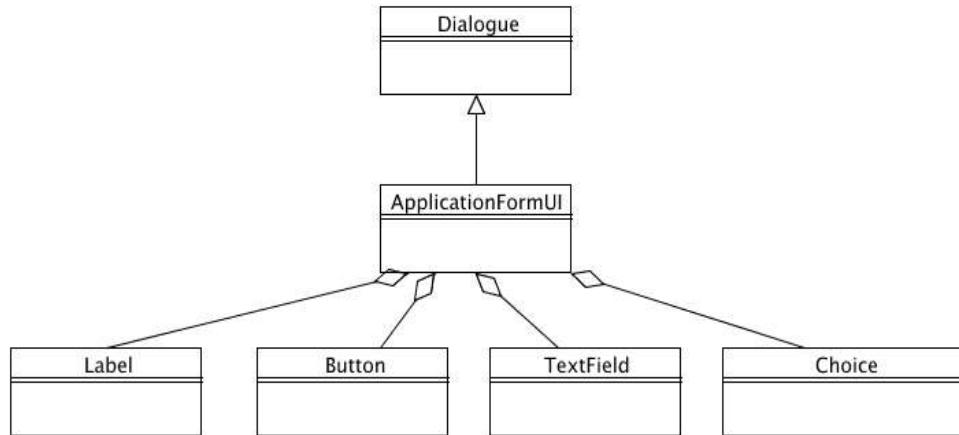


Figure 4.39 Dialog Class Diagram

In Java, you can use the Java AWT/SWING component. They are imported in the class that use them using the following Java statements:

```
import java.awt.*;  
import java.awt.swing.*;
```

STEP 2: For each screen, design the screen class.

This step would require us to model the screen as a class. As an example, consider Figure 4.40 which provides the class definition for the **Athlete** screen. The name of the class is **AthleteRecordUI**. It has a <<screen>> stereotype. In this example, we assume that we will be using the JAVA Abstract Windowing Toolkit (AWT) for support components. You also need to model the screens **Find an Athlete** and **Athlete List** as classes; they are named **FindAthleteRecordUI** and **AthleteListUI** respectively. Package all the screens together to form the subsystem. This subsystem should implement the **IAthleteRecordUI** interface for the boundary class. See Figure 4.41.

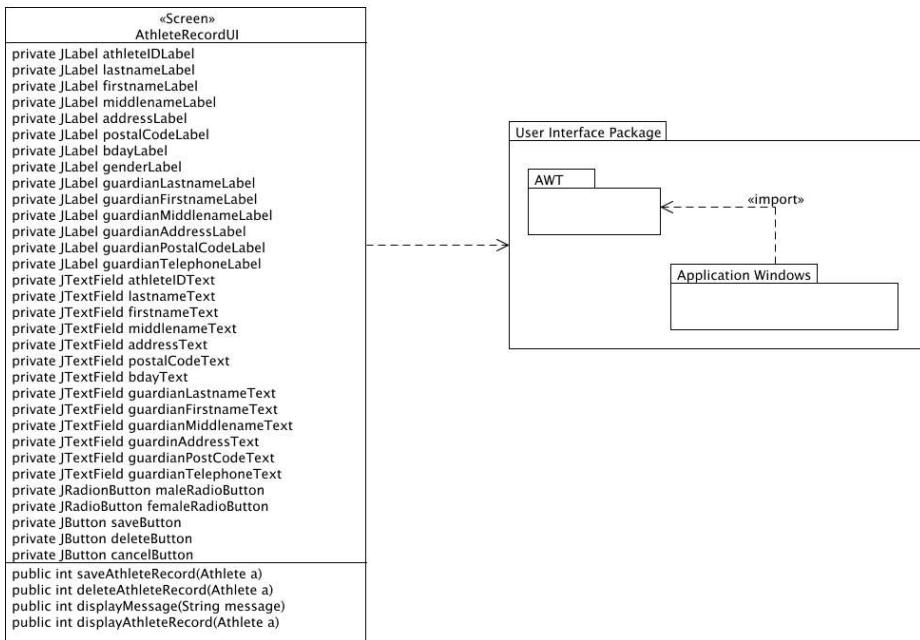


Figure 4.40 Class Definition of AthleteRecordUI

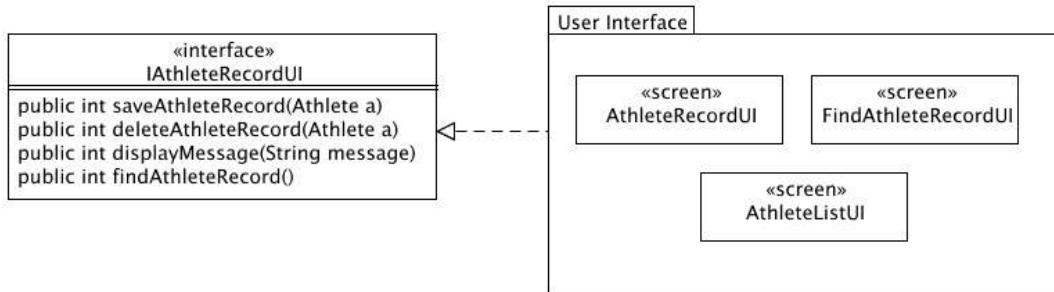


Figure 4.41: IAthleteRecordUI Interface and Subsystem Definition

STEP 3: For each screen class, model the behavior of the screen with other classes.

The behavior of the screen class relative to the other classes is known as the dialog design. The collaboration diagrams and sequence diagrams are used in this step. As in our working example, there are three main functionality for managing an athlete record that must be supported, namely, add, edit and delete an athlete record. Consider the collaboration diagrams as modeled in Figure 4.42. The first collaboration diagram models how the system manages an athlete record; and, it consists of the **AthleteRecordUI**, **MaintainAthleteRecord** and **DBAthlete** classes. It is elaborated by the second collaboration diagram. Additional classes are identified and defined to

support retrieval of an athlete record from the database as needed by the edit and delete athlete record functionality. These classes are **FindAthleteRecordUI**, **FindAthleteRecord** and **AthleteListUI**.

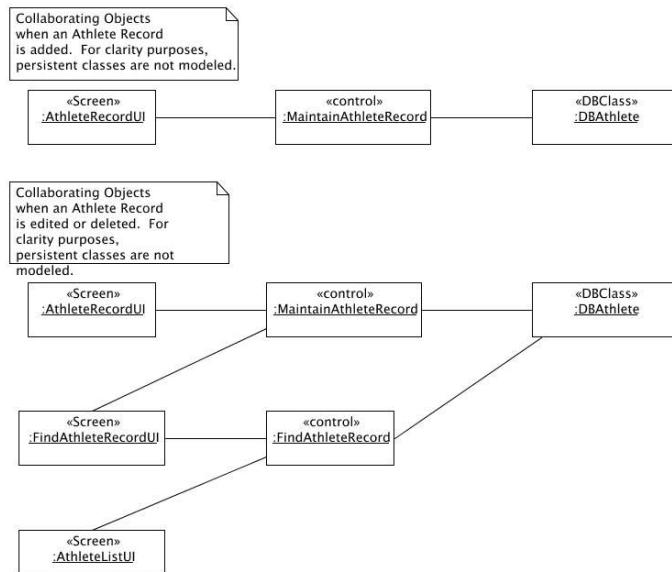


Figure 4.42 Modified Collaboration Diagram with AthleteRecordUI

Once the collaborating objects are identified, we need to model their behavior when the functionalities (add, edit and delete athlete record) are performed. We use the Sequence Diagram to model this behavior. Figure 4.43 shows the modified sequence diagram of adding an athlete record (This sequence diagram is an elaboration of DBAthlete JDBC Create of adding an athlete record as was discussed in the Data Design Section.)

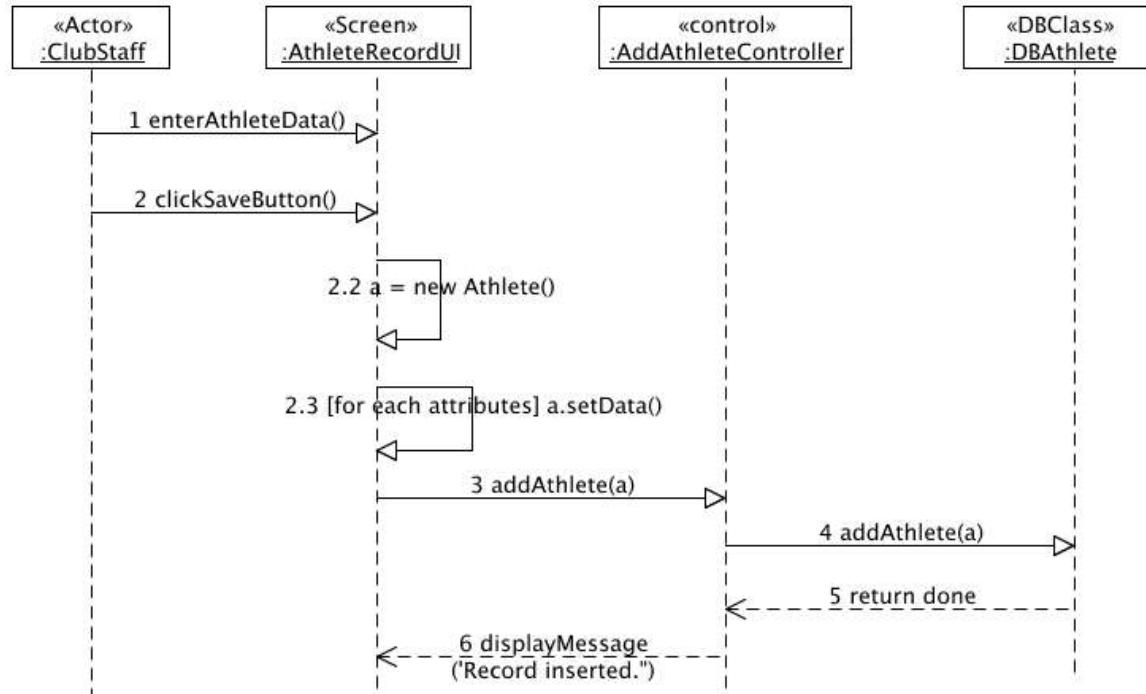


Figure 4.43 Modified Sequence Diagram of Adding an Athlete Record

Both the sequence diagrams for editing and deleting an athlete record would require a retrieval of a record from the database. Prior to editing and deleting, the sequence diagram shown in Figure 4.44 must be performed. (This sequence diagram is an elaboration of DBAthlete JDBC Read of an athlete record as it was discussed under the Data Design Section).

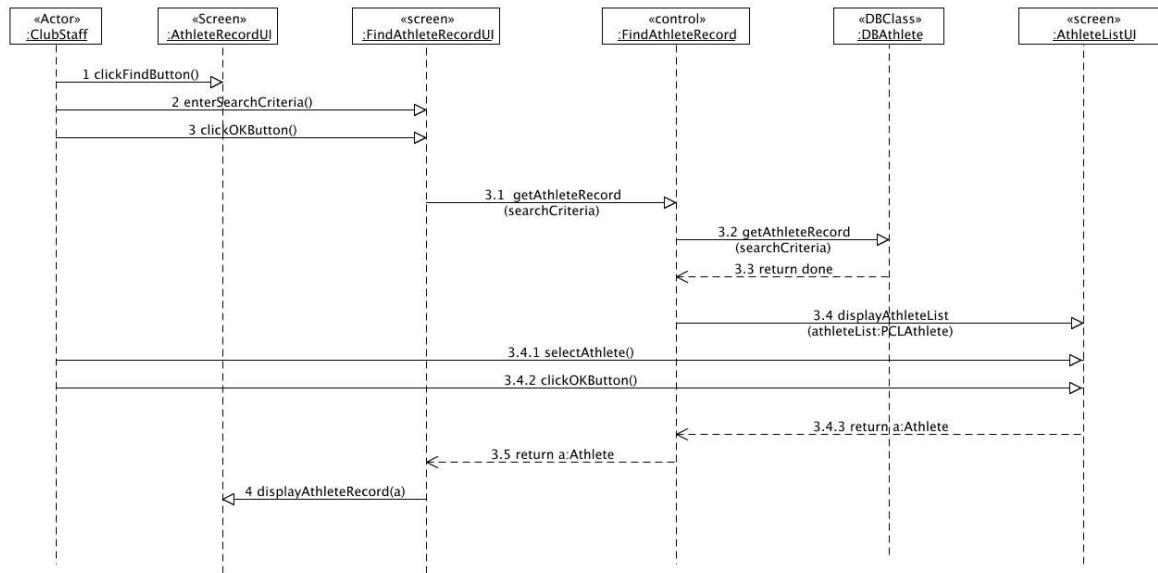


Figure 4.44 Modified Sequence Diagram of Retrieving an Athlete Record

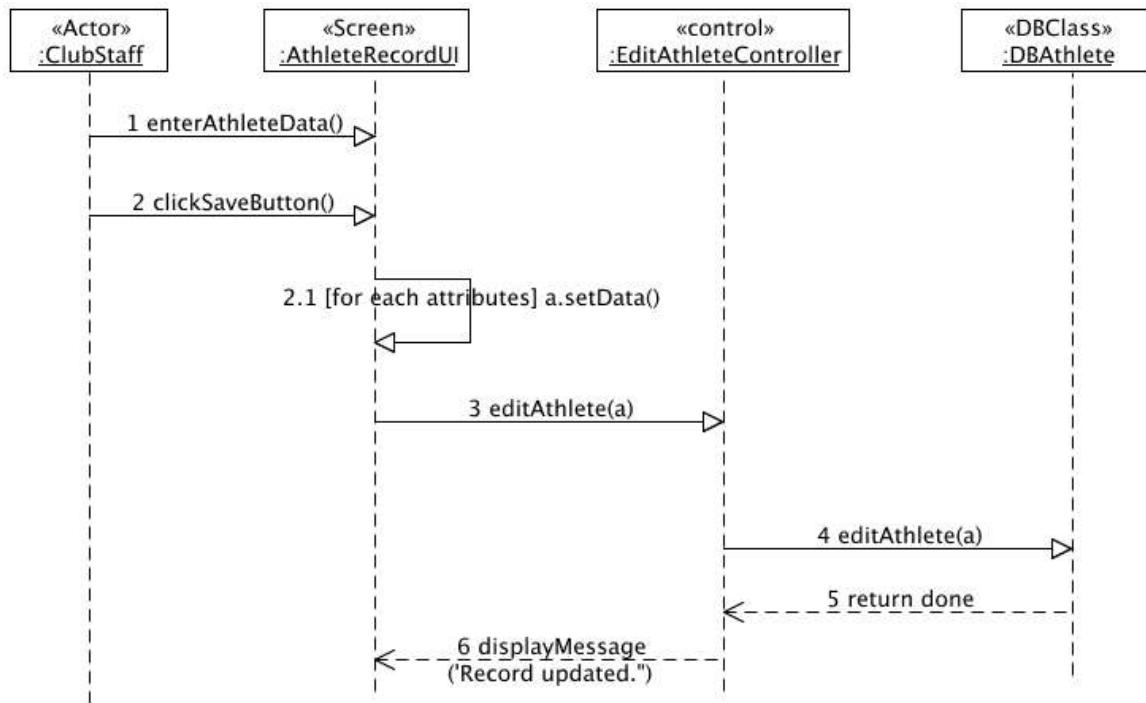


Figure 4.45 Modified Sequence Diagram of Editing an Athlete Record

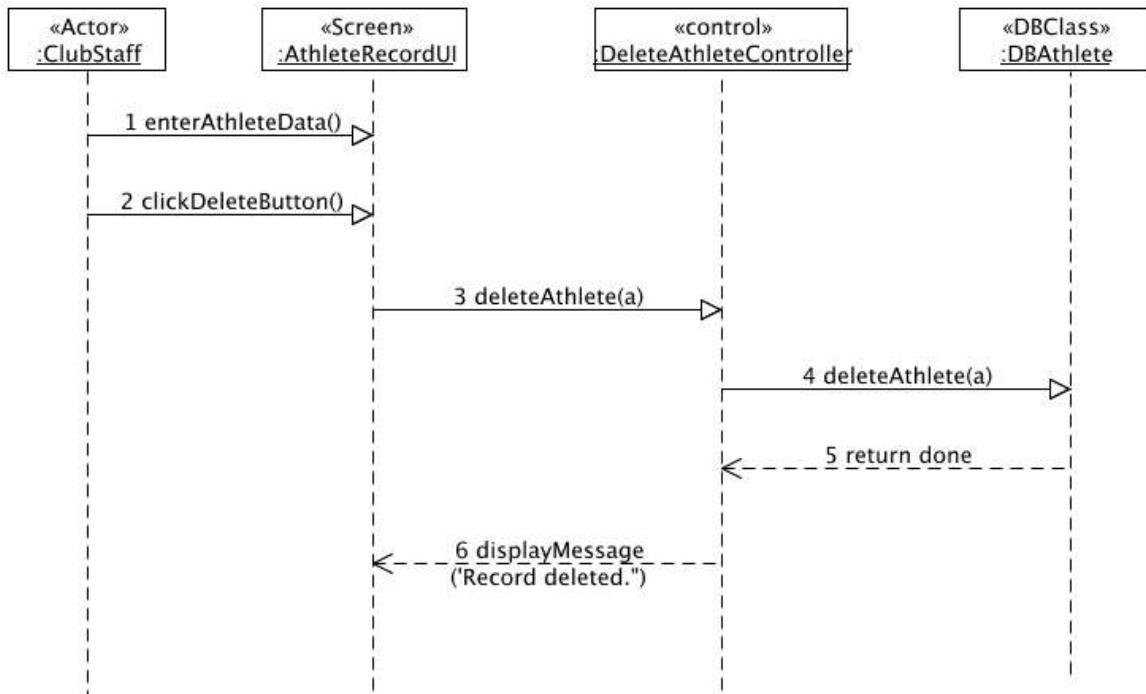


Figure 4.46 Modified Sequence Diagram of Deleting an Athlete Record

The modified sequence diagram for editing and deleting an athlete record are shown in Figure 4.45 and Figure 4.46 respectively. (They are also an elaboration of the DBAthlete JDBC Update and DBAthlete JDBC Delete respectively).

STEP 4. For each screen class, model its internal behavior.

The State Chart Diagram is used to model the internal behavior of the screen. It basically models the behavior of the screen when a user does something on the screen elements such as clicking the **OK Button**.

Describing the State Chart Diagram

The State Chart Diagram shows the possible states that an object can have. It also identifies the events that causes the state change. It shows how the object's state changes as a result of events that are handled by the object. Figure 4.47 shows the basic notation of the diagram.

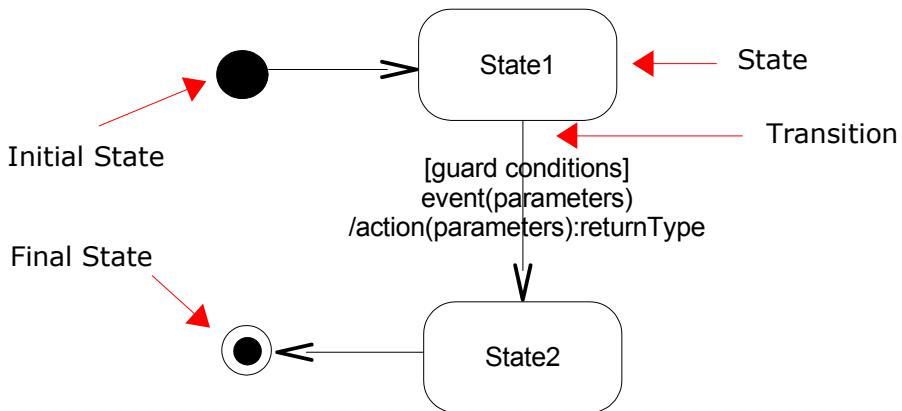


Figure 4.47 State Chart Diagram Basic Notation

The **initial state**, as represented by a solid filled circle, signifies the entry point of the transition. An object cannot remain in its initial state but needs to move to a named **state**. The state is represented as a rounded rectangular object with the state name inside the object. A **transition** is represented by an arrow line. A transition should have a **transition string** that shows the event that causes the transition, any guard conditions that needs to be evaluated during the triggering of the event, and the action that needs to be performed when the event is triggered. The event should be an operation defined in the class. The **final state**, as represented by the bull's eye, signifies the end of the diagram.

Similar with other diagrams in UML, there is an extended or enhanced notation that can be used when defining the state of a class. Figure 4.48 shows the these notations.

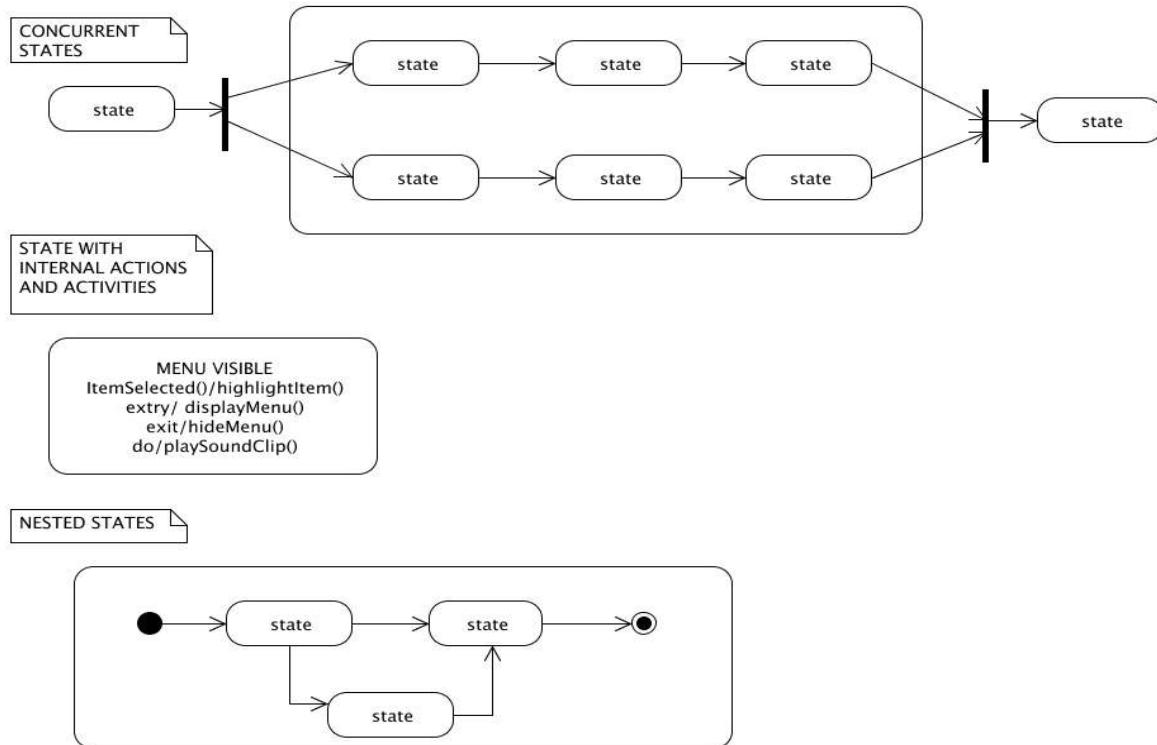


Figure 4.48 State Chart Diagram Enhanced Notation

Concurrent states means that the behavior of the object can best be explained by regarding the product as two distinct set of substates, each state of which can be entered and exited independently of substates in the other set. **Nested states** allow one to model a complex state through defining different level of detail. States can have internal actions and activities. Internal actions of the state defines what operation should be perform upon entering the state, what operation should be perform upon exiting the state, and what operation should be perform while the object is in that state.

Figure 4.49 shows the state chart diagram of the **AthleteRecordUI** class. When the screen is first displayed, it will show an initialized screen where most of the text boxes are empty. Some fields may have default values such as **Gender** with a default value of 'Male' and **Status** with the default value as 'NEW'. This state is the **With Default Value** state. When a user enters a value on any text field (this triggers an **enterTextField()** event), the screen's state will move to the **With Entered Value** state where the entered value is reflected on the screen element. When a user clicks the **Cancel Button** (which triggers the **clickCancelButton()** event), the screen state will move to the **Show Exit Confirmation** state. In this state, a message dialog box will be displayed asking the user if he or she really wants to exit. If the user answers YES, we exit the screen. Otherwise, we go back to the previous state.

Figure 4.50 shows the **FindAthleteRecordUI** state chart diagram and Figure 4.51 shows the **AthleteListUI** state chart diagram.

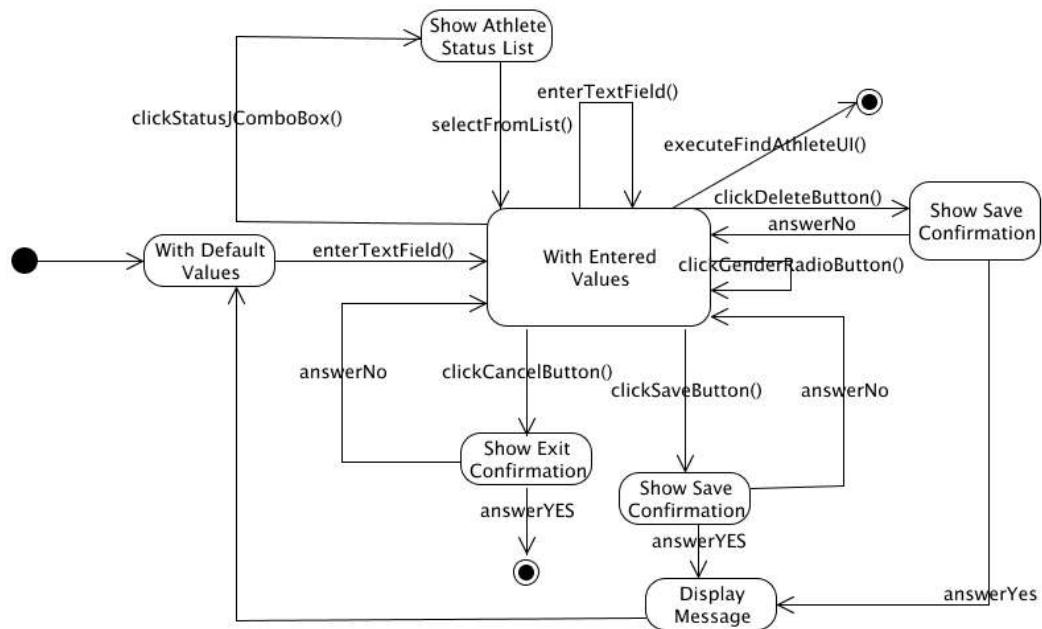


Figure 4.49 AthleteRecordUI State Chart Diagram

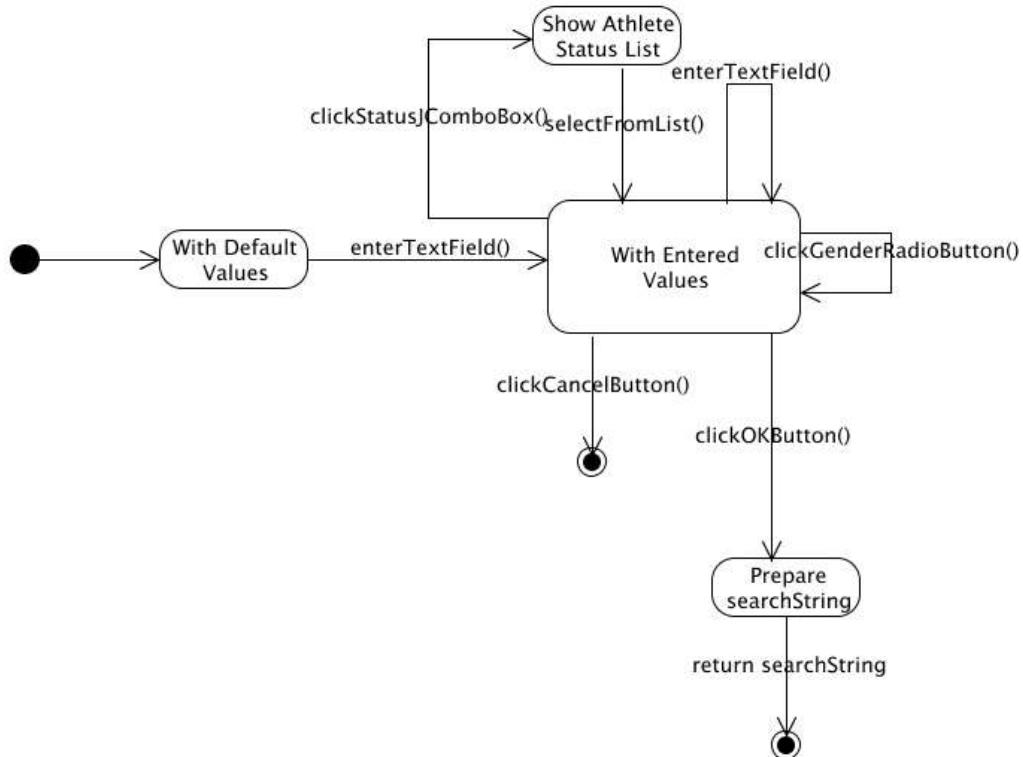


Figure 4.50 FindAthleteRecordUI State Chart Diagram

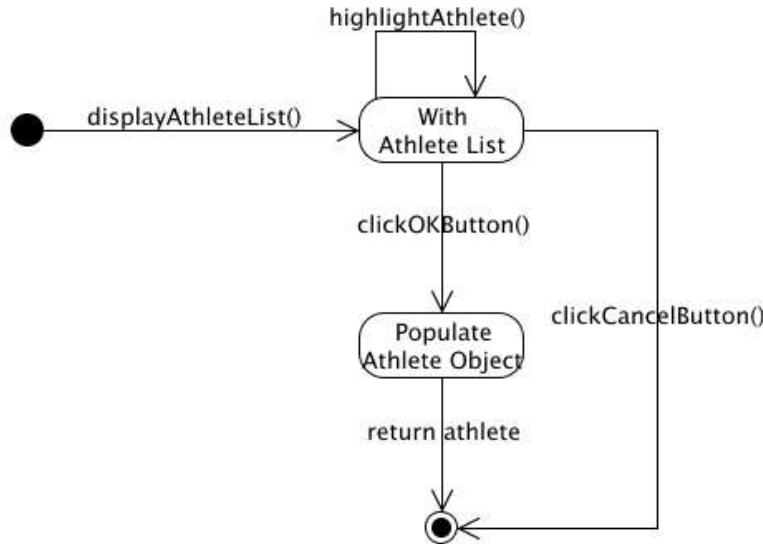


Figure 4.51 AthleteListUI State Chart Diagram

STEP 5: Document the screen classes.

The documentation is similar to documenting DBClasses and Persistent Classes. Identify all the attributes and operations of the screen. Figure 4.52 shows the class definition of all screens.

«Screen» AthleteRecordUI	«Screen» FindAthleteRecordUI	«Screen» AthleteListUI
<pre> private JLabel athleteIDLabel private JLabel statusLabel private JLabel lastnameLabel private JLabel firstnameLabel private JLabel middlenameLabel private JLabel addressLabel private JLabel postalCodeLabel private JLabel bdayLabel private JLabel genderLabel private JLabel guardianLastnameLabel private JLabel guardianFirstnameLabel private JLabel guardianMiddleNameLabel private JLabel guardianAddressLabel private JLabel guardianPostalCodeLabel private JTextField guardianTelephoneLabel private JTextField athleteIDText private JComboBox statusComboBox private JTextField lastnameText private JTextField firstnameText private JTextField middlenameText private JTextField addressText private JTextField postalCodeText private JTextField bdayText private JTextField guardianLastnameText private JTextField guardianFirstnameText private JTextField guardianMiddleNameText private JTextField guardianAddressText private JTextField guardianPostalCodeText private JRadioButton maleRadioButton private JRadioButton femaleRadioButton private JButton saveButton private JButton deleteButton private JButton cancelButton public int saveAthleteRecord(Athlete a) public int deleteAthleteRecord(Athlete a) public int displayMessage(String message) public int findAthleteRecord() </pre>	<pre> private JLabel athleteIDLabel private JLabel statusLabel private JLabel lastnameLabel private JLabel firstnameLabel private JLabel middlenameLabel private JLabel addressLabel private JLabel postalCodeLabel private JLabel bdayLabel private JLabel toLabel private JLabel genderLabel private JTextField athleteIDText private JComboBox statusComboBox private JTextField lastnameText private JTextField firstnameText private JTextField middlenameText private JTextField addressText private JTextField postalCodeText private JTextField fromBdayText private JTextField toBdayText private JRadioButton maleRadioButton private JRadioButton femaleRadioButton private JButton okButton private JButton cancelButton private String findAthleteRecord() public void clickOKButton() public void clickCancelButton() </pre>	<pre> private JLabel athleteListingLabel private JTable athleteList private JButton okButton private JButton cancelButton private void populateTable() public void clickOKButton() public void clickCancelButton() </pre>

Figure 4.52 Refined Screen Class Definition

It would also help to list the action in a table. This table is called the **Event-Action** Table. This is used if the state chart is complex. In the point of view of a programmer, this table is easier to use than a state chart labelled with actions. It should help in validating the state chart and to test the code once it is implemented. The following are the columns of the Event-Action Table:

- The current state of the object being modeled. For easy reference, we number the states.
- The event that can take place.
- The actions associated with the combination of state and event.
- The next state of the object after the event has taken place. If more than one state variable is used, another column is used.

Figure 4.53 shows the state chart diagram of the **AthleteRecordUI** screen with the states numbered.

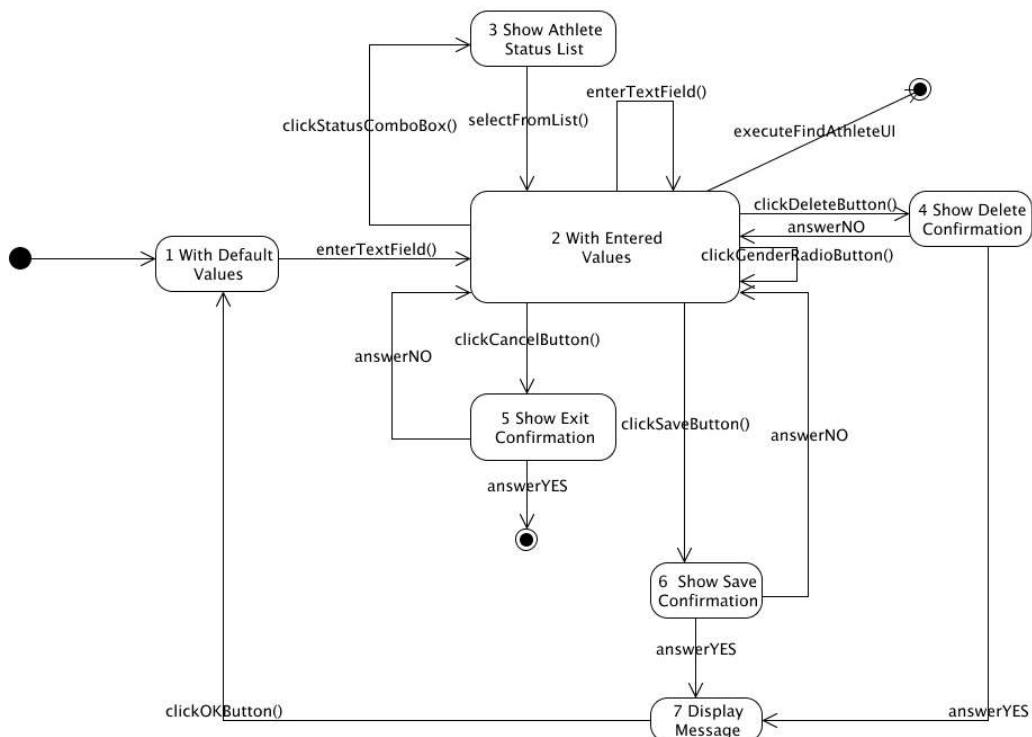


Figure 4.53: AthleteRecordUI State Chart Diagram

Table 23 shows the Event-Action of this screen.

Current State	Event	Action	Next State
-	Access to the AthleteRecordUI Screen	Display an empty Athlete Screen except for identified screen elements with default values	1
1	enterTextField()	Display the typed value inside the text field	2
2	clickStatusComboBox()	Shows a list of athlete status	3
3	selectFromList()	Display the selected athlete status in the display field	2
2	enterTextField()	Display the typed value inside the text field	2
2	clickCancelButton()	Show Exit Confirmation Dialog Box	5
5	AnswerYES	Exits the Athlete screen without saving any data displayed on the screen	-
5	AnswerNO	Close the Exit Confirmation Dialog Box	2
...

Table 23: Event-Action Table of AthleteRecordUI Screen

STEP 6: Modify the software architecture.

The boundary subsystems are replaced by the classes for the screens or user interfaces. If additional control classes are defined, add them to the architecture. Figure 4.54 shows the modified software architecture.

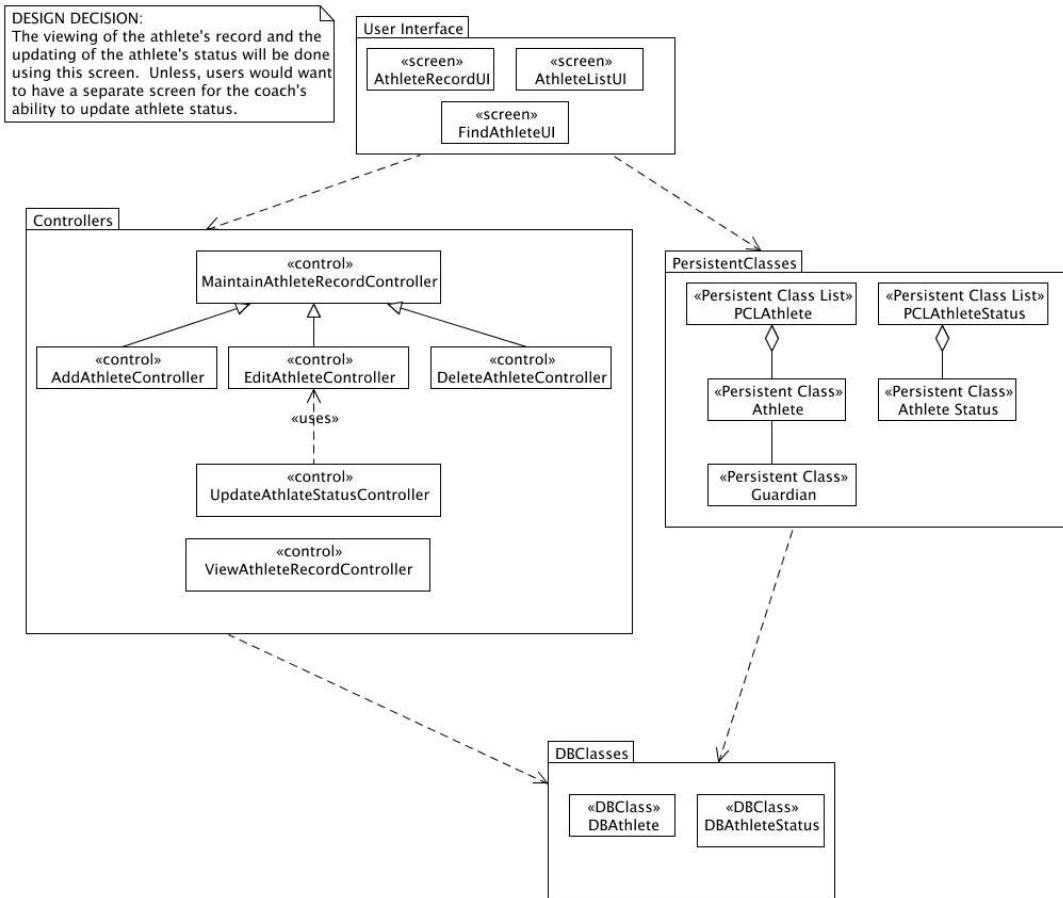


Figure 4.54: Modified Software Architecture from User Interface Design Elaboration

4.6 Component-level Design

The **Component-level Design** defines the data structure, algorithms, interface characteristics, and communication mechanism allocated to each software component. A **component** is the building block for developing computer software. It is a replaceable and almost independent part of a software that fulfills a clear function in the context of a well-defined architecture.

In object-oriented software engineering, a component is a set of collaborating classes that fulfills a particular functional requirement of the software. It can be independently developed. Each class in the component should be fully defined to include all attributes and operations related to the implementation. It is important that all interfaces and messages that allow classes within the component to communicate and collaborate be defined.

4.6.1 Basic Component Design Principles

Four basic principles are used for designing components. They are used to guide the software engineer in developing components that are more flexible and amenable to change and reduces the propagation of side effects when change do occur.

1. *The Open-Close Principle.* When defining modules or components, they should be open for extension but they should not be modifiable. The software engineer should define a component such that it can be extended without the need to modify the internal structure and behavior of the component. Abstractions in the target programming language supports this principles.
2. *Liskov Substitution Principle.* The derived class or subclass can be a substitute for the base class. If a class is dependent on a base class, that class can use any derived class as a substitute for the base class. This principle enforces that any derived class should comply with any implied contract between the base class and any component that uses it.
3. *Dependency Principle.* Classes should depend on abstraction; not on concretions. The more a component depends on other concrete components, the more difficult it can be extended.
4. *Interface Segregation Principle.* Software engineers are encouraged to develop client-specific interface rather than a single general purpose interface. Only operations specific to a particular client should be defined in the client-specific interface. This minimizes inheriting irrelevant operations to the client.

4.6.2 Component-level Design Guidelines

This guideline is applied to the design of the component, its interface, its dependencies and inheritance.

1. Component

- Architectural component names should come from the problem domain and is easily understood by the stakeholders, particularly, the end-users. As an example, a component called **Athlete** is clear to any one reading the component.
- Implementation component name should come from the implementation

specific name. As an example, **PCLAthlete** is the persistent class list for the athletes.

- Use stereotypes to identify the nature of the component such as <<table>>, <<database>> or <<screen>>.

2. Interfaces

- The canonical representation of the interface is recommended when the diagram becomes complex.
- They should flow from the left-hand side of the implementing component.
- Show only the interfaces that are relevant to the component under consideration.

3. Dependencies and Inheritance

- Dependencies should be modeled from left to right.
- Inheritance should be modeled from bottom (subclass or derived class) to top (superclass or base class).
- Component interdependence are modeled from interface to interface rather than component to component.

4.6.3 Component Diagram

The Component Diagram is used to model software components, their interface, dependencies and hierarchies. Figure 4.55 shows the notation of this diagram.

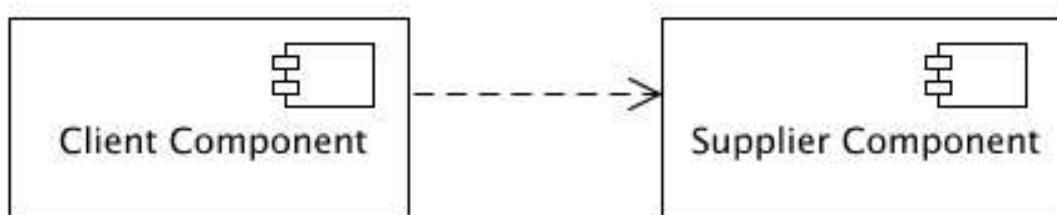


Figure 4.55 Component Diagram Notation

A **component** is represented by a rectangular box with the component symbol inside. The **dependency** is depicted by the broken arrow. The dependency can be named. The Client Component is dependent on the Supplier Component.

4.6.4 Developing the Software Component

All classes identified so far should be elaborated including the screens, and data design classes. In this section, the **MaintainAthleteRecordController** will be used as an

example but the component design should be done for all classes.

The **MaintainAthleteRecordController** is an abstract class that represents, in general, behavior of maintaining an athlete record. It is extended by a type of maintenance—adding a record, editing a record or deleting a record. Figure 4.56 shows the hierarchy of the type of athlete record maintenance that is modeled during the requirements engineering phase. The relationship is “is-a-kind” relationship; **AddAthleteController** is a kind of **MaintainAthleteRecordController**.

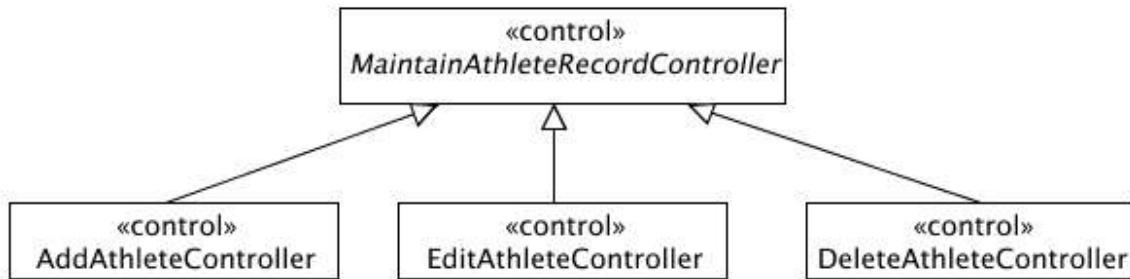


Figure 4.56 *MaintainAthleteRecordController Hierarchy*

Developing the component-level design involves the following steps.

STEP 1: Refine all classes.

The classes are refined in such a way that it can be translated into a program. Refactoring may be required before proceeding. This step involves a series of sub-steps.

STEP 1.1: If necessary, redefine the sequence and collaboration diagrams to reflect the interaction of the current classes.

STEP 1.2: Distribute operations and refine operation signature of the class. Use data types and naming conventions of the target programming language. Figure 4.57 shows the elaboration of the operation signature of the **AddAthleteController**. A short description is attached to the operation.

STEP 1.3: Refine attributes of each class. Use data types and naming conventions of the target programming language. Figure 4.57 shows the elaboration of the attributes of the **AddAthleteController**. A short description is attached to the attribute.

STEP 1.4: Identify visibility of the attributes and operations. The visibility symbols are enumerated in Table 24.

Visibility Symbols		Descriptions
+ or public	PUBLIC.	The attribute or operation is directly accessible to an instance of a class.
- or private	PRIVATE.	The attribute or operation is only accessible to the instance of the class that includes it.
# or protected	PROTECTED.	The attribute or operation may be used either by instances of the class that includes it or by a subclass of the class.
~	PACKAGE.	The attribute or operation is only accessible only be instances of a class in the same package.

Table 24: Attribute or Operation Visibility

Visibility symbols are placed before the name of the attribute or operation. See Figure 4.57 as an example.

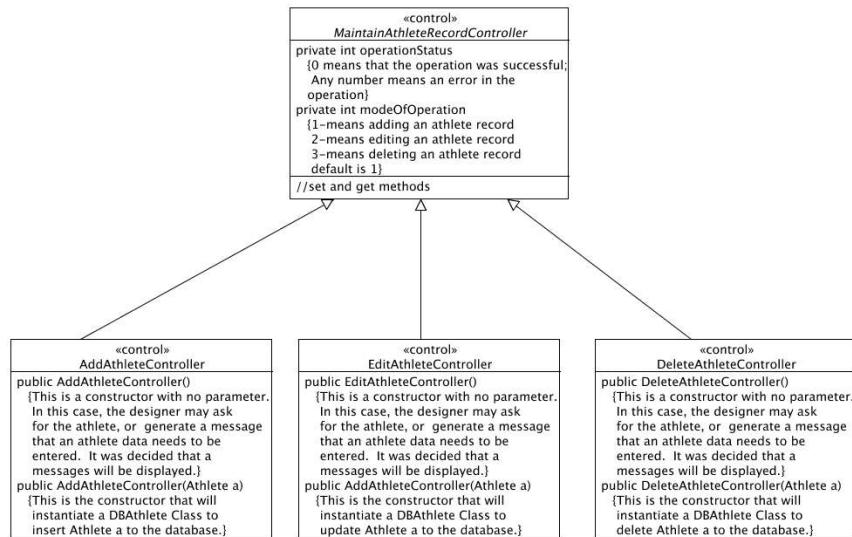


Figure 4.57 Controller Class Definition

STEP 1.5: Document the class. Particularly, identify the pre-conditions and post-conditions. **Pre-conditions** are the conditions that must exists before the class can be used. **Post-conditions** are the conditions that exists after the class is used. Also, include the description of the attributes and operations.

STEP 2: If necessary, refine the packaging of the classes.

In our example, it was decided that no repackaging is necessary.

STEP 3. Define the software components.

Define the components using the component diagram. Figure 4.58 shows the software components of the **Club Membership Maintenance System**. It does not include components that are reused.

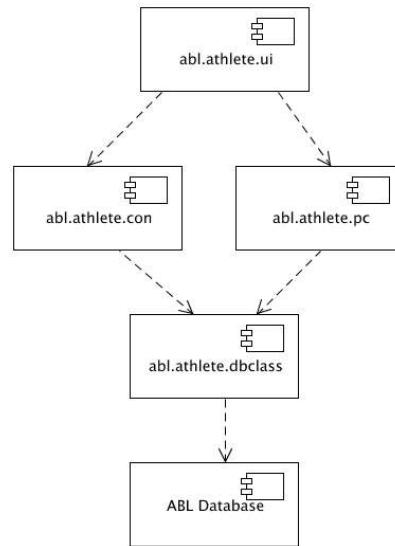


Figure 4.58 Club Membership Maintenance Component Diagram

The naming of the packages follows the convention used by the Java Programming Language. The package `abl.athlete` handles the **Club Membership Maintenance System**. The list below contains the packages and files

- User-interface Package (`abl.athlete.ui`)
 - `AthleteRecordUI.java`
 - `AthleteListUI.java`
 - `FindAthleteRecordUI.java`
- Controllers (`abl.athlete.con`)
 - `MaintainAthleteRecord.java`
 - `AddAthlete.java`
 - `EditAthlete.java`
 - `DeleteAthlete.java`
 - `FindAthleteRecord.java`

- DBClasses (**abl.athlete.dbclass**)
 - **DBAthlete.java**
- Persistent Classes (**abl.athlete.pc**)
 - **Athlete.java**
 - **Guardian.java**

The **ABL Database** is the database server that contains the data that is needed by the applications.

4.7 Deployment-level Design

The **Deployment-level Design** creates a model that shows the physical architecture of the hardware and software of the system. It highlights the physical relationship between software and hardware. The components identified in the Component-level design are distributed to hardware.

4.7.1 Deployment Diagram Notation

The Deployment Diagram is made up of nodes and communication associations. **Nodes** would represent the computers. They are represented as a three-dimensional box. The **communication associations** show network connectivity. Figure 4.59 illustrates a deployment diagram.

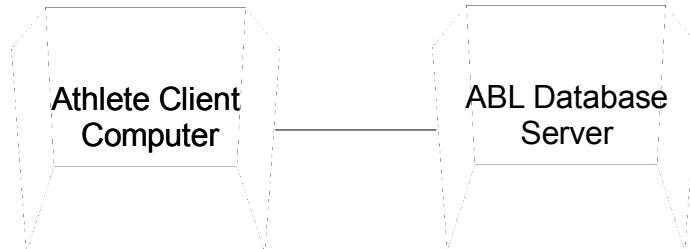


Figure 4.59 Deployment Diagram Notation

4.7.2 Developing the Deployment Model

Developing the deployment diagram is very simple. Just distribute the software components identified in the component-level design to the computer node where it will reside. Figure 4.59 illustrates the deployment diagram for the **Club Membership Maintenance System**.

4.8 Design Model Validation Checklist

Validation is needed to see if the design model:

- fulfills the requirements of the system
- is consistent with design guidelines
- serves as a good basis for implementation

Software Component

1. Is the name of the software component clearly reflect its role within the system?
2. Are the dependencies defined?
3. Is there any model element within the software component visible outside?
4. If the software component has an interface, is the operation defined in the interface mapped to a model element within the software component?

Class

1. Is the name of the class clearly reflect its role within the system?
2. Is the class signifies a single well-defined abstraction?
3. Are all operations and attributes within the class functionally coupled?
4. Are there any attributes, operations or associations that needs to be generalized?
5. Are all specific requirements about the class addressed?
6. Does the class exhibit the required behavior that the system needs?

Operation

1. Can we understand the operation?
2. Does the operation provide the behavior that the class needs?
3. Is the operation signature correct?
4. Are the parameters defined correctly?
5. Are the implementation specifications for an operation correct?
6. Does the operation signature comply with the standards of the target programming language?
7. Is the operation needed or used by the class?

Attribute

1. Is the attribute signify a single conceptual thing?
2. Is the name of the attribute descriptive?
3. Is the attribute needed or used by the class?

4.9 Mapping the Design Deliverables to the Requirements Traceability Matrix

Once the software components are finalized, we need to tie them up to the RTM of the project. This ensures that the model is related to a requirement. It also aids the software engineer in tracking the progress of the development. Table 25 shows the recommended RTM elements that should be added to the matrix.

RTM Design Components	Description
Software Component ID	The name of the software component package that can be independently implemented, eg., abl.athlete (software component that handles club membership application)
Classes	The name of the class that is implemented which is part of the software component, eg., abl.athlete.AthleteRecordUI (a JAVA class responsible for actor interface in processing athlete record).
Class Documentation	The Document Name of the class. It is a file that contains the specification of the class.
Status	The status of the class. Users can define a set of status codes such as: <ul style="list-style-type: none">• CLG – class design on going• COG – coding on going• TOG – testing on going• DONE

Table 25: RTM Design Model Elements

4.10 Design Metrics

In object-oriented software engineering, the class is the fundamental unit. Measures and metrics for an individual class, the class hierarchy, and class collaboration is important to the software engineer especially for the assessment on the design quality.

There are many sets of metrics for the object-oriented software but the widely used is the CK Metrics Suite proposed by Chidamber and Kemerer⁶. It consists of six class-based design metrics which are listed below.

1. *Weighted Methods per Class (WMC)*. This is computed as the summation of the complexity of all methods of a class. Assume that there are n methods defined for the class. We compute for the complexity of each method and find the sum. There are many complexity metric that can be used but the common one is the cyclomatic complexity. This is discussed in the chapter for Software Testing. The number of methods and their complexity indicates:
 - the amount of effort required to implement and test a class
 - the larger the number of methods, the more complex is the inheritance tree
 - as the number of methods grows within the class, the likely it will become more complicated.
2. *Depth of the Inheritance Tree (DIT)*. It is defined as the maximum length from the root superclass to the lowest subclass. Consider the class hierarchy in Figure 4.60, the DIT value is 5. As this number grows, it is likely that the lower-level classes will inherit many methods. It can possibly lead to potential problems in predicting the behavior of the lower classes, and to greater design complexity. On the other hand, a large value implies that many methods are reused.
3. *Number of Children (NOC)*. Children of a class are the immediate subordinate of that class. Consider the class hierarchy in Figure 4.60, the NOC of **class4** is 2. As the number of children increases, reuse increases. However, care is given such that the abstraction represented by the parent class is not diluted by its children which are not appropriately members of the parent class. Of course, as the number of children increases, the number of testing the children of the parent class also increases.
4. *Coupling Between Object Classes (CBO)*. It is the number of collaboration that a class does with other object. As this number increases, the reusability factor of the class decreases. It also complicates modifications and testing. It is for this reason that CBO is kept to a minimum.
5. *Response for a class (RFC)*. It is the number of methods that are executed in response to a message given to an object of the class. As this number increases, the effort required to test also increases because it increases the possible test sequence.
6. *Lack of Cohesion in Methods (LCOM)*. It is the number of methods that access an attribute within the class. If this number is high, methods are coupled together through this attribute. This increases the complexity of the class design. It is best to keep LCOM as low as possible.

⁶ Chidamber, S.R., and C.F. Kemerer, *A Metrics Suite for Object-oriented Design*, IEEE Trans. Software Engineering, vol SE-20, no. 6, June 1994, pp. 476-493

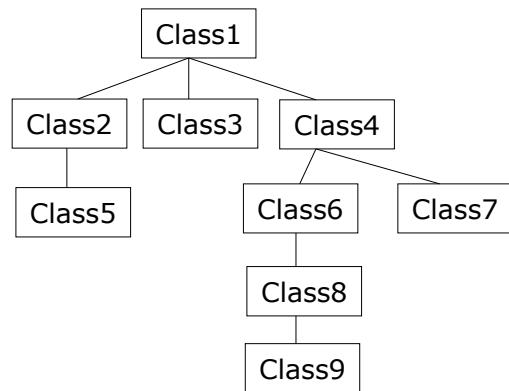


Figure 4.60 Sample Class Hierarchy

4.11 Exercises

4.11.1 Creating the Data Design Model

1. Create the Data Design Model of the Coach Information System.
 - Create the static view of the persistent classes.
 - Model the dynamic behavior of the persistent classes which includes initialization, create, read, update and delete.
2. Create the Data Design Model of the Squad & Team Maintenance System.
 - Create the static view of the persistent classes.
 - Model the dynamic behavior of the persistent classes which includes initialization, create, read, update and delete.

4.11.2 Creating the Interface Design

1. Design the Report Layouts of the following, and use the standards as defined in this chapter.
 - List of Members by Status
 - List of Members who will be moving on or graduating next year
 - List of members who have not yet paid their renewal fee
2. Redesign the Application Form that accommodate the additional athlete information that the club owners want. Use the standards as defined in this chapter.
3. Refine the screen design of the Athlete Screen as shown in [to include additional information needed by the club owners](#).
 - Create a new screen prototype.
 - Define the screen as a class.
 - Model the behavior of the screen.
4. Create the screen design and dialog design of the Coach Information System.
 - Create the screen prototype.
 - Define the screen as a class.
 - Model the behavior of the screen.
5. Create the screen design and dialog design of the Squad and Team Maintenance System.
 - Create a the screen prototypes.
 - Define the screens as classes.
 - Model the behavior of the screen.

4.11.3 Creating the Control Design

1. Refine the control classes of the Coach Information System.
 - Refine and redefine the control classes.

- Model the behavior of the control classes with the screen and persistent classes.
 - Model the component using the component diagram.
2. Refine the control classes of the Squad & Team Maintenance.
- Refine and redefine the control classes.
 - Model the behavior of the control classes with the screen and persistent classes.
 - Model the component using the component diagram.
3. Define the component diagram of the software
- Refine the software architecture by repackaging the classes.
 - Define the component diagram of each package in the software architecture.

4.12 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in this chapter. Particularly, they are:

1. Translating the Analysis Model into the Design Model
2. Developing the Data Design Model
3. Developing the Interface Design Model
4. Developing the Software Component Design
5. Developing the Software Architecture

MAJOR WORK PRODUCTS:

1. Software Architecture
2. Data Design Model
3. Interface Design Model
4. Control Class Model
5. Software Component Design Model (IMPORTANT!)
6. Requirements Traceability Matrix
7. Action List

5 Implementation

Programs are written in accordance to what is specified in the design. Translating design can be a daunting task. First, the designer of the system might not able to address the specifics of the platform and programming environment. Second, codes should be written in a way that is understandable not only to the programmer who did it but by others (other programmers and testers). Third, the programmer must take advantage of the characteristics of the design's organization, the data's structure and programming language constructs while still creating code that is easily maintainable and reusable. This chapter does not intend to teach programming as this is reserved for a programming course. This chapter gives some programming guidelines that may be adopted when creating codes. It also gives some software engineering practices that every programmer must keep in mind.

5.1 Programming Standards and Procedures

Teams are involved in developing a software. A variety of tasks performed by different people is required to generate a quality product. Sometimes when writing a code, several people are involved. Thus, a great deal of cooperation and coordination is required. It is important that others be able to understand the code that was written, why it was written, and how it fits in the software to be developed. It is for this reason that programming standards and procedures be defined and consistently used.

Programming standards and procedures are important for several reasons. First, it can help programmers organize their thoughts and avoid mistakes. How to document the source codes to make them clear and easy to follow makes it easier for the programmer to write and maintain the codes. It also helps in locating faults and aids in making changes because it gives the section where the changes are applied. It also helps in the translation from the design code to the source code through maintaining the correspondence between the design components with the implementation components. Second, it can assist other team members such as software testers, software integrators and software maintainers. It allows communication and coordination to be smooth among the software development teams.

The people involved in the development software must decide on implementation specific standards. These may be:

1. Platform where the software will be developed and used. This includes the minimum and recommended requirements of the hardware and software. For the software, they include versions. Upgrades are not really recommended. However, if upgrades are necessary, the impact should be analyzed and addressed.
2. Standards on the source code documentation. This will be discussed in the programming documentation guidelines.
3. Standards for correspondence between design and source codes. The design model has no value if the design's modularity is not carried forward to the implementation. Design characteristics such as low coupling, high-cohesion and well-defined interfaces should also be the program characteristics. The software's general purpose might remain the same throughout its life cycle but its characteristics and nature may change over time as customer requirements are modified and enhancements are identified. Changes are first reflected on the design. However, it will be traced down

to lower level components. Design correspondence with source code allows us to locate the necessary source code to be modified.

5.2 Programming Guidelines

Programming is a creative skill. The programmer has the flexibility to implement the code. The design component is used as a guide to the function and purpose of the component. Language-specific guidelines are not really discussed here but for the Java Programming Standard Guidelines see their website at <http://java.sun.com/docs/codeconv/index.html>. This section discusses several guidelines that apply to programming in general.

5.2.1 Using Pseudocodes

The design usually provides a framework for each component. It is an outline of what is to be done in a component. The programmer adds his creativity and expertise to build the lines of code that implement the design. He has the flexibility to choose the particular programming language constructs to use, how to use them, how the data will be represented, and so on.

Pseudocodes can be used to adapt the design to the chosen programming language. They are structured English that describes the flow of a program code. By adopting the constructs and data representations without becoming involved immediately in the specifics of a command or statement, the programmer can experiment and decide which implementation is best. Codes can be rearranged or reconstructed with a minimum of rewriting.

5.2.2 Control Structure Guidelines

The control structure is defined by the software architecture. In object-oriented software, it is based on messages being sent among objects of classes, system states and changes in the variables. It is important for the program structure to reflect the design's control structure. Modularity is a good design characteristic that must be translated to program characteristic. By building the program in modular blocks, a programmer can hide implementation details at different levels, making the entire system easier to understand, test and maintain. Coupling and cohesion are another design characteristics that must be translated into program characteristics.

5.2.3 Documentation Guidelines

Program Documentation is a set of written description that explain to the reader what the programs do and how they do it. Two program documentation are created: ***internal documentation*** and ***external documentation***.

Internal Documentation

It is a descriptive document directly written within the source code. It is directed at some who will be reading the source code. A summary information is provided to describe its data structures, algorithms, and control flow. Normally, this information is placed at the beginning of the code. This section of the source code is known as the ***header comment block***. It acts as an introduction to the source code. It identifies

the following elements:

- Component Name
- Author of the Component
- Date the Component was last created or modified
- Place where the component fits in the general system
- Details of the component's data structure, algorithm and control flow

Optionally, one can add history of revisions that was done to the component. The history element consists of:

- Who modified the component?
- When the component was modified?
- What was the modification?

Internal documents are created for people who will be reading the code.

Tips in writing codes

1. Use meaningful variable names and method names.
2. Use formatting to enhance readability of the codes such as indentation and proper blocking.
3. Place additional comments to enlighten readers on some program statements.
4. Have separate methods for input and output.
5. Avoid using GOTO's. Avoid writing codes that jump wildly from one place to another.
6. Writing code is also iterative, i.e., one starts with a draft. If the control flow is complex and difficult to understand, one may need to restructure the program.

External Documentation

All other documents that are not part of the source code but is related to the source code are known as **external documents**. These type of documents are indented for people who may not necessarily read the codes. They describe how components interact with one another which include object classes and their inheritance hierarchy. For object-oriented system, it identifies the pre-conditions and post-conditions of the source code.

5.3 Implementing Packages

Packages provide a mechanism for software reuse. One of the goals of programmers is to create reusable software components so that codes are not repeatedly written. The Java Programming Language provides a mechanism for defining packages. They are actually directories used to organize classes and interfaces. Java provides a convention for unique package and class names. With hundreds of thousands Java programmers around the world, the name one uses for his classes may conflict with classes developed by other programmers. Package names should be in all-lowercase ASCII letters. It should follow the Internet Domain Name Convention as specified in X.500 format for distinguished names.

The following are the steps in defining a package in Java.

1. Define a public class. If the class is not public, it can be used only by other classes in the same package. Consider the code of the **Athlete** persistent class shown in Text 9. The **Athlete** class is made public.

```
package abl.athlete.pc;

public class Athlete{
    private int athleteID;
    private String lastName;
    private String firstName;
    ... // the rest of the attributes

    //set methods
    public void setAthleteID( int id ){
        athleteID = id;
    }
    ... // the other set methods

    //get methods
    public int getAthleteID() {
        return athleteID
    }
    ... //the other get methods
}
```

Text 9: Sample Package Definition

2. Choose a package name. Add the **package** statement to the source code file for a reusable class definition. In this example, the package name is **abl.athlete.pc**

which is the name of the package for persistent classes and class lists. In the example, it is:

```
package abl.athlete.pc;
```

Placing a package statement at the beginning of the source file indicates that the class defined in the file is part of the specified package.

3. Compile the class so that it is placed in the appropriate package directory structure. The compiled class is made available to the compiler and interpreter. When a Java file containing a package statement is compiled, the result *.class file is placed in the directory specified by the package statement. In the example, the **athlete.class** file is placed under the **pc** directory under the **athlete** which is under the **abl** directory. If these directories do not exist, the compiler creates them.
4. To reuse the class, just import the package. The statement shown in Text 10 is an example of importing the **Athlete** class which is used by the **DBAthlete** class.

```
import abl.athlete.pc.*; // All public class of this package is
                         //available for DBAthlete use.

public class DBAthlete{
    private Athlete ath;// defines a reference to an
                         // object Athlete
    ... // the rest of the code

}
```

Text 10: Sample Package Importation

5.4 Implementing Controllers

Implementing controllers is similar to writing programs in your previous courses. A good programming practice is using abstract classes and interfaces. This section serves as a review of abstract classes and interfaces. The use of abstract classes and interfaces greatly increase the ability of the software to be reusable and manageable. It also allows software to have a 'plug-and-play' capabilities. This section serves as a review of abstract classes and interfaces in Java⁷.

5.4.1 Abstract Classes

Now suppose we want to create a superclass wherein it has certain methods in it that contains some implementation, and some methods wherein we just want to be overridden by its subclasses.

For example, we want to create a superclass named `LivingThing`. This class has certain methods like breath, eat, sleep and walk. However, there are some methods in this superclass wherein we cannot generalize the behavior. Take for example, the walk method. Not all living things walk the same way. Take the humans for instance, we humans walk on two legs, while other living things like dogs walk on four legs. However, there are many characteristics that living things have in common, that is why we want to create a general superclass for this.

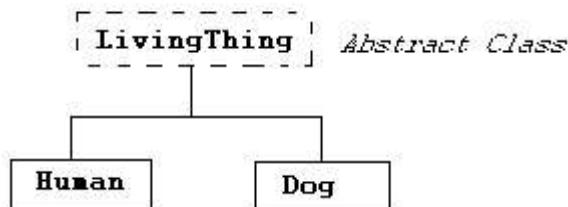


Figure 5.1: Abstract class

In order to do this, we can create a superclass that has some methods with implementations and others which do not. This kind of class is called an abstract class.

An **abstract class** is a class that cannot be instantiated. It often appears at the top of an object-oriented programming class hierarchy, defining the broad types of actions possible with objects of all subclasses of the class.

Those methods in the abstract classes that do not have implementation are called **abstract methods**. To create an abstract method, just write the method declaration without the body and use the `abstract` keyword. For example,

```
public abstract void someMethod();
```

⁷ The use of the text verbatim from the JEDI Introduction to Programming Course has prior approval from their authors.

Now, let's create an example abstract class.

```
public abstract class LivingThing
{
    public void breath(){
        System.out.println("Living Thing breathing...");

    }

    public void eat(){
        System.out.println("Living Thing eating...");

    }

    /**
     * abstract method walk
     * We want this method to be overridden by subclasses of
     * LivingThing
     */
    public abstract void walk();
}
```

When a class extends the LivingThing abstract class, it is required to override the abstract method walk(), or else, that subclass will also become an abstract class, and therefore cannot be instantiated. For example,

```
public class Human extends LivingThing
{
    public void walk(){
        System.out.println("Human walks...");

    }
}
```

If the class Human does not override the walk method, we would encounter the following error message,

```
Human.java:1: Human is not abstract and does not override
abstract method walk() in LivingThing
public class Human extends LivingThing
 ^
1 error
```

Coding Guidelines:

Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

5.4.2 Interfaces

An **interface** is a special kind of block containing method signatures (and possibly constants) only. Interfaces define the signatures of a set of methods without the body.

Interfaces define a standard and public way of specifying the behavior of classes. They allow classes, regardless of their location in the class hierarchy, to implement common behaviors. Note that interfaces exhibit polymorphism as well, since program may call an interface method and the proper version of that method will be executed depending on the type of object passed to the interface method call.

5.4.3 Why do we use Interfaces?

We need to use interfaces if we want unrelated classes to implement similar methods. Through interfaces, we can actually capture similarities among unrelated classes without artificially forcing a class relationship.

Let's take as an example a class **Line** which contains methods that computes the length of the line and compares a **Line** object to objects of the same class. Now, suppose we have another class **MyInteger** which contains methods that compares a **MyInteger** object to objects of the same class. As we can see here, both of the classes have some similar methods which compares them from other objects of the same type, but they are not related whatsoever. In order to enforce a way to make sure that these two classes implement some methods with similar signatures, we can use an interface for this. We can create an interface class, let's say interface **Relation** which has some comparison method declarations. Our interface Relation can be declared as,

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Another reason for using an object's programming interface is to *reveal an object's programming interface without revealing its class*. As we can see later on the section *Interface vs. Classes*, we can actually use an interface as data type.

Finally, we need to use interfaces to *model multiple inheritance* which allows a class to have more than one superclass. Multiple inheritance is not present in Java, but present in other object-oriented languages like C++.

5.4.4 Interface vs. Abstract Class

The following are the main differences between an interface and an abstract class: interface methods have no body, an interface can only define constants and an interface have no direct inherited relationship with any particular class, they are defined independently.

5.4.5 Interface vs. Class

One common characteristic of an interface and class is that they are both types. This means that an interface can be used in places where a class can be used. For example, given a class Person and an interface PersonInterface, the following declarations are valid:

```
PersonInterface pi = new Person();
Person          pc = new Person();
```

However, you cannot create an instance from an interface. An example of this is:

```
PersonInterface pi = new PersonInterface(); //COMPILE
                                                //ERROR!!!
```

Another common characteristic is that both interface and class can define methods. However, an interface does not have an implementation code while the class have one.

5.4.6 Creating Interfaces

To create an interface, we write,

```
public interface [InterfaceName]
{
    //some methods without the body
}
```

As an example, let's create an interface that defines relationships between two objects according to the "natural order" of the objects.

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Now, to use the interface, we use the **implements** keyword. For example,

```
/**
 * This class defines a line segment
 */
public class Line implements Relation
{
    private double x1;
    private double x2;
    private double y1;
    private double y2;

    public Line(double x1, double x2, double y1, double y2){
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }
}
```

```
}

public double getLength(){
    double length = Math.sqrt((x2-x1)*(x2-x1) +
                               (y2-y1)*(y2-y1));
    return length;
}

public boolean isGreater( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen > bLen);
}

public boolean isLess( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen < bLen);
}

public boolean isEqual( Object a, Object b){
    double aLen = ((Line)a).getLength();
    double bLen = ((Line)b).getLength();
    return (aLen == bLen);
}
}
```

When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override
abstract method isGreater(java.lang.Object,java.lang.Object) in
Relation
public class Line implements Relation
^
1 error
```

Coding Guidelines:

Use interfaces to create the same standard method definitions in many different classes. Once a set of standard method definition is created, you can write a single method to manipulate all of the classes that implement the interface.

5.4.7 Relationship of an Interface to a Class

As we have seen in the previous section, a class can implement an interface as long as it provides the implementation code for all the methods defined in the interface.

Another thing to note about the relationship of interfaces to classes is that, a class can only EXTEND ONE super class, but it can IMPLEMENT MANY interfaces. An example of a class that implements many interfaces is,

```
public class Person implements PersonInterface,  
    LivingThing,  
    WhateverInterface {  
  
    //some code here  
}
```

Another example of a class that extends one super class and implements an interface is,

```
public class ComputerScienceStudent extends Student  
    implements PersonInterface,  
    LivingThing {  
  
    //some code here  
}
```

Take note that an interface is not part of the class inheritance hierarchy. Unrelated classes can implement the same interface.

5.4.8 Inheritance among Interfaces

Interfaces are not part of the class hierarchy. However, interfaces can have inheritance relationship *among themselves*. For example, suppose we have two interfaces **StudentInterface** and **PersonInterface**. If StudentInterface extends PersonInterface, it will inherit all of the method declarations in PersonInterface.

```
public interface PersonInterface {  
    . . .  
}  
  
public interface StudentInterface extends PersonInterface {  
    . . .  
}
```

An **interface** formalizes polymorphism. It defines polymorphism in a declarative way, unrelated to implementation. This is the key to the "plug-and-play" ability of an architecture. It is a special kind of block containing method signatures (and possibly constants) only. Interfaces define the signatures of a set of methods without the body.

Interfaces define a standard and public way of specifying the behavior of classes. They allow classes, regardless of their location in the class hierarchy, to implement common behaviors. Note that interfaces exhibit polymorphism as well, since program may call an interface method and the proper version of that method will be executed depending on the type of object passed to the interface method call.

Why do we use Interfaces? We need to use interfaces if we want unrelated classes to implement similar methods. Thru interfaces, we can actually capture similarities among unrelated classes without artificially forcing a class relationship.

Let's take as an example a class **Line** which contains methods that computes the length of the line and compares a **Line** object to objects of the same class. Now, suppose we have another class **MyInteger** which contains methods that compares a **MyInteger** object to objects of the same class. As we can see here, both of the classes have some similar methods which compares them from other objects of the same type, but they are not related whatsoever. In order to enforce a way to make sure that these two classes implement some methods with similar signatures, we can use an interface for this. We can create an interface class, let's say interface **Relation** which has some comparison method declarations. Our interface Relation can be declared as,

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Another reason for using an object's programming interface is to *reveal an object's programming interface without revealing its class*. As we can see later on the section *Interface vs. Classes*, we can actually use an interface as data type.

Finally, we need to use interfaces to *model multiple inheritance* which allows a class to have more than one superclass. Multiple inheritance is not present in Java, but present in other object-oriented languages like C++.

The interface methods have no body, an interface can only define constants and an interface have no direct inherited relationship with any particular class, they are defined independently.

One common characteristic of an interface and class is that they are both types. This means that an interface can be used in places where a class can be used. For example, given a class Person and an interface PersonInterface, the following declarations are valid:

```
PersonInterface pi = new Person();
Person          pc = new Person();
```

However, you cannot create an instance from an interface. An example of this is:

```
PersonInterface pi = new PersonInterface(); //COMPILE
                                                //ERROR!!!
```

Another common characteristic is that both interface and class can define methods. However, an interface does not have an implementation code while the class have one.

To create an interface, we write,

```
public interface [InterfaceName]
{
    //some methods without the body
```

```
}
```

As an example, let's create an interface that defines relationships between two objects according to the "natural order" of the objects.

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Now, to use the interface, we use the **implements** keyword. For example,

```
/**
 * This class defines a line segment
 */
public class Line implements Relation
{
    private double x1;
    private double x2;
    private double y1;
    private double y2;

    public Line(double x1, double x2,
               double y1, double y2){
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1) +
                                  (y2-y1)*(y2-y1));
        return length;
    }

    public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen > bLen);
    }

    public boolean isLess( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen < bLen);
    }

    public boolean isEqual( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen == bLen);
    }
}
```

```
    }  
}
```

When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override  
abstract method isGreater(java.lang.Object, java.lang.Object) in  
Relation  
public class Line implements Relation  
^  
1 error
```

As we have seen in the previous section, a class can implement an interface as long as it provides the implementation code for all the methods defined in the interface.

Another thing to note about the relationship of interfaces to classes is that, a class can only EXTEND ONE super class, but it can IMPLEMENT MANY interfaces. An example of a class that implements many interfaces is,

```
public class Person implements PersonInterface, LivingThing,  
    WhateverInterface {  
  
    //some code here  
}
```

Another example of a class that extends one super class and implements an interface is,

```
public class ComputerScienceStudent extends Student  
implements PersonInterface, LivingThing {  
  
    //some code here  
}
```

Take note that an interface is not part of the class inheritance hierarchy. Unrelated classes can implement the same interface.

Interfaces are not part of the class hierarchy. However, interfaces can have inheritance relationship *among themselves*. For example, suppose we have two interfaces **StudentInterface** and **PersonInterface**. If StudentInterface extends PersonInterface, it will inherit all of the method declarations in PersonInterface.

```
public interface PersonInterface {  
    . . .  
}  
  
public interface StudentInterface extends PersonInterface {  
    . . .  
}
```

5.5 Implementing Java Database Connectivity (JDBC)

Most applications use relational database system as the repository of data. A language called *Structured Query Language (SQL)* is almost universally used among these systems. They are used to make queries, i.e., request information that satisfy certain criteria. Java enables programmer to write code that uses SQL queries to access the information in a relational database systems. The engineering of the database that implements the data of the application is beyond the scope of this course and is normally discussed in a database course. This section discusses how Java uses JDBC to connect and request services to a database server. In the example, we will show a fraction of the code that retrieves athlete records in the database. The code assumes that **Athlete** and **PCLAthlete** have been implemented already.

The fraction of codes shown in Text 11 and Text 12 is an implementation of the **DBAthlete** Read wherein the **PCLAthlete** is populated with athlete records that satisfy a criteria based on user input. The **import java.sql** statement imports the package that contains classes and interfaces for manipulating relational databases in Java.

The constructor of the class attempts to connect to the database. If successful, queries can be made to the database. The following lines specify the database URL (Uniform Resource Locator) that helps the program locate the database, username and password. The URL specifies the protocol for communication (**jdbc**), the subprotocol (**odbc**) and the name of the database (**ABLDATABASE**). The username and password are also specified for logging into the database; in the example, the username is "postgre" and the password is "postres1".

```
String url = "jdbc:odbc:ABLDATABASE";
String username = "postgre";
String password = "postres1";
```

The class definition for the database driver must be loaded before the program can connect to the database. The following line loads the driver. In this example, the default JDBC-to-ODBC-bridge database driver is used to allow any Java program to access any ODBC data source. For more information on JDBC driver and supported databases visit the following site: <http://java.sun.com/products/jdbc>.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

To establish a connection, a connection object is created. It manages the connection between the database and the Java program. It also provides support for executing SQL statements. In the example, the following statement provides this connection.

```
connection = DriverManager.getConnection(url, username, password);
```

The initial method that is invoke to start the retrieval of athlete records to the database is the **getAthleteRecord()** method. The code of this method is shown below. The string

```
import java.sql.*;
import java.util.*;

public class DBAthlete()
{
    private Connection connection;
    private PCLAthlete athleteList;

    public DBTeam() {

        // To connect to the database...
        String url = "jdbc:odbc:ABLDatabase";
        String username = "postgre";
        String password = "postgres1";

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            connection = DriverManager.getConnection
                (url, username, password);
        }
        catch ( ClassNotFoundException) {
            . . .
        }
    }

    . . .

    // to execute SELECT statement
    public void getAthleteRecord(String searchCriteria) {

        Statement statement;
        ResultSet rs;
        String selectStmt;

        selectStmt = prepareSelectStmt(searchCriteria);
        statement = connection.createStatement();
        rs = statement.executeQuery(selectStmt);
        populatePCLAthlete( rs );
        statement.close();

    }

    // to generate the SELECT-statement with the search criteria
    private String prepareSelectStmt(String searchCriteria) {
        String query = "SELECT * FROM athlete, guardian "
        query = query +
            "WHERE athlete.guardianID = guardian.guardianID";
        query = query + " AND " + searchCriteria + ";" ;

        return query;
    }
}
```

Text 11 DBAthlete Implementation

selectStmt is used to define the select-statement that specifies what records are retrieved from the database. The private method **prepareSelectStmt()** generates the select-statement with the appropriate criteria. The **statement** object is used to query the database and is instantiated using the **createStatement()** method of the **connection** object. The **executeQuery()** method of the **statement** object is used to execute the Select-statement as specified in **selectStmt** string. The result of the query is placed and referenced in **rs** which is a **ResultSet** object.

The **PCLAthlete** object is populated by executing the **populatePCLAthlete()** method which takes as input the **ResultSet**. For each record in the result set, an **athlete** object is instantiated and the data of the athlete are assigned to the attributes of this object. The **getPCLAthlete()** method returns a reference to the generated athlete list.

```
//to populate the persistent class list.
private void populatePCLAthlete(ResultSet rs){
    ...
    athleteList = new PCLAthlete();

    ...
    //inside a try and catch block
    ResultSetMetaData rsmd = rs.getMetaData();
    do {
        athleteList.add( getNextMember( rs, rsmd ) );
    } while ( rs.next() )
}

//to generate an athlete object to be added to the PCLAthlete

private Athlete getNextMember (ResultSet rs,
    ResultSetMetaData rsmd){

    Athlete a = new Athlete();

    a.setAthleteID( rs.getString( 1 ) ); //assumes member has a
        // constructor to convert string to integer
    a.setLastName( rs.getString(2) );
    a.setFirstName( rs.getString(3) );
    ... // set attributes to the athlete object a

    return a;
}

// return a reference to the athlete list (PCLAthlete)
public PCLAthlete getPCLAthlete() {
    return athleteList;
}
```

Text 12 DBAthlete Implementation Continuation

5.6 Implementing the Graphical User Interface

Without learning about graphical user interface (GUI) APIs, you would still be able to create quite a descent range of different programs. However, your applications are very likely to be bland and unappealing to the users. Having a good GUI affects the usage of your application. This results in ease of use and enjoyment of use for the users of your program. Java provides tools like Abstract Windowing Toolkit (AWT) and Swing to develop interactive GUI applications. The following section provides a discussion on the Abstract Windowing Toolkit and Swing⁸.

The Java Foundation Classes (JFCs), which is an important part of the Java SDK, refers to a collection of APIs that simplifies the development Java GUI applications. It primarily consists of five APIs including AWT and Swing. The three other APIs are Java2D, Accessibility, and Drag and Drop. All these APIs assist developers in designing and implementing visually-enhanced applications.

Both AWT and Swing provides GUI components that can be used in creating Java applications and applets. You will learn about applets in a latter section. Unlike some AWT components that use native code, Swing is written entirely using the Java programming language. As a result, Swing provides a platform-independent implementation ensuring that applications deployed across different platforms have the same appearance. AWT, however, does ensure that the look and feel of an application run on two different machines be comparable. The Swing API is built around a number of APIs that implement various parts of the AWT. As a result, AWT components can still be used with Swing components.

5.6.1 AWT GUI Components

Fundamental Window Classes

In developing GUI applications, the GUI components such as buttons or text fields are placed in containers. These are the list of important container classes provided in the AWT.

⁸ The use of the text verbatim from the JEDI Introduction to Programming II Course has prior approval from their authors.

AWT Class	Description
Component	An abstract class for objects that can be displayed on the console and interact with the user. The root of all other AWT classes.
Container	An abstract subclass of the <i>Component</i> class. A component that can contain other components.
Panel	Extends the <i>Container</i> class. A frame or window without the titlebar, the menubar nor the border. Superclass of the <i>Applet</i> class.
Window	Also extends <i>Container</i> class. A top-level window, which means that it cannot be contained in any other object. Has no borders and no menubar.
Frame	Extends the <i>Window</i> class. A window with a title, menubar, border, and resizing corners. Has four constructors, two of which have the following signatures: <code>Frame()</code> <code>Frame(String title)</code>

Table 1.2.1: AWT Container classes

To set the size of the window, the overloaded *setSize* method is used.

```
void setSize(int width, int height)
```

Resizes this component to the *width* and *height* provided as parameters.

```
void setSize(Dimension d)
```

Resizes this component to *d.width* and *d.height* based on the *Dimension d* specified.

A window by default is not visible unless you set its visibility to *true*. Here is the syntax for the *setVisible* method.

```
void setVisible(boolean b)
```

In designing GUI applications, *Frame* objects are usually used. Here's an example of how to create such an application.

```
import java.awt.*;

public class SampleFrame extends Frame {
    public static void main(String args[]) {
        SampleFrame sf = new SampleFrame();
        sf.setSize(100, 100); //Try removing this line
        sf.setVisible(true); //Try removing this line
    }
}
```

Note that the close button of the frame doesn't work yet because no event handling mechanism has been added to the program yet. You'll learn about event handling in the next module.

Graphics

Several graphics method are found in the *Graphics* class. Here is the list of some of these methods.

drawLine()	drawPolyline()	setColor()
fillRect()	drawPolygon()	getFont()
drawRect()	fillPolygon()	setFont()
clearRect()	getColor()	drawString()

Table 1.2.2a: Some methods of class Graphics

Related to this class is the *Color* class, which has three constructors.

Constructor Format	Description
Color(int r, int g, int b)	Integer value is from 0 to 255.
Color(float r, float g, float b)	Float value is from 0.0 to 1.0.
Color(int rgbValue)	Value range from 0 to $2^{24}-1$ (black to white). Red: bits 16-23 Green: bits 8-15 Blue: bits 0-7

Table 1.2.2b: Color constructors

Here is a sample program that uses some of the methods in the *Graphics* class.

```
import java.awt.*;

public class GraphicPanel extends Panel {
    GraphicPanel() {
        setBackground(Color.black); //constant in Color class
    }
    public void paint(Graphics g) {
        g.setColor(new Color(0,255,0)); //green
        g.setFont(new Font("Helvetica",Font.PLAIN,16));
        g.drawString("Hello GUI World!", 30, 100);
        g.setColor(new Color(1.0f,0,0)); //red
        g.fillRect(30, 100, 150, 10);
    }
    public static void main(String args[]) {
        Frame f = new Frame("Testing Graphics Panel");
        GraphicPanel gp = new GraphicPanel();
        f.add(gp);
        f.setSize(600, 300);
        f.setVisible(true);
    }
}
```

For a panel to become visible, it should be placed inside a visible window like a frame.

More AWT Components

Here is a list of AWT controls. Controls are components such as buttons or text fields that allow the user to interact with a GUI application. These are all subclasses of the *Component* class.

Label	Button	Choice
TextField	Checkbox	List
TextArea	CheckboxGroup	Scrollbar

Table 1.2.3: AWT Components

The following program creates a frame with controls contained in it.

```
import java.awt.*;  
  
class FrameWControls extends Frame {  
    public static void main(String args[]) {  
        FrameWControls fwc = new FrameWControls();  
        fwc.setLayout(new FlowLayout()); //more on this later  
        fwc.setSize(600, 600);  
        fwc.add(new Button("Test Me!"));  
        fwc.add(new Label("Label"));  
        fwc.add(new TextField());  
        CheckboxGroup cbg = new CheckboxGroup();  
        fwc.add(new Checkbox("chk1", cbg, true));  
        fwc.add(new Checkbox("chk2", cbg, false));  
        fwc.add(new Checkbox("chk3", cbg, false));  
        List list = new List(3, false);  
        list.add("MTV");  
        list.add("V");  
        fwc.add(list);  
        Choice chooser = new Choice();  
        chooser.add("Avril");  
        chooser.add("Monica");  
        chooser.add("Britney");  
        fwc.add(chooser);  
        fwc.add(new Scrollbar());  
        fwc.setVisible(true);  
    }  
}
```

5.6.2 Layout Managers

The position and size of the components within each container is determined by the layout manager. The layout managers governs the layout of the components in the container. These are some of the layout managers included in Java.

1. FlowLayout
2. BorderLayout
3. GridLayout
4. GridBagConstraints
5. CardLayout

The layout manager can be set using the `setLayout` method of the `Container` class. The method has the following signature.

```
void setLayout(LayoutManager mgr)
```

If you prefer not to use any layout manager at all, you pass null as an argument to this method. But then, you would have to position the elements manually with the use of the `setBounds` method of the `Component` class.

```
public void setBounds(int x, int y, int width, int height)
```

The method controls the position based on the arguments `x` and `y`, and the size based on the specified `width` and `height`. This would be quite difficult and tedious to program if you have several `Component` objects within the `Container` object. You'll have to call this method for each component.

In this section, you'll learn about the first three layout managers.

The FlowLayout Manager

The `FlowLayout` is the default manager for the `Panel` class and its subclasses, including the `Applet` class. It positions the components in a left to right and top to bottom manner, starting at the upper-lefthand corner. Imagine how you type using a particular word editor. This is how the `FlowLayout` manager works.

It has three constructors which are as listed below.

<i>FlowLayout Constructors</i>
<code>FlowLayout()</code>
Creates a new <code>FlowLayout</code> object with the center alignment and 5-unit horizontal and vertical gap applied to the components by default.
<code>FlowLayout(int align)</code>
Creates a new <code>FlowLayout</code> object with the specified alignment and the default 5-unit horizontal and vertical gap applied to the components.
<code>FlowLayout(int align, int hgap, int vgap)</code>
Creates a new <code>FlowLayout</code> object with the first argument as the alignment applied and the <code>hgap</code> -unit horizontal and <code>vgap</code> -unit vertical gap applied to the components.

Table 1.3.1: FlowLayout constructors

The gap refers to the spacing between the components and is measured in pixels. The alignment argument should be one of the following:

1. FlowLayout.LEFT
2. FlowLayout.CENTER
3. FlowLayout.RIGHT

What is the expected output of the following program?

```
import java.awt.*;  
  
class FlowLayoutDemo extends Frame {  
    public static void main(String args[]) {  
        FlowLayoutDemo fld = new FlowLayoutDemo();  
        fld.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 10));  
        fld.add(new Button("ONE"));  
        fld.add(new Button("TWO"));  
        fld.add(new Button("THREE"));  
        fld.setSize(100, 100);  
        fld.setVisible(true);  
    }  
}
```

Shown below is a sample output running on Windows platform.



Table 13.1: Output of sample code in Windows

The BorderLayout Manager

The BorderLayout divides the *Container* into five parts- north, south, east, west and the center. Each components is added to a specific region. The north and south regions stretch horizontally whereas the east and west regions adjust vertically. The center region, on the other hand, adjusts in both horizontally and vertically. This layout is the default for *Window* objects, including objects of *Window*'s subclasses *Frame* and *Dialog* type.

<i>BorderLayout Constructors</i>
<code>BorderLayout()</code>
Creates a new BorderLayout object with no spacing applied among the different components.
<code>BorderLayout(int hgap, int vgap)</code>
Creates a new BorderLayout object with <i>hgap</i> -unit horizontal and <i>vgap</i> -unit spacing applied among the different components.

Table 1.3.2: BorderLayout constructors

Like in the *FlowLayout* manager, the parameters *hgap* and *vgap* here also refers to the spacing between the components within the container.

To add a component to a specified region, use the *add* method and pass two arguments: the component to add and the region where the component is to be positioned. Note that only one component can be placed in one region. Adding more than one component to a particular container results in displaying only the last component added. The following list gives the valid regions are predefined fields in the *BorderLayout* class.

1. `BorderLayout.NORTH`
2. `BorderLayout.SOUTH`
3. `BorderLayout.EAST`
4. `BorderLayout.WEST`
5. `BorderLayout.CENTER`

Here is a sample program demonstrating how the *BorderLayout* works.

```
import java.awt.*;

class BorderLayoutDemo extends Frame {
    public static void main(String args[]) {
        BorderLayoutDemo bld = new BorderLayoutDemo();
        bld.setLayout(new BorderLayout(10, 10)); //may remove
        bld.add(new Button("NORTH"), BorderLayout.NORTH);
        bld.add(new Button("SOUTH"), BorderLayout.SOUTH);
        bld.add(new Button("EAST"), BorderLayout.EAST);
        bld.add(new Button("WEST"), BorderLayout.WEST);
        bld.add(new Button("CENTER"), BorderLayout.CENTER);
        bld.setSize(200, 200);
        bld.setVisible(true);
    }
}
```

Here is a sample output of this program. The second figure shows the effect of resizing the frame.

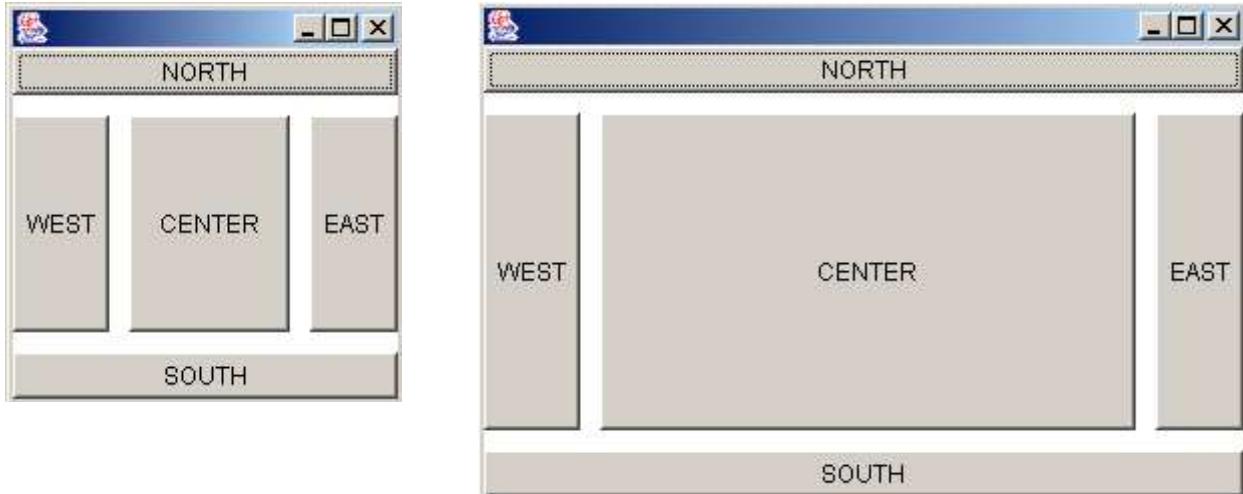


Figure 1.3.2: Output of sample code

The GridLayout Manager

With the *GridLayout* manager, components are also positioned from left to right and top to bottom as in the *FlowLayout* manager. In addition to this, the *GridLayout* manager divides the container into a number of rows and columns. All these regions are equally sized. It always ignores the component's preferred size.

The following is the available constructors for the *GridLayout* class.

GridLayout Constructors
<code>GridLayout()</code>
Creates a new <i>GridLayout</i> object with a single row and a single column by default.
<code>GridLayout(int rows, int cols)</code>
Creates a new <i>GridLayout</i> object with the specified number of rows and columns.
<code>GridLayout(int rows, int cols, int hgap, int vgap)</code>
Creates a new <i>GridLayout</i> object with the specified number of rows and columns. <i>hgap</i> -unit horizontal and <i>vgap</i> -unit vertical spacings are applied to the components.

Table 1.3.3: *GridLayout* constructors

Try out this program.

```
import java.awt.*;

class GridLayoutDemo extends Frame {
    public static void main(String args[]) {
        GridLayoutDemo gld = new GridLayoutDemo();
        gld.setLayout(new GridLayout(2, 3, 4, 4));
```

```
        gld.add(new Button("ONE"));
        gld.add(new Button("TWO"));
        gld.add(new Button("THREE"));
        gld.add(new Button("FOUR"));
        gld.add(new Button("FIVE"));
        gld.setSize(200, 200);
        gld.setVisible(true);
    }
}
```

This is the output of the program. Observe the effect of resizing the frame.

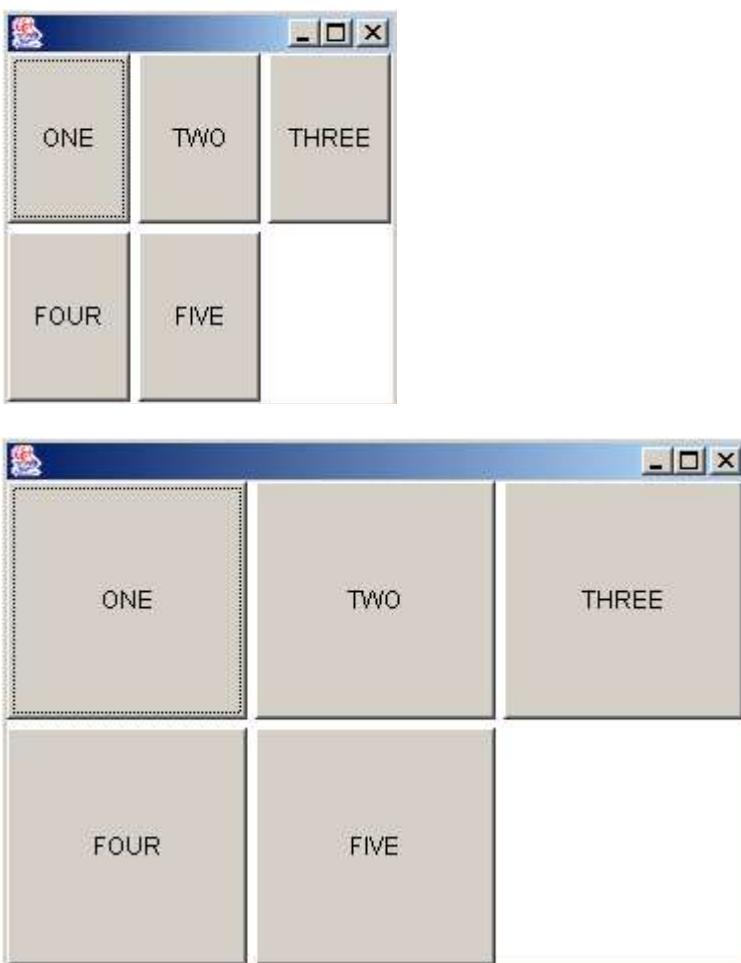


Figure 1.3.3: Output of sample code

Panels and Complex Layouts

To create more complex layouts, you can combine the different layout managers with the use of panels. Remember that a *Panel* is a *Container* and a *Component* at the same time. You can insert *Components* into the *Panel* and then add the *Panel* to a specified region in the *Container*.

Observe the technique used in the following example.

```
import java.awt.*;

class ComplexLayout extends Frame {
    public static void main(String args[]) {
        ComplexLayout cl = new ComplexLayout();
        Panel panelNorth = new Panel();
        Panel panelCenter = new Panel();
        Panel panelSouth = new Panel();
        /* North Panel */
        //Panels use FlowLayout by default
        panelNorth.add(new Button("ONE"));
        panelNorth.add(new Button("TWO"));
        panelNorth.add(new Button("THREE"));
        /* Center Panel */
        panelCenter.setLayout(new GridLayout(4,4));
        panelCenter.add(new TextField("1st"));
        panelCenter.add(new TextField("2nd"));
        panelCenter.add(new TextField("3rd"));
        panelCenter.add(new TextField("4th"));
        /* South Panel */
        panelSouth.setLayout(new BorderLayout());
        panelSouth.add(new Checkbox("Choose me!"),
                      BorderLayout.CENTER);
        panelSouth.add(new Checkbox("I'm here!"),
                      BorderLayout.EAST);
        panelSouth.add(new Checkbox("Pick me!"),
                      BorderLayout.WEST);
        /* Adding the Panels to the Frame container */
        //Frames use BorderLayout by default
        cl.add(panelNorth, BorderLayout.NORTH);
        cl.add(panelCenter, BorderLayout.CENTER);
        cl.add(panelSouth, BorderLayout.SOUTH);
        cl.setSize(300,300);
        cl.setVisible(true);
    }
}
```

Here is the output of the program.

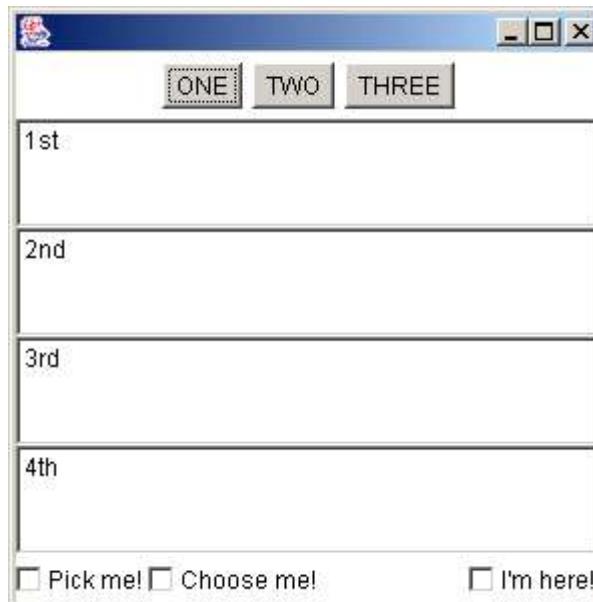


Figure 1.3.4: Output of sample code

5.6.3 Swing GUI Components

Like the AWT package, the Swing package provides classes for creating GUI applications. The package is found in `javax.swing`. The main difference between these two is that Swing components are written entirely using Java whereas the latter is not. As a result, GUI programs written using classes from the Swing package have the same look and feel even when executed on different platforms. Moreover, Swing provides more interesting components such as the color chooser and the option pane.

The names of the Swing GUI components are almost similar to that of the AWT GUI components. An obvious difference is in the naming conventions of the components. Basically, the name of Swing components are just the name of AWT components but with an additional prefix of J. For example, one component in AWT is the *Button* class. The corresponding component for this in the Swing package is the *JButton* class. Provided below is the list of some of the Swing GUI components.

Swing Component	Description
JComponent	The root class for all Swing components, excluding top-level containers.
JButton	A "push" button. Corresponds to the <i>Button</i> class in the AWT package.

Swing Component	Description
JCheckBox	An item that can be selected or deselected by the user. Corresponds to the <i>Checkbox</i> class in the AWT package.
JFileChooser	Allows user to select a file. Corresponds to the <i>FileChooser</i> class in the AWT package.
JTextField	Allows for editing of a single-line text. Corresponds to <i>TextField</i> class in the AWT package.
JFrame	Extends and corresponds to the <i>Frame</i> class in the AWT package but the two are slightly incompatible in terms of adding components to this container. Need to get the current content pane before adding a component.
JPanel	Extends <i>JComponent</i> . A simple container class but not top-level. Corresponds to <i>Panel</i> class in the AWT package.
JApplet	Extends and corresponds to the <i>Applet</i> class in the AWT package. Also slightly incompatible with the <i>Applet</i> class in terms of adding components to this container.
JOptionPane	Extends <i>JComponent</i> . Provides an easy way of displaying pop-up dialog box.
JDialog	Extends and corresponds to the <i>Dialog</i> class in the AWT package. Usually used to inform the user of something or prompt the user for an input.
JColorChooser	Extends <i>JComponent</i> . Allow the user to select a color.

Table 1.4: Some Swing components

For the complete list of Swing components, please refer to the API documentation.

Setting Up Top-Level Containers

As mentioned, the top-level containers like *JFrame* and *JApplet* in Swing are slightly incompatible with those in AWT. This is in terms of adding components to the container. Instead of directly adding a component to the container as in AWT containers, you have to first get the content pane of the container. To do this, you'll have to use the *getContentPane* method of the container.

A JFrame Example

```

import javax.swing.*;
import java.awt.*;

class SwingDemo {
    JFrame frame;
    JPanel panel;
    JTextField textField;
    JButton button;
    Container contentPane;
    void launchFrame() {
        /* initialization */
    }
}

```

```
frame = new JFrame("My First Swing Application");
panel = new JPanel();
textField = new JTextField("Default text");
button = new JButton("Click me!");
contentPane = frame.getContentPane();
/* add components to panel- uses FlowLayout by default */
panel.add(textField);
panel.add(button);
/* add components to contentPane- uses BorderLayout */
contentPane.add(panel, BorderLayout.CENTER);
frame.pack();
//causes size of frame to be based on the components
frame.setVisible(true);
}
public static void main(String args[]) {
    SwingDemo sd = new SwingDemo();
    sd.launchFrame();
}
}
```

Note that the `java.awt` package is still imported because the layout managers in use are defined in this package. Also, giving a title to the frame and packing the components within the frame is applicable for AWT frames too.

Coding Guidelines:

Observe the coding style applied in this example as opposed to the examples for AWT. The components are declared as fields, a launchFrame method is defined, and initialization and addition of components are all done in the launchFrame method. We no longer just extend the Frame class. The advantage of using this style would become apparent when we get to event handling.

Here is a sample output.



Figure 1.4.2: Output of sample code

A JOptionPane Example

```
import javax.swing.*;

class JOptionPaneDemo {
    JOptionPane optionPane;
    void launchFrame() {
        optionPane = new JOptionPane();
        String name = optionPane.showInputDialog("Hi, what's your
                                               name? ");
        optionPane.showMessageDialog(null,
                                   "Nice to meet you, " + name + ".", "Greeting...",
                                   JOptionPane.PLAIN_MESSAGE);
        System.exit(0);
    }
}
```

```
    }
    public static void main(String args[]) {
        new JOptionPaneDemo().launchFrame();
    }
}
```

See how easy it is ask input from the user.
Here is the sample output for the given program.



Figure 1.4.3: Output of sample code

Using AWT and Swing in Sun Java™ Studio Enterprise 8

1. Double-click the Java Studio Enterprise 8 Desktop Icon.

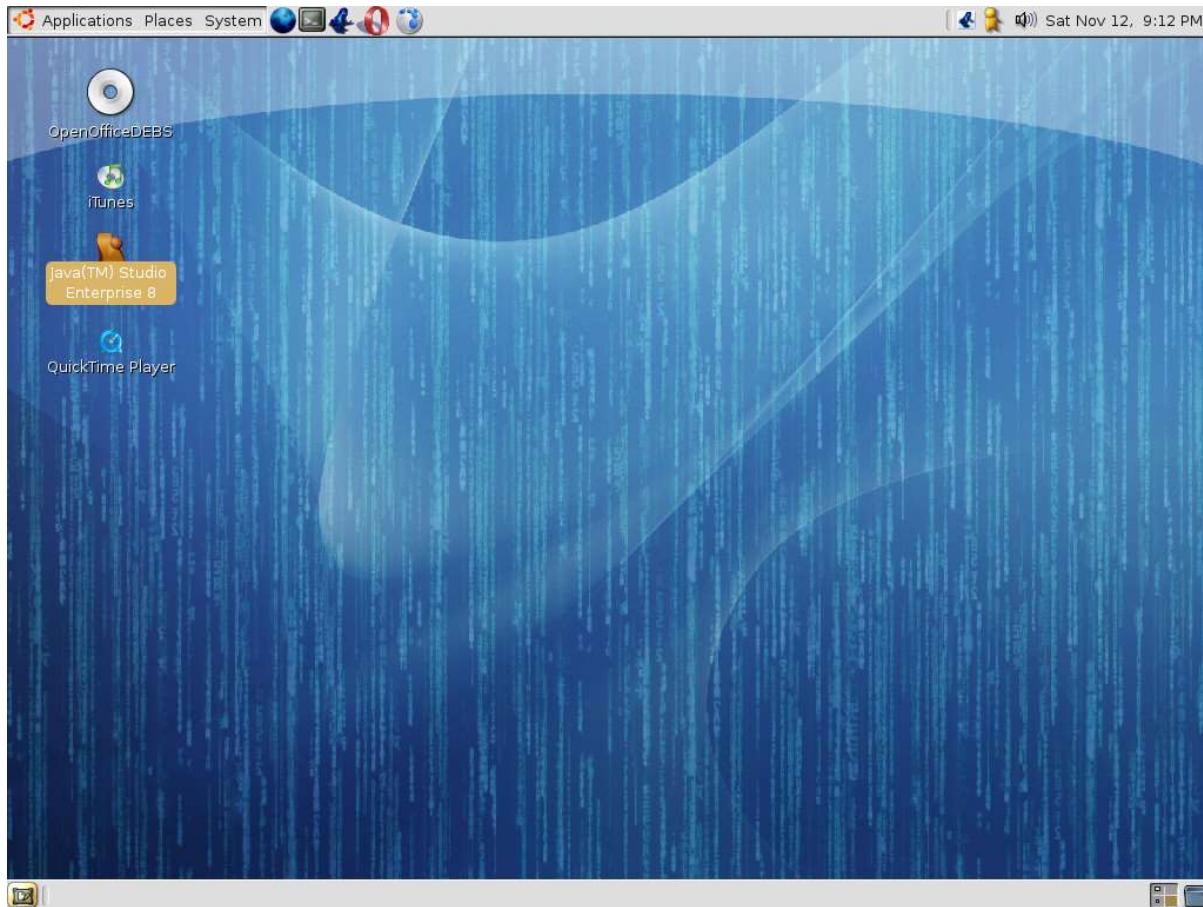


Figure 5.2 Starting Java Studio Enterprise 8

2. You might need to start a new project for the creation of the user interface. To do that, select **File -> New Project**. The New Project dialog box shown on Figure 5.4 will appear.

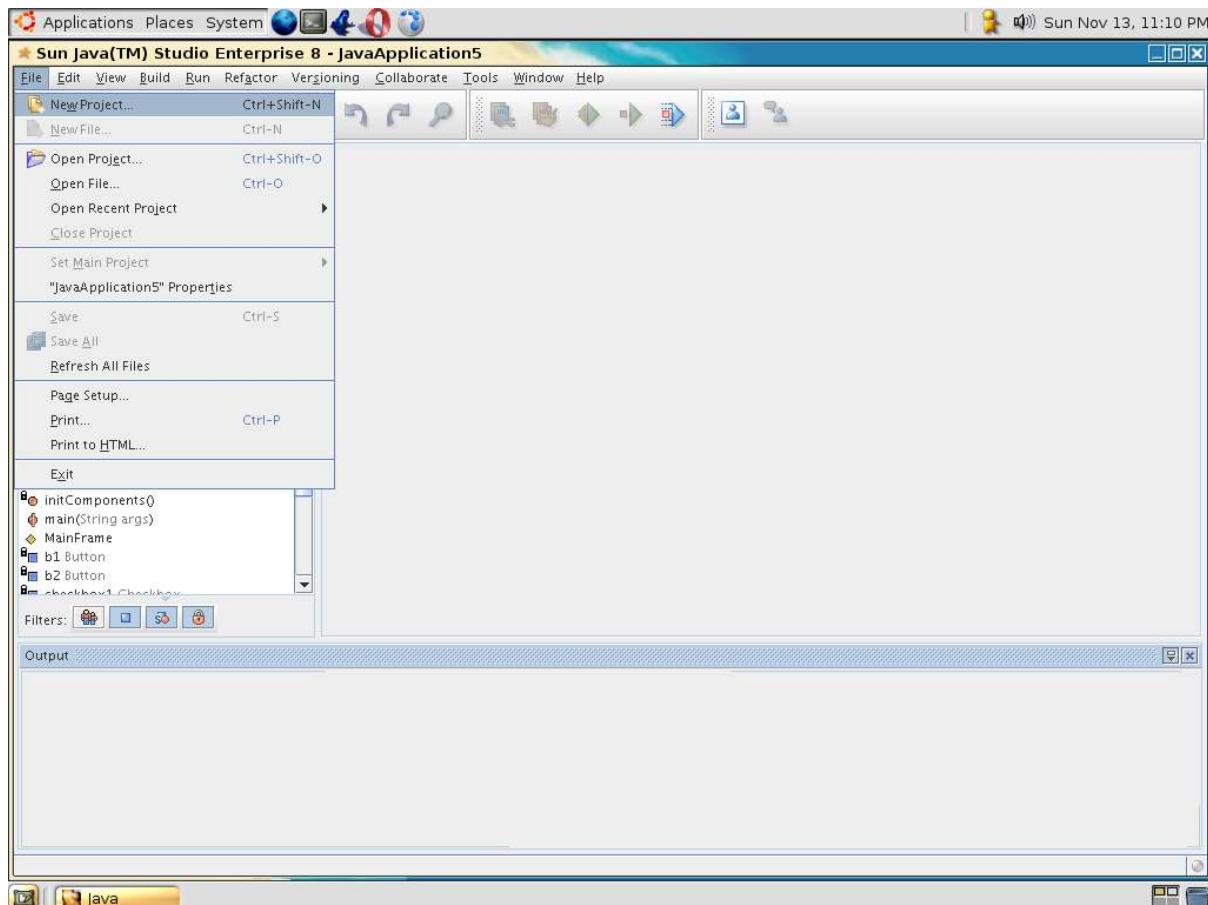


Figure 5.3 Starting a New Project

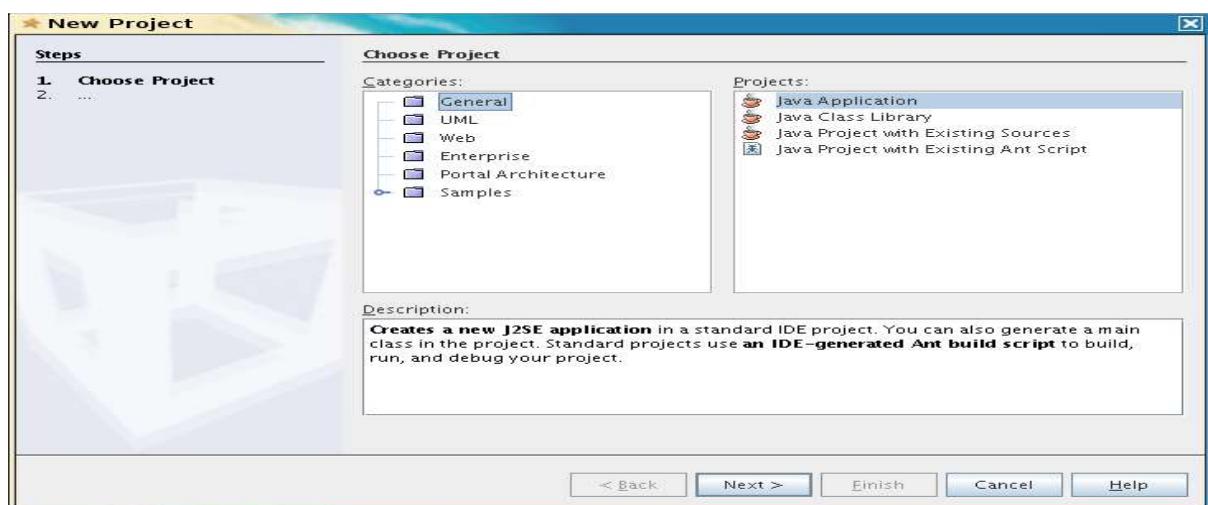


Figure 5.4 Selecting the Type of Project

From this dialog box, select **General -> Java Application**. Click **Next** and the **New Java Application** dialog box will appear.

3. In the **New Java Application** dialog box, enter the project name and the directory that will be used to keep all project-related files. Figure 5.5 shows an example, the project name is **AthleteDB** and the directory is **/home/noel/TestHD/JStudio**.

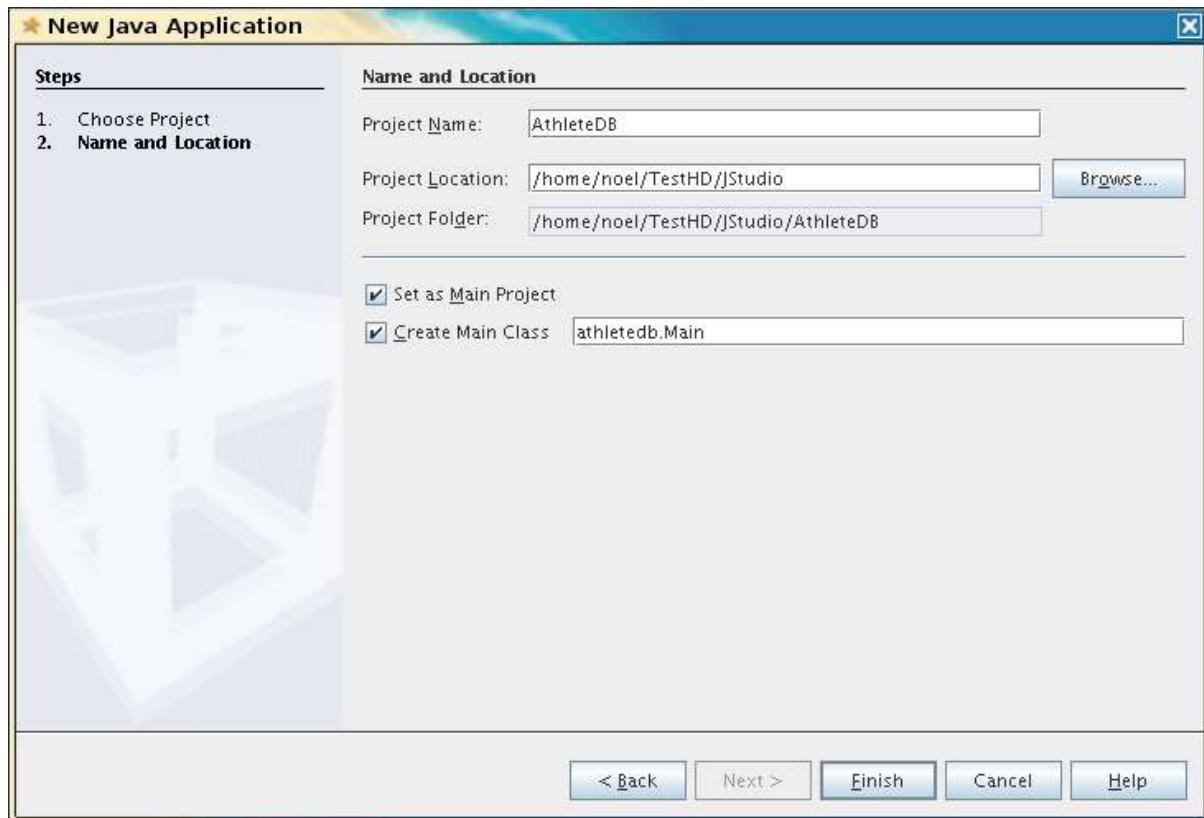


Figure 5.5 New Java Application Dialog Box

4. Java Studio Enterprise will present the default setup as shown in Figure 5.6.

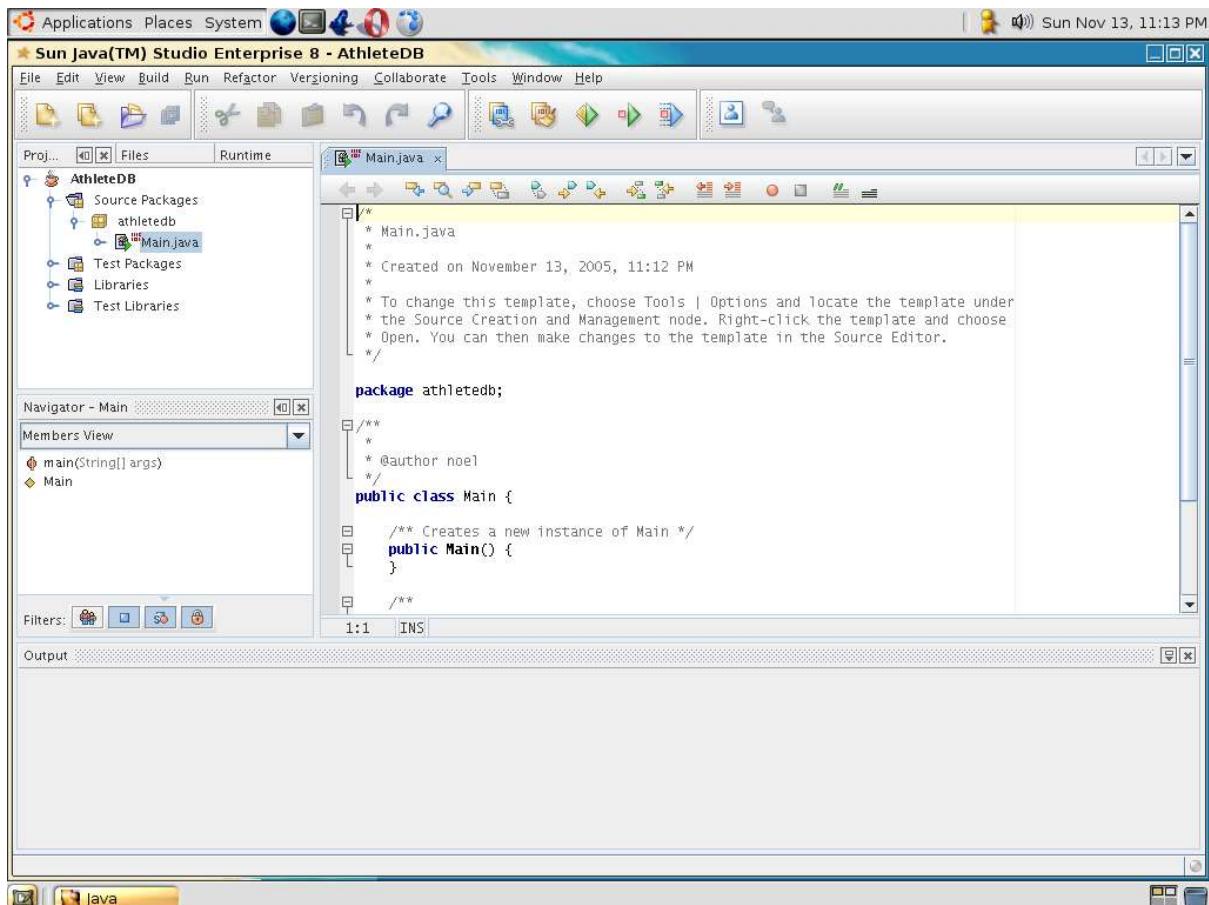


Figure 5.6 Default Set-up Screen

5. To add a frame, select **File -> New File -> Java GUI Forms -> JFrame Form**. Figure 5.7 and Figure 5.8 show how to do this.

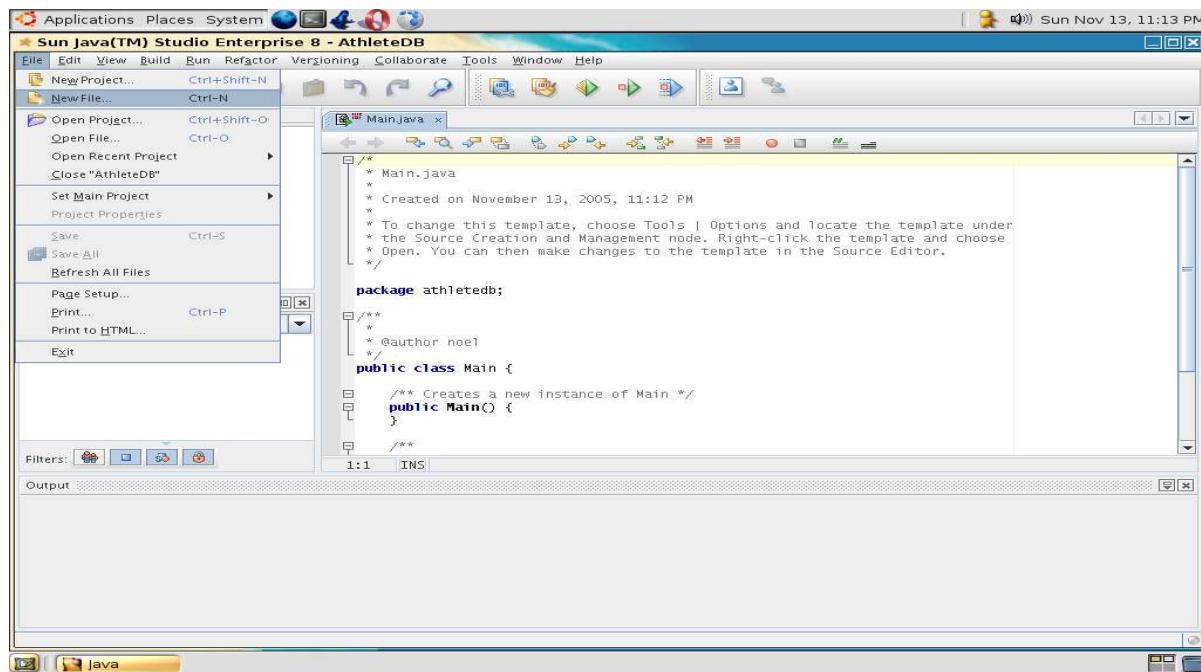


Figure 5.7 Selecting New File option

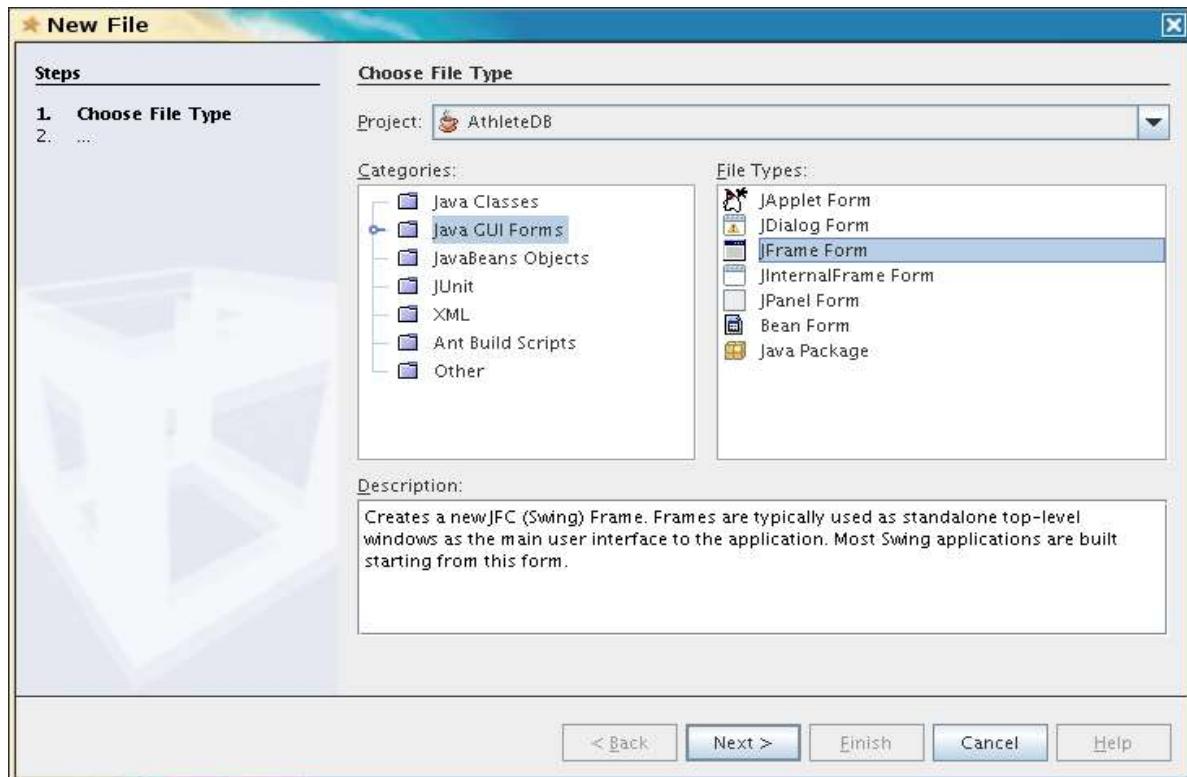


Figure 5.8 Selection Java GUI Forms and JFrame Form

6. Click **Next**. The New JFrame Form dialog box as shown in Figure 5.9 will appear. Enter the class name. In this case, the name of the screen is **AForm**. Then, click **Finish**.

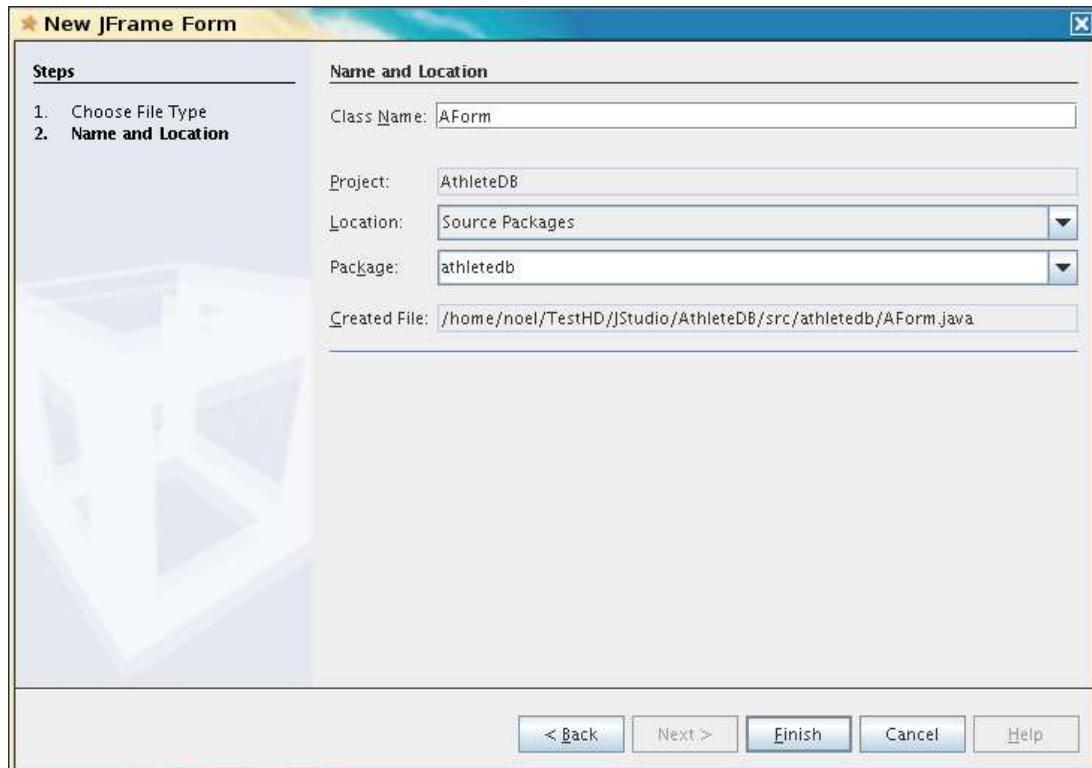


Figure 5.9 New JFrame Form Dialog Box

7. The Design View initialized and is shown in Figure 5.10.

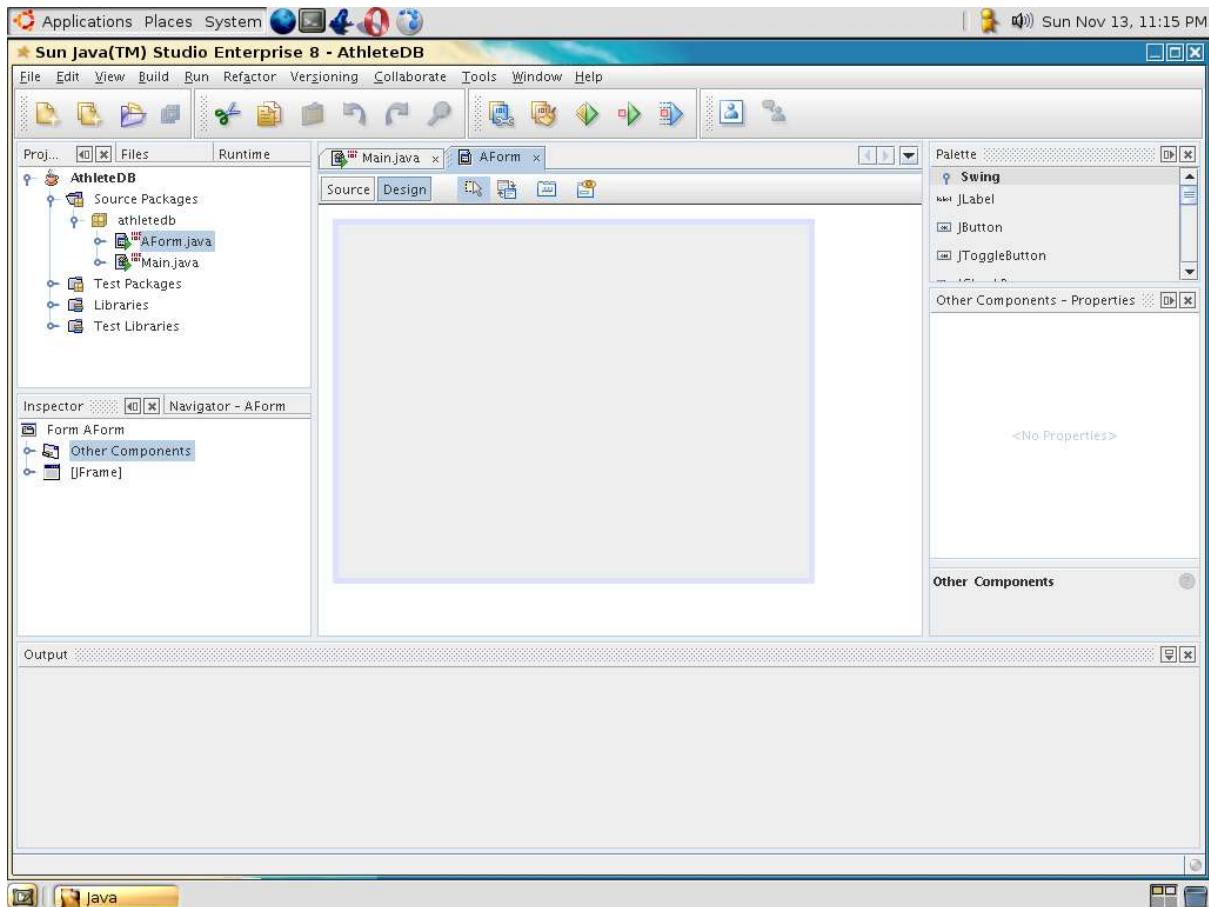


Figure 5.10 Design View Initialization

8. The Palette Manager is located at the top right window and below it is the Component Properties. The Palette Manager contains the Swing, AWT, Layouts, and Beans.

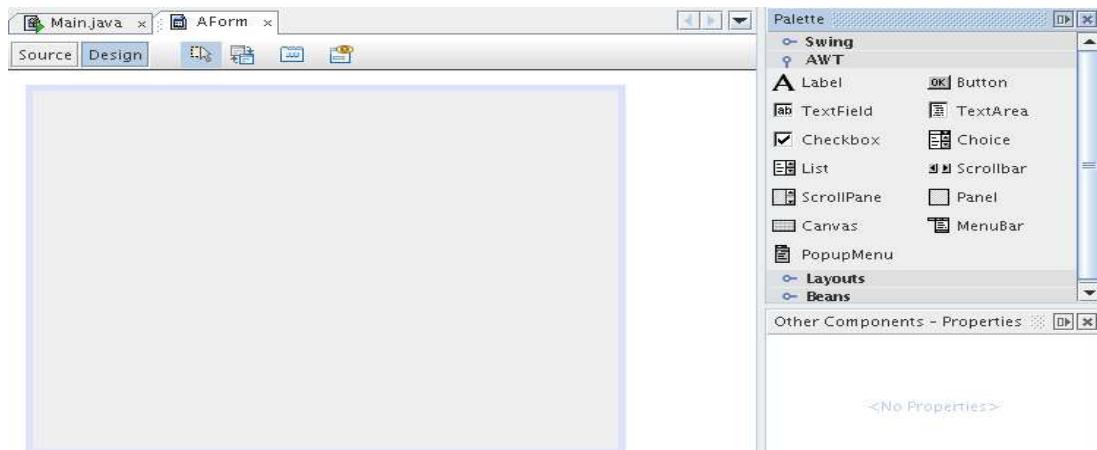


Figure 5.11 Palette Manager

9. To be able to use the Palette Manager by drag-n-drop style to the Design window, make sure that the **Selection Mode** is enabled. It is the first icon beside the Design Tab. Figure 5.12 shows the Design Tab. The selection button should be highlighted; it is the button with the arrow and box.

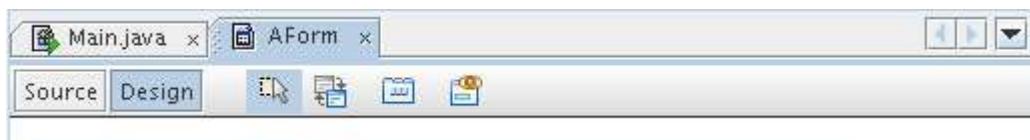


Figure 5.12 Selection Model Enabled

10. The Component Inspector consists of an overview of the GUIs added to the Frame. Figure 5.13 shows the Component Inspector. It is part of the window labeled as Inspector. A Tree View of the components is used to show the list of components.

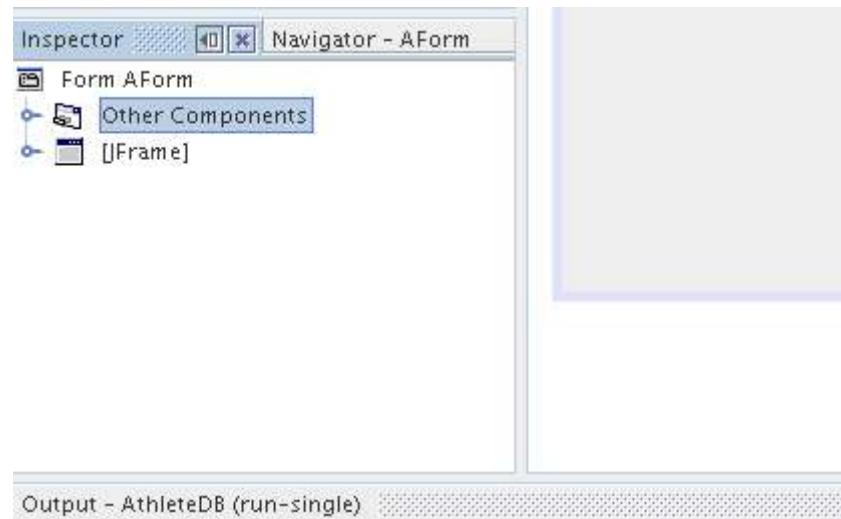


Figure 5.13 Component Inspector

11. To build or save the project, click **Build -> Build Main Project**. This is shown by Figure 5.14.

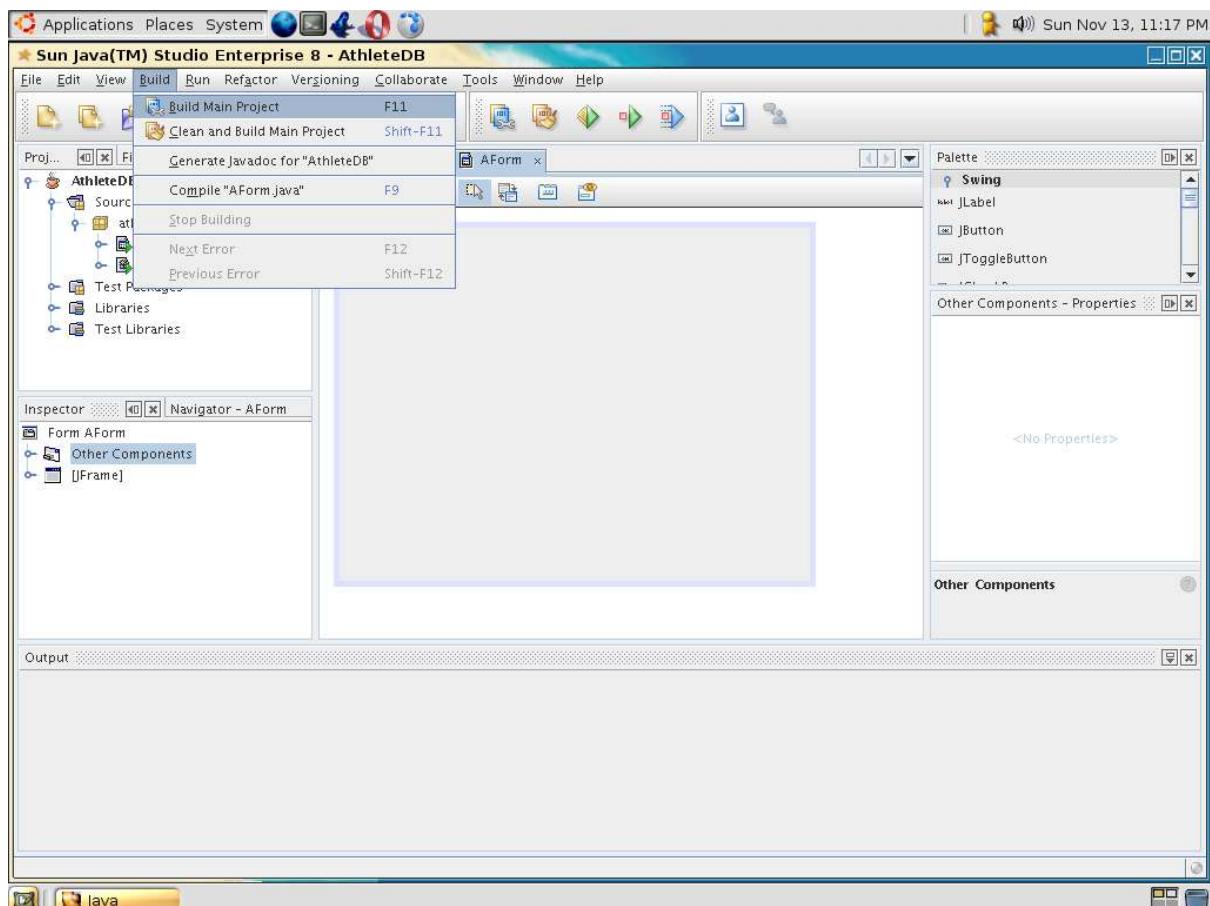


Figure 5.14 Build Option

The Build Output is generated at the Output Tab at the bottom of the program. This is shown in Figure 5.15.

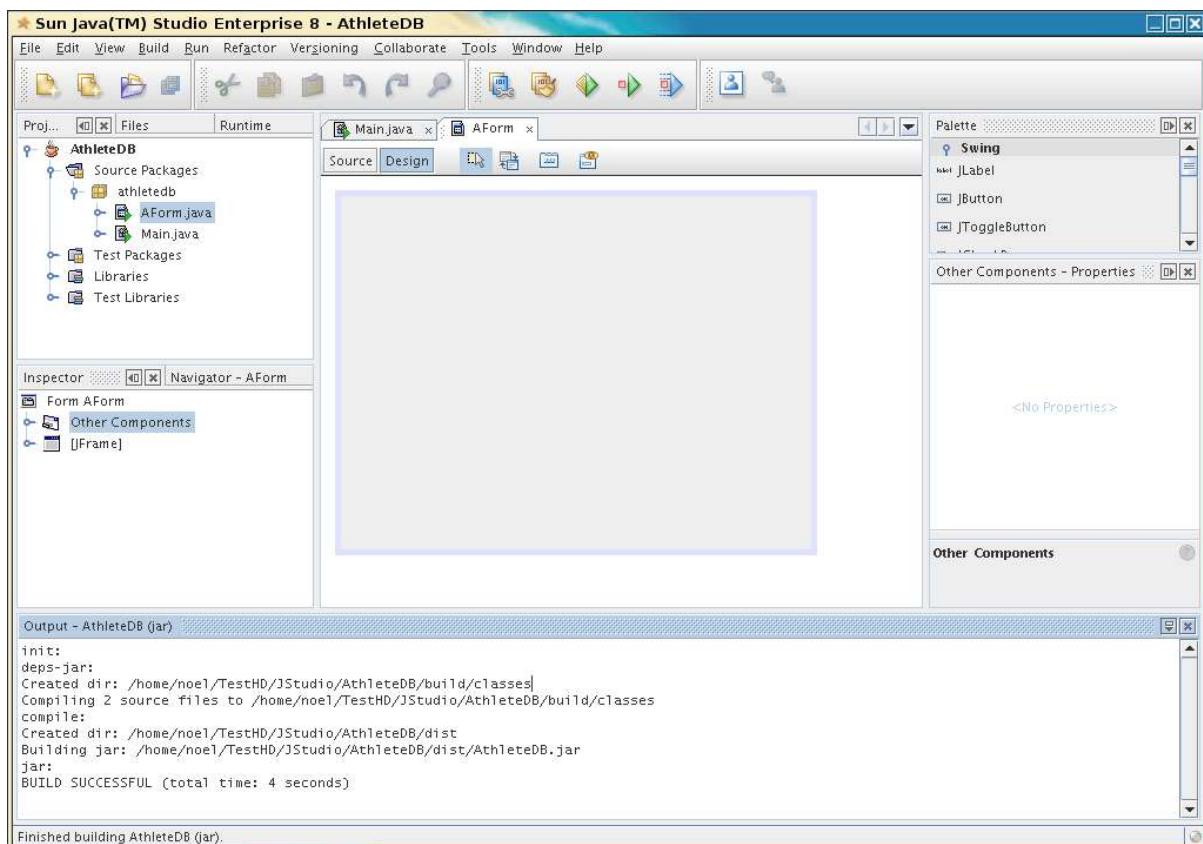


Figure 5.15 Build Output

To run the specific program, click **Run -> Run File -> Run "Aform.java"**. This is shown in Figure 5.16.

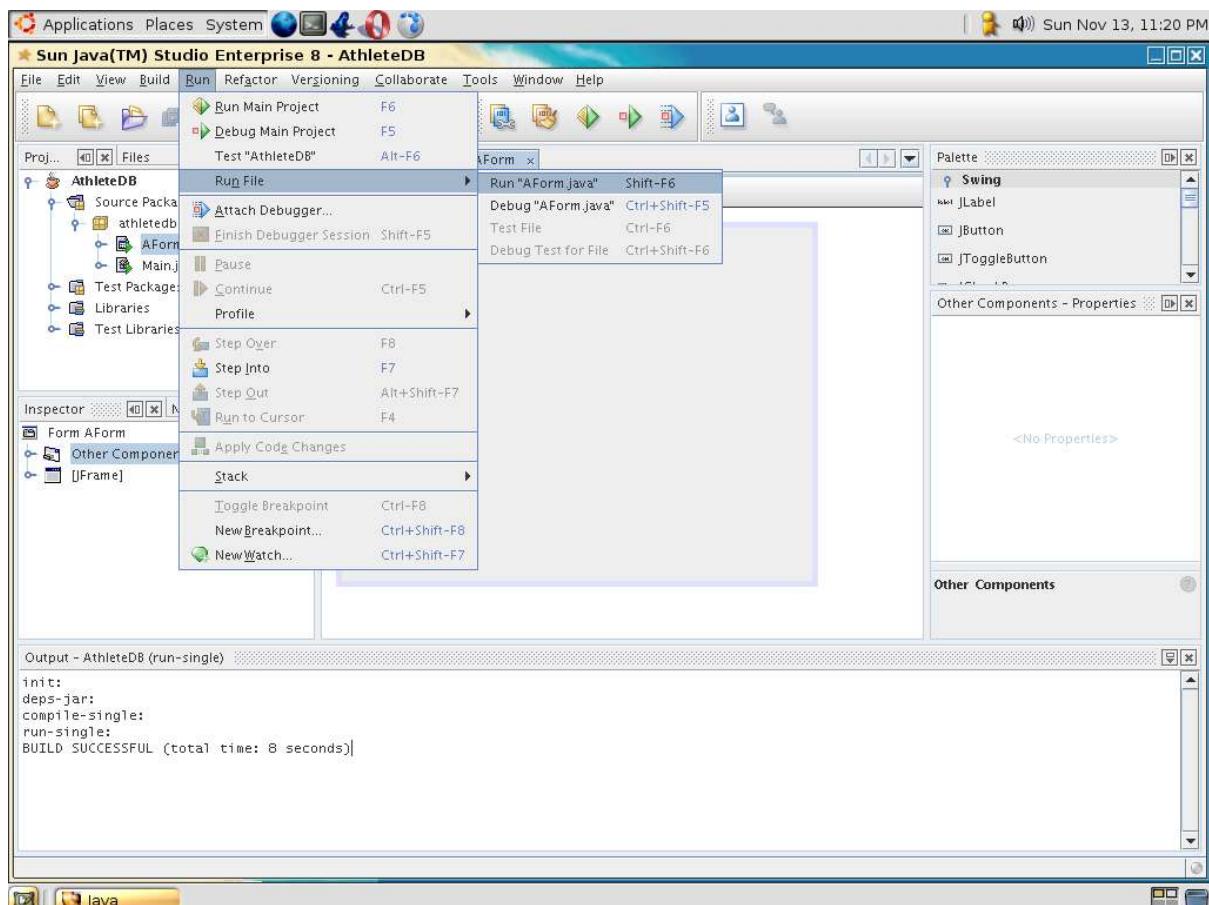


Figure 5.16 Run Option

Drafting the Sample Athlete Form Application

1. Right-click the **JFrame** in the **Component Inspector**. It will open a menu containing its layout as well as the other properties. The **GridLayout** is selected as default. Look at Figure 5.17 to see how this is done.

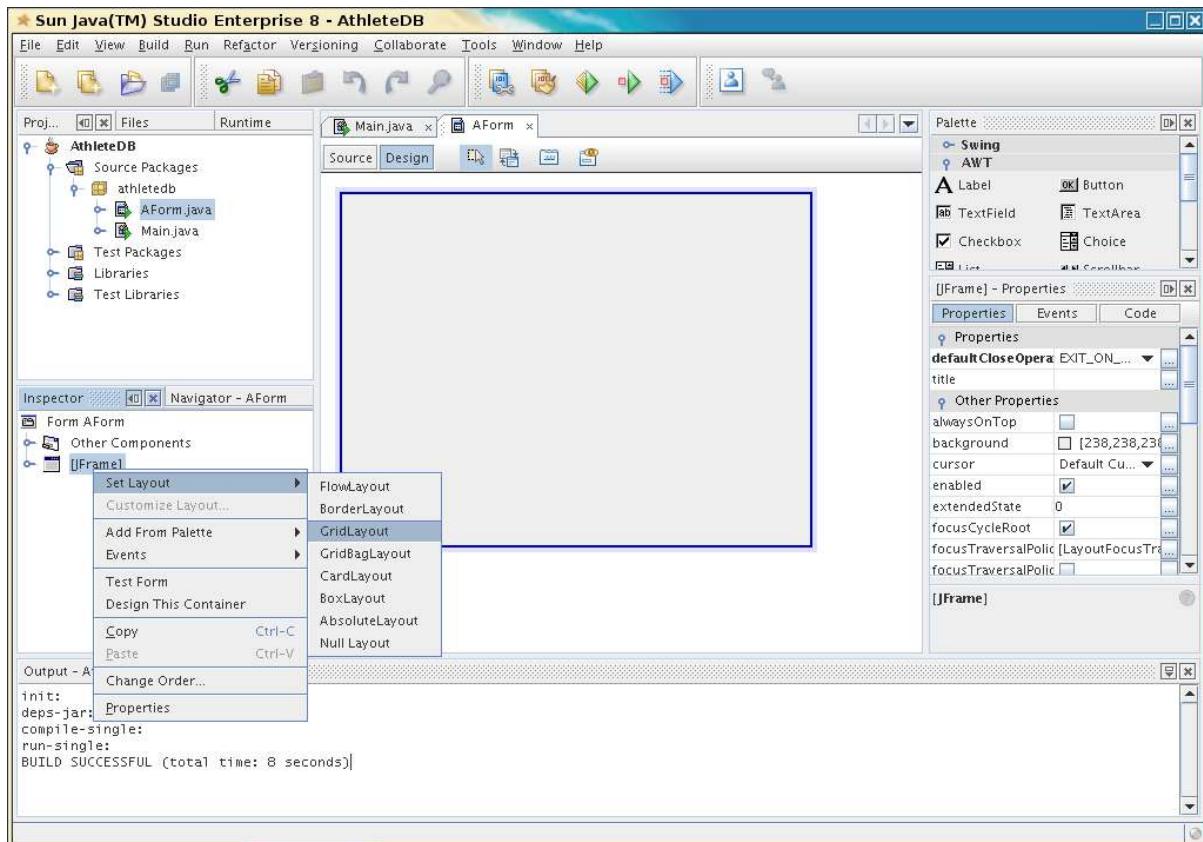


Figure 5.17 Setting the Layout

2. Looking at the **Properties Tab** below the Palette Manager, the properties for **GridLayout** is illustrated and it can be modified as well. Figure 5.18 shows an instance of the **Properties Tab**.

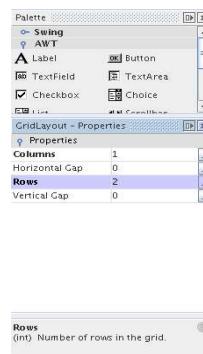


Figure 5.18
Properties Tab

3. To add a panel, right-click the name of the frame in the Component Inspector window. In this example, it is **Jframe1**. Click **Add from Palette -> AWT -> Panel**. Figure 5.19 shows this.

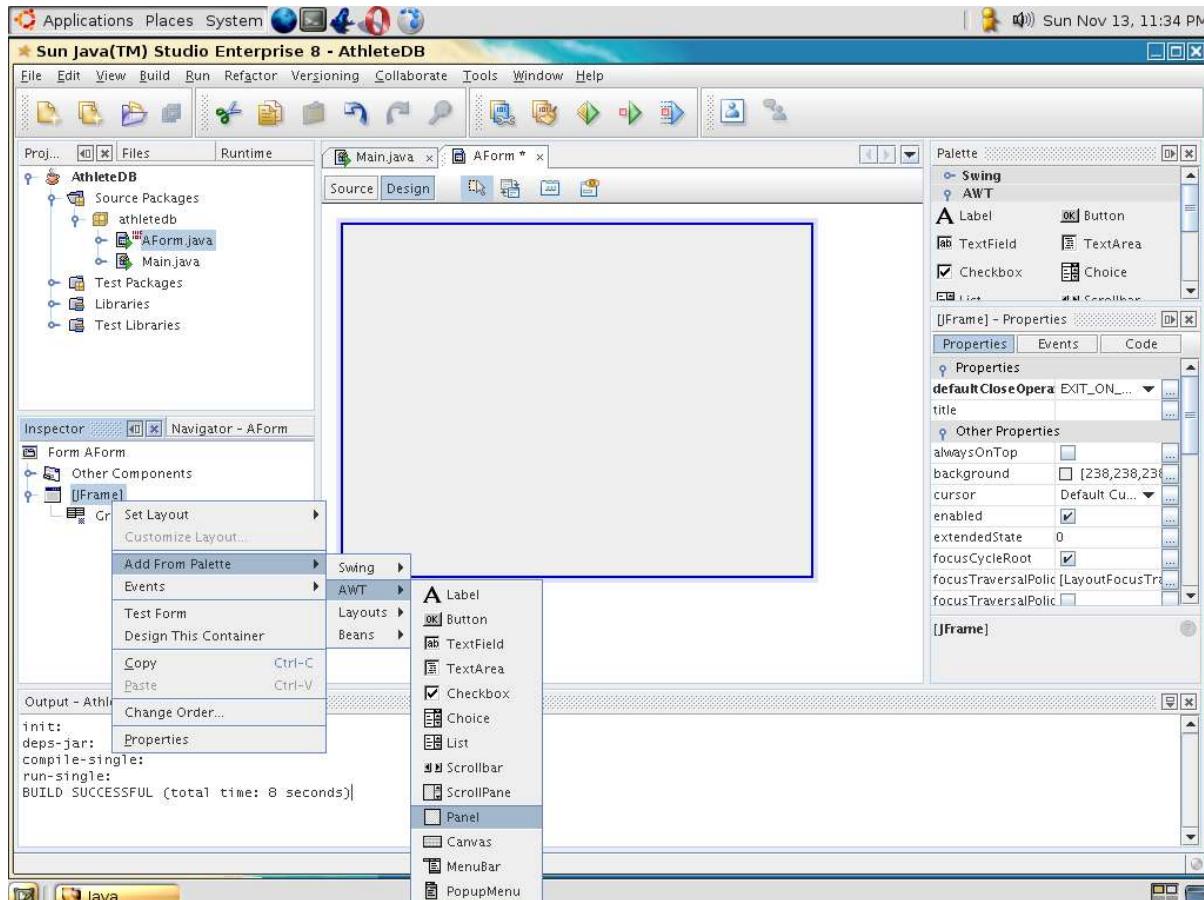


Figure 5.19 Adding a Panel

To rename the panel, right-click on the name of the panel in the Component Inspector window. Select **Rename**. Enter the new name of the panel. Figure 5.20 shows this.

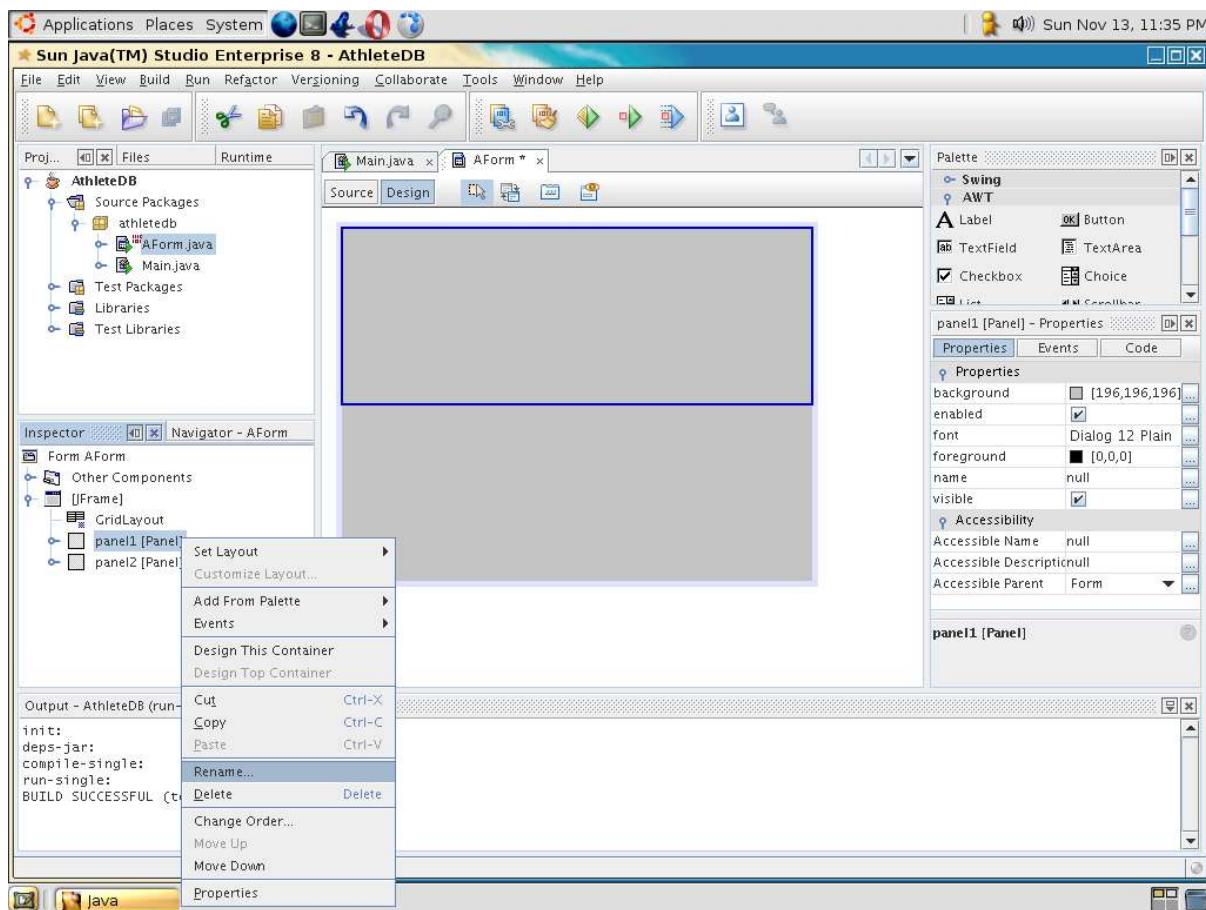


Figure 5.20 Renaming a Panel

4. Add Labels by DragNDrop style. Click the **Selection Mode icon** and the **Label icon**. Then, drag and drop it to the Design window. Notice the message at the footer of the program showing **Add Label to p1**. Figure 5.21 shows this.

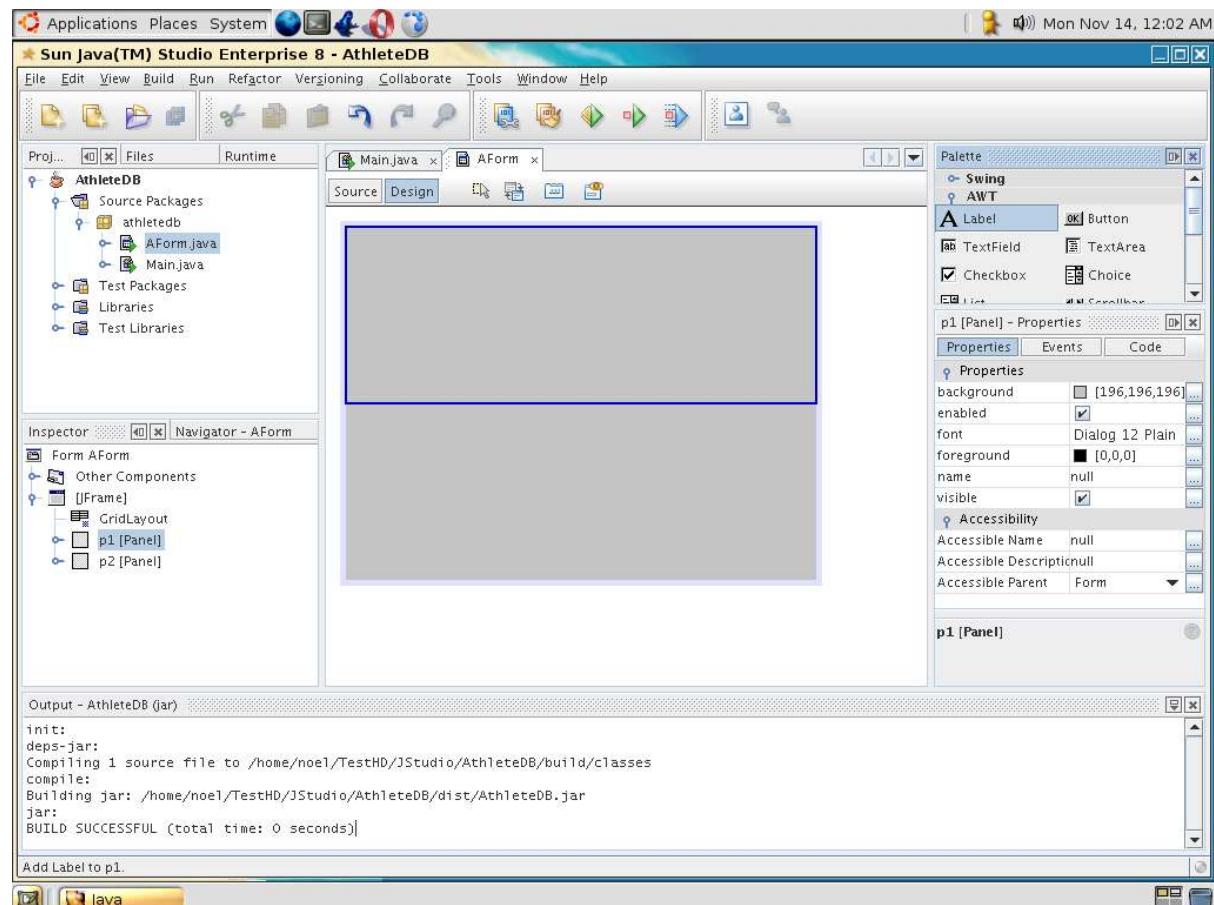


Figure 5.21 Adding a Label

5. The **Component Inspector** shows now that Label **label1** was added to Panel **p1** and the **Properties Tab** now shows the Properties, Events, and Code for that label. Figure 5.22 shows this.

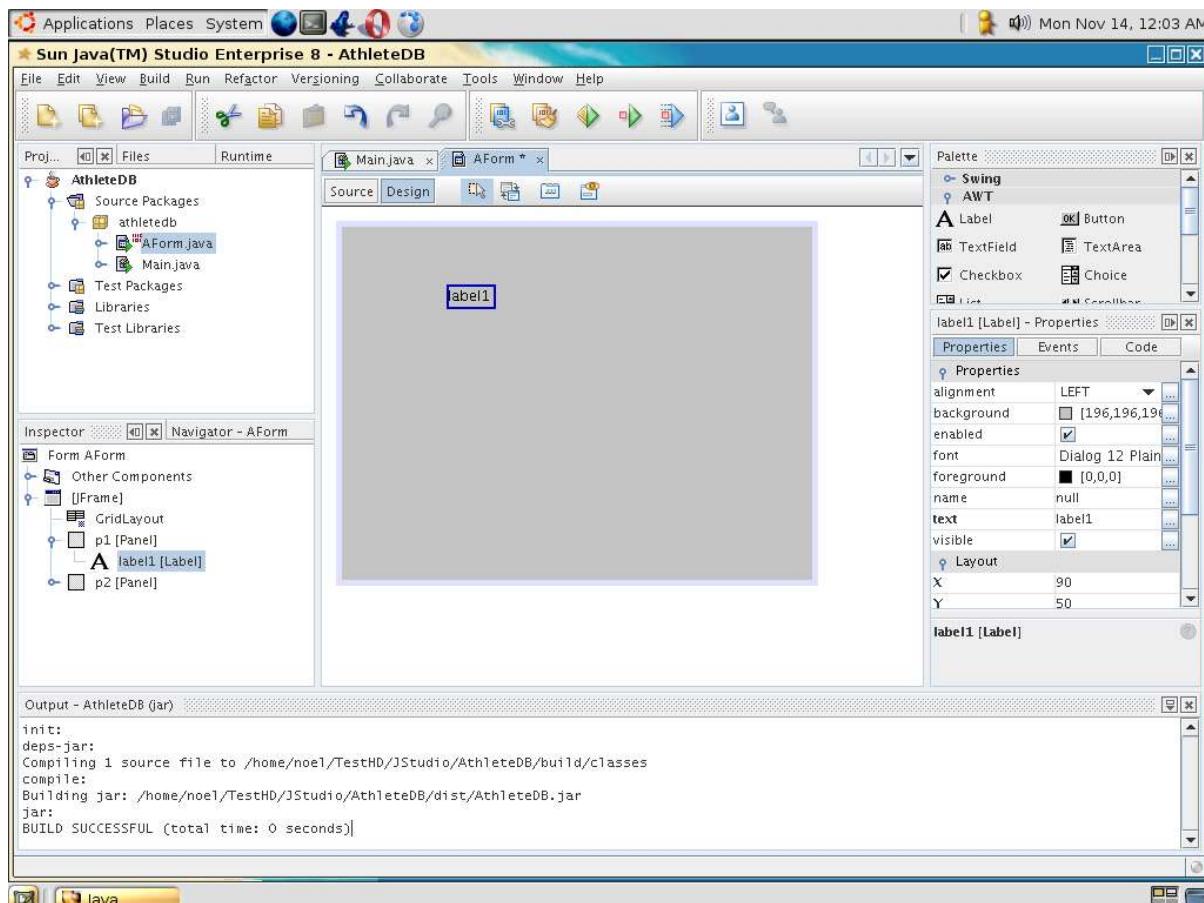


Figure 5.22 Sample Label1 and Properties

6. Adding GUIs to the Design can also be made by right-clicking the frame or panel in the **Component Inspector**. This is shown in Figure 5.23.

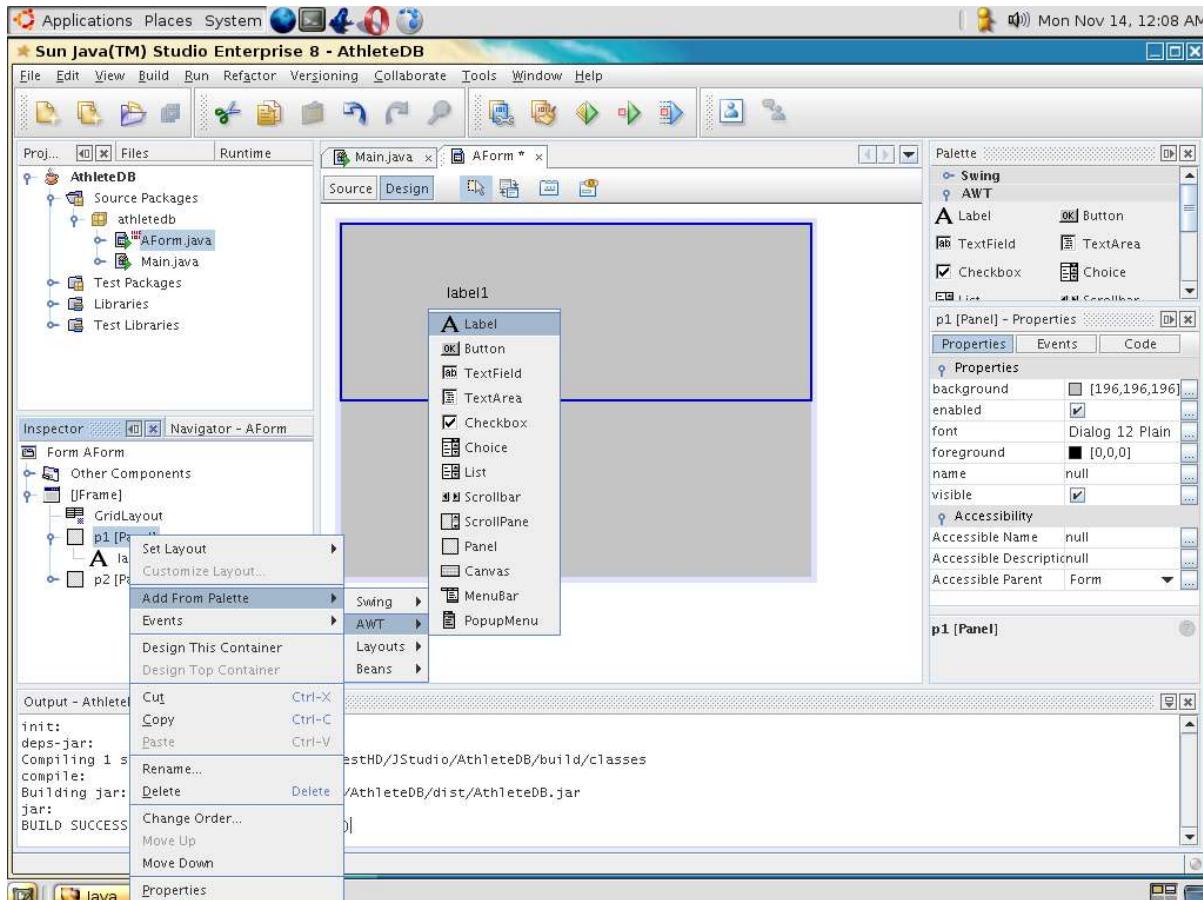


Figure 5.23 Right-clicking Frame or Panel

7. Continue adding and modifying the properties of the succeeding Labels, TextFields, TextArea, Buttons and the Check boxes to Panel **p1** and do the same in Panel **p2**. You can also drag and drop between panels as well. A sample of **AForm.java** is shown in Figure 5.24 with additional windows component.

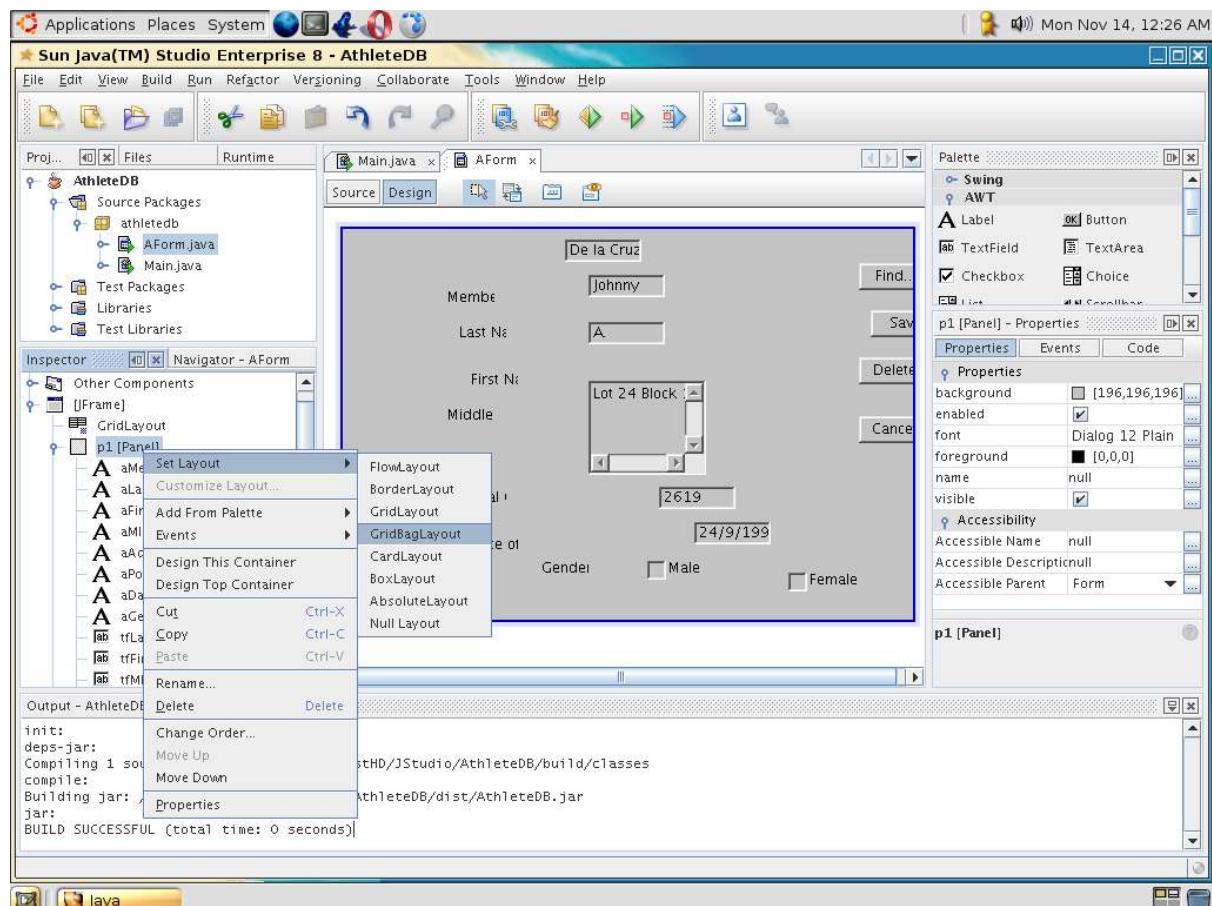


Figure 5.24 Sample AForm.java

9. To test the form, click the **Test Form Icon** at the Design Panel to see the current layout. Figure 5.25 and Figure 5.26 show what happens.

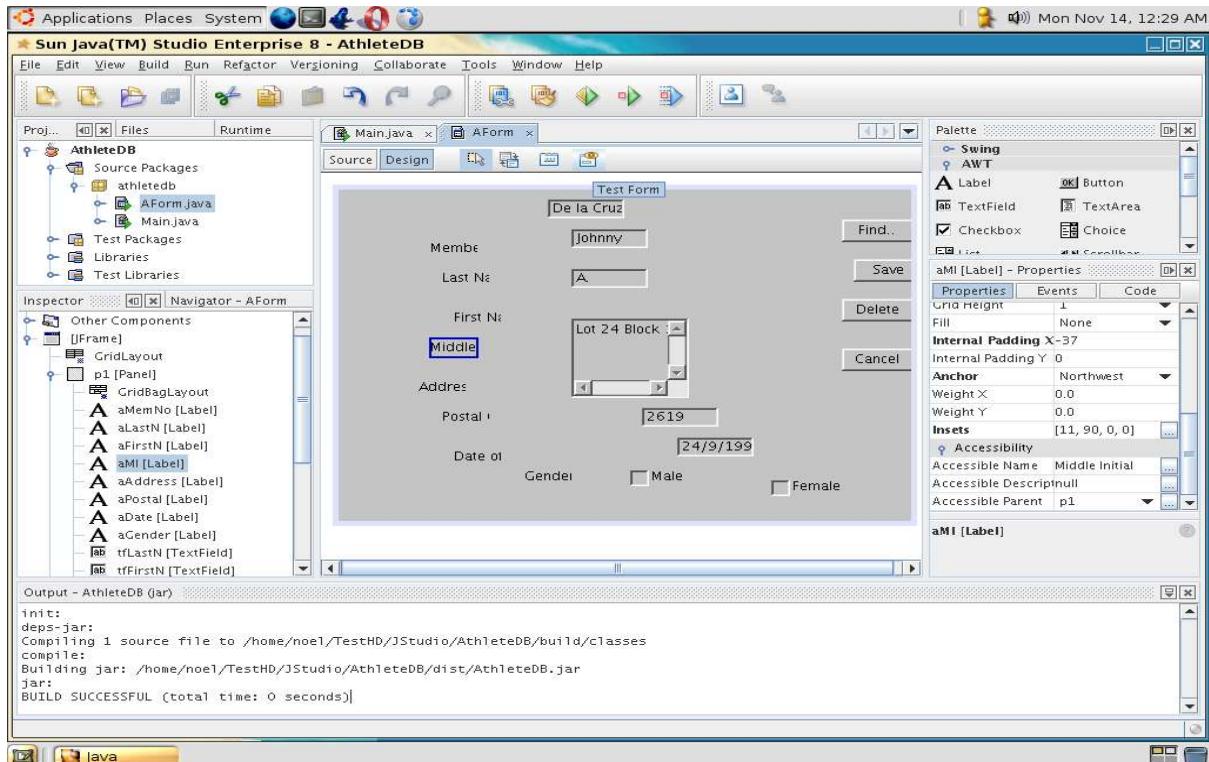


Figure 5.25 Clicking Test Form Icon

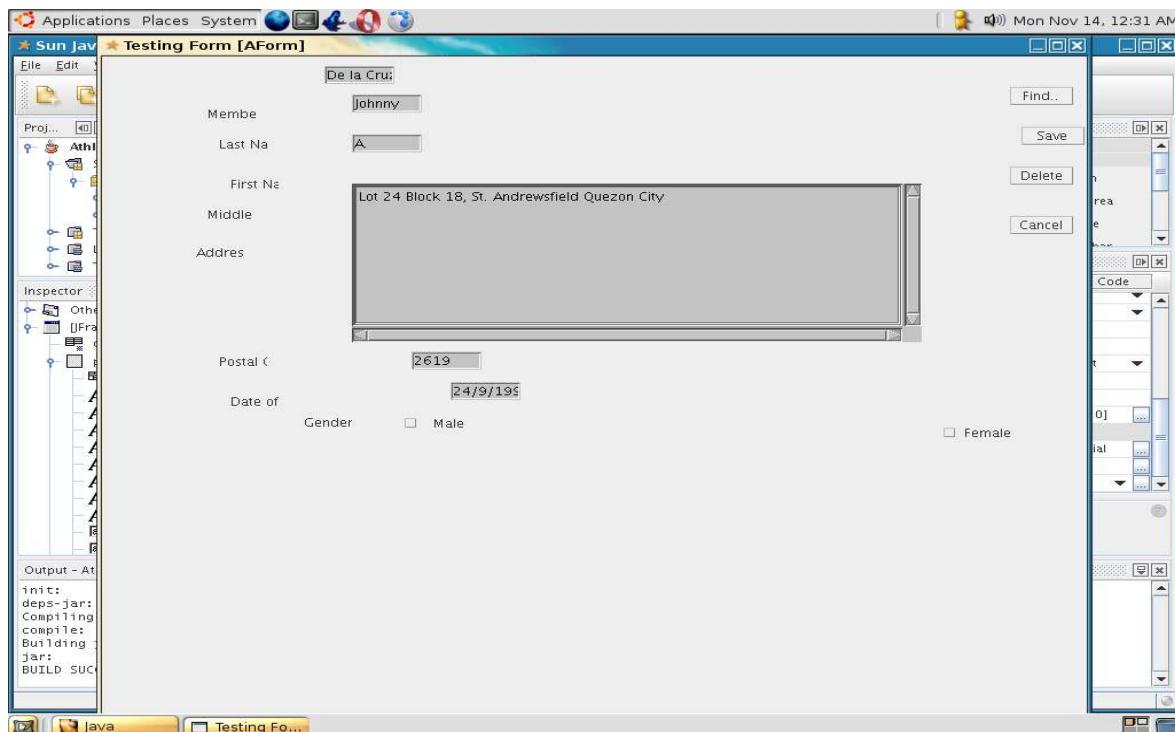


Figure 5.26 Running the Test Form

10.Right-Click **GridBagLayout** at the **Component Inspector**. Select **Customize** to be able to fix the layouts by dragNdrop style as well. The Customizer Window will be displayed. Figure 5.27 shows an example.

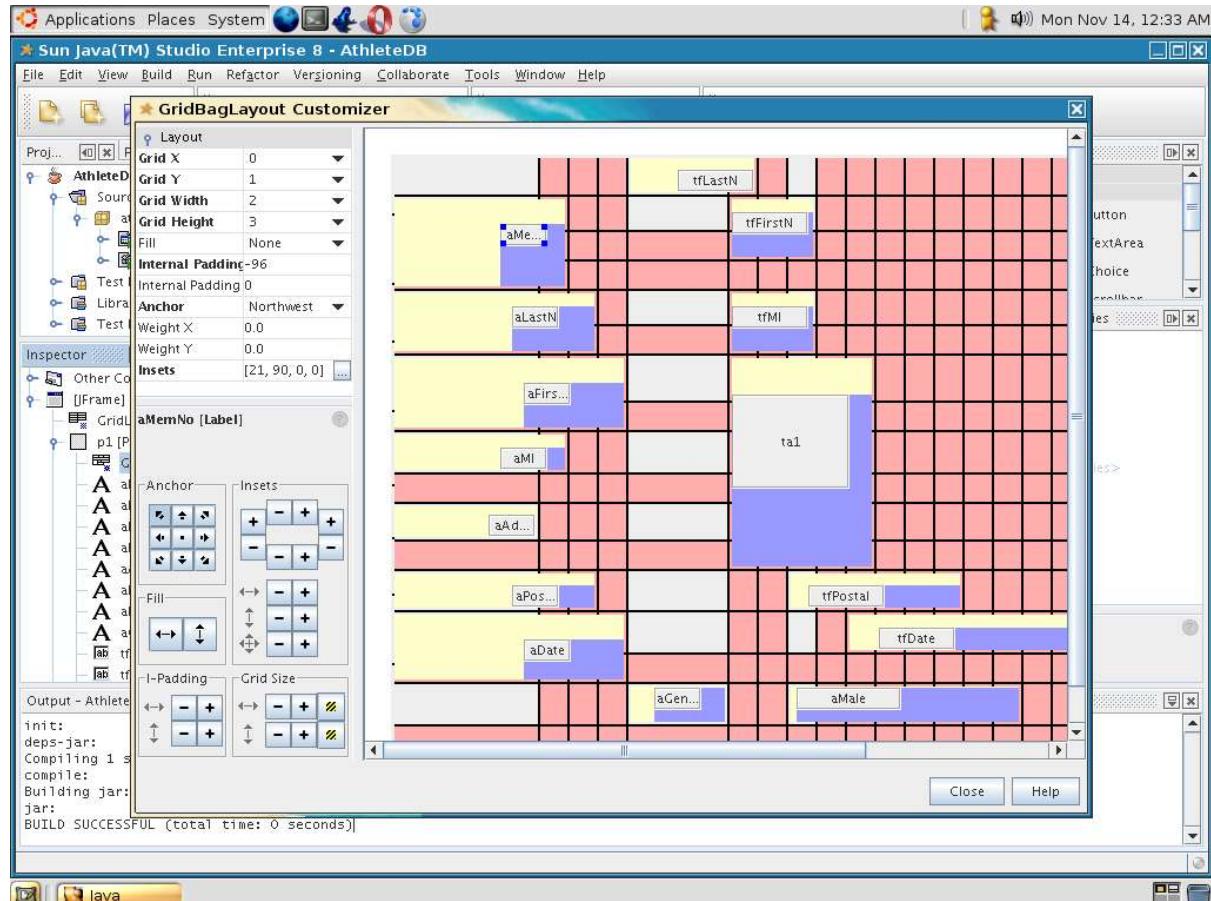


Figure 5.27 GridBagLayout Customizer

11. DragnDrop the components in their right locations in the way it is shown on the Figure 5.28.

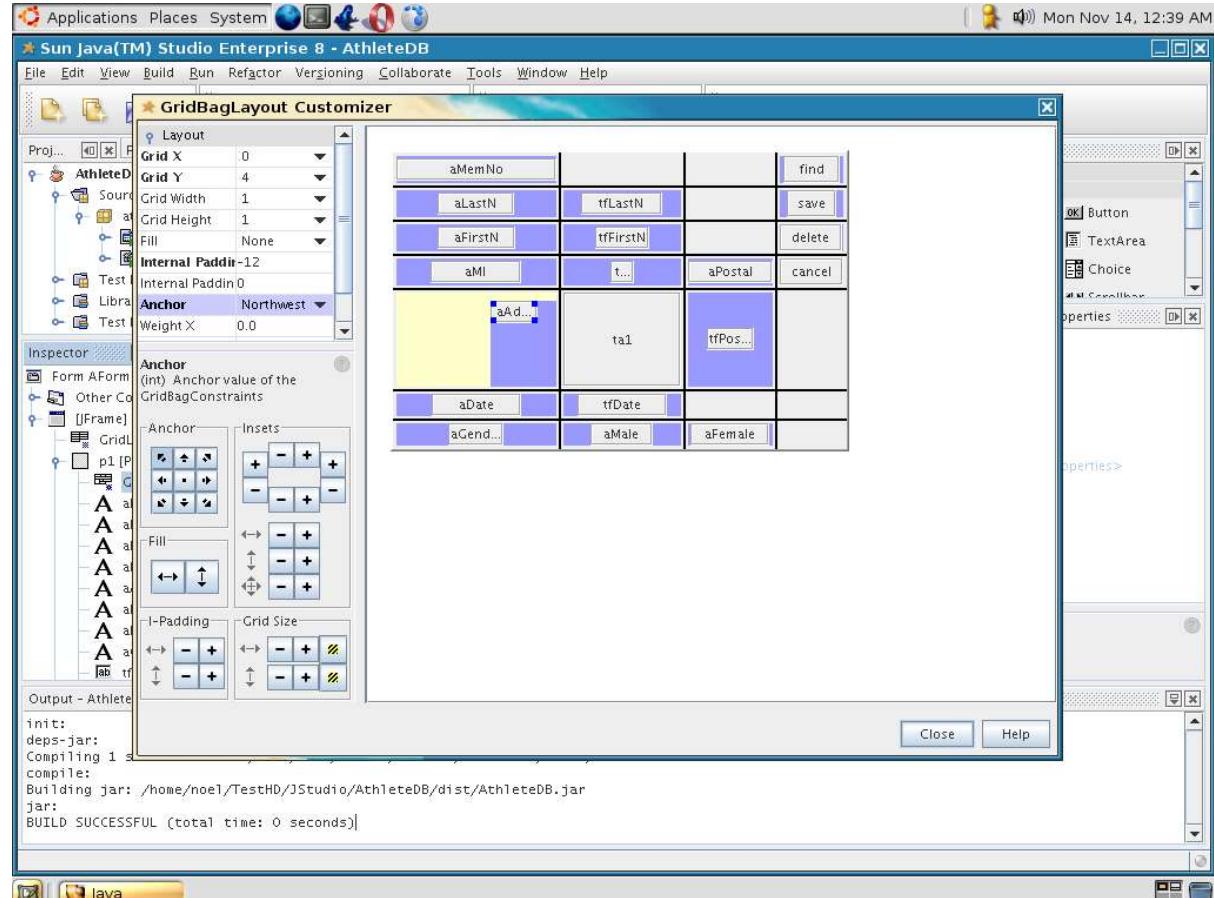


Figure 5.28 New Positions

Close it and click the **Test Form** icon to see the differences. The screen should more or less look like the screen in Figure 5.11.

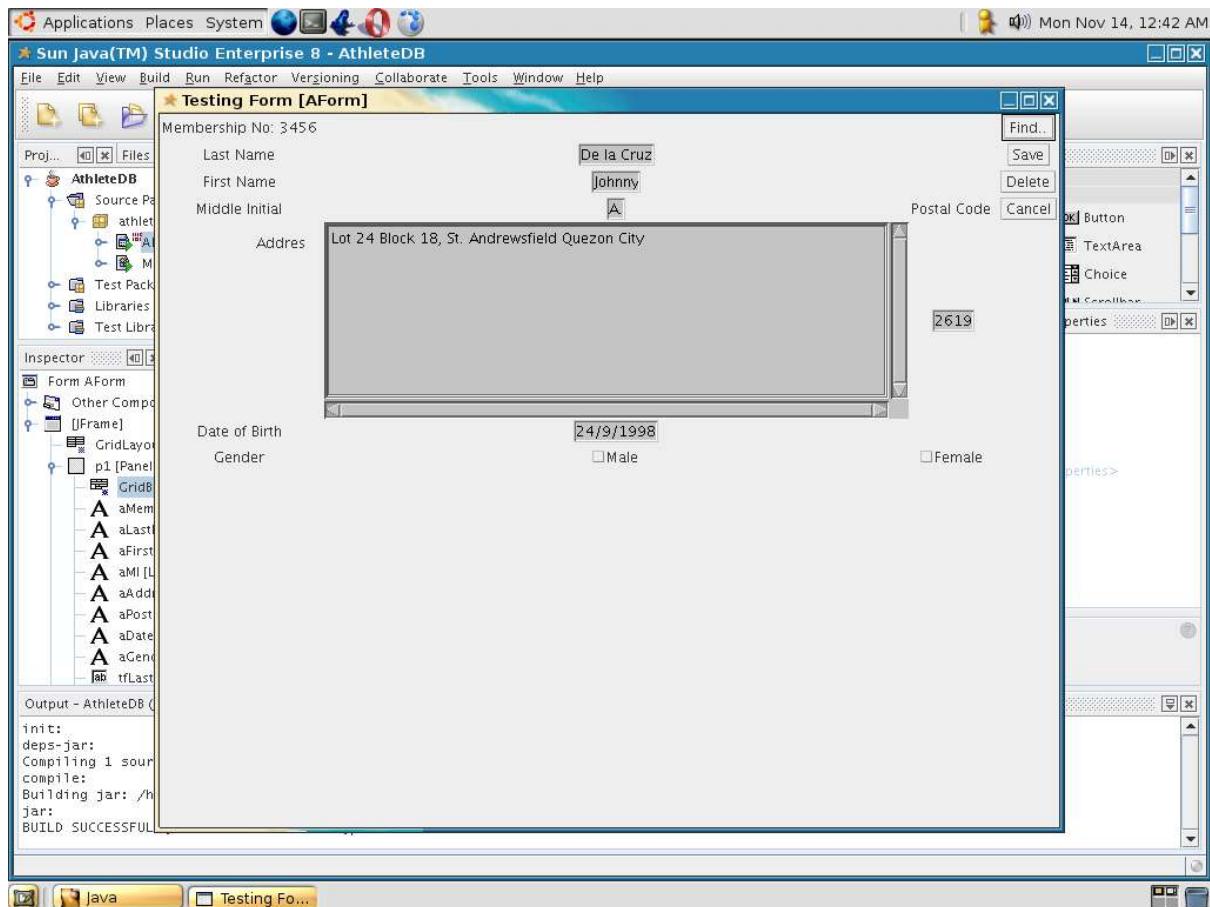


Figure 5.29 New GridBagLayout

Notice that everything is centered. Go back to the **GridBagLayout -> Customize**, to edit the Anchor values of each, as well as other necessary modifications. The final layout when AForm.java is executed (BUILD or Shift+F6) should look like Figure 5.3.

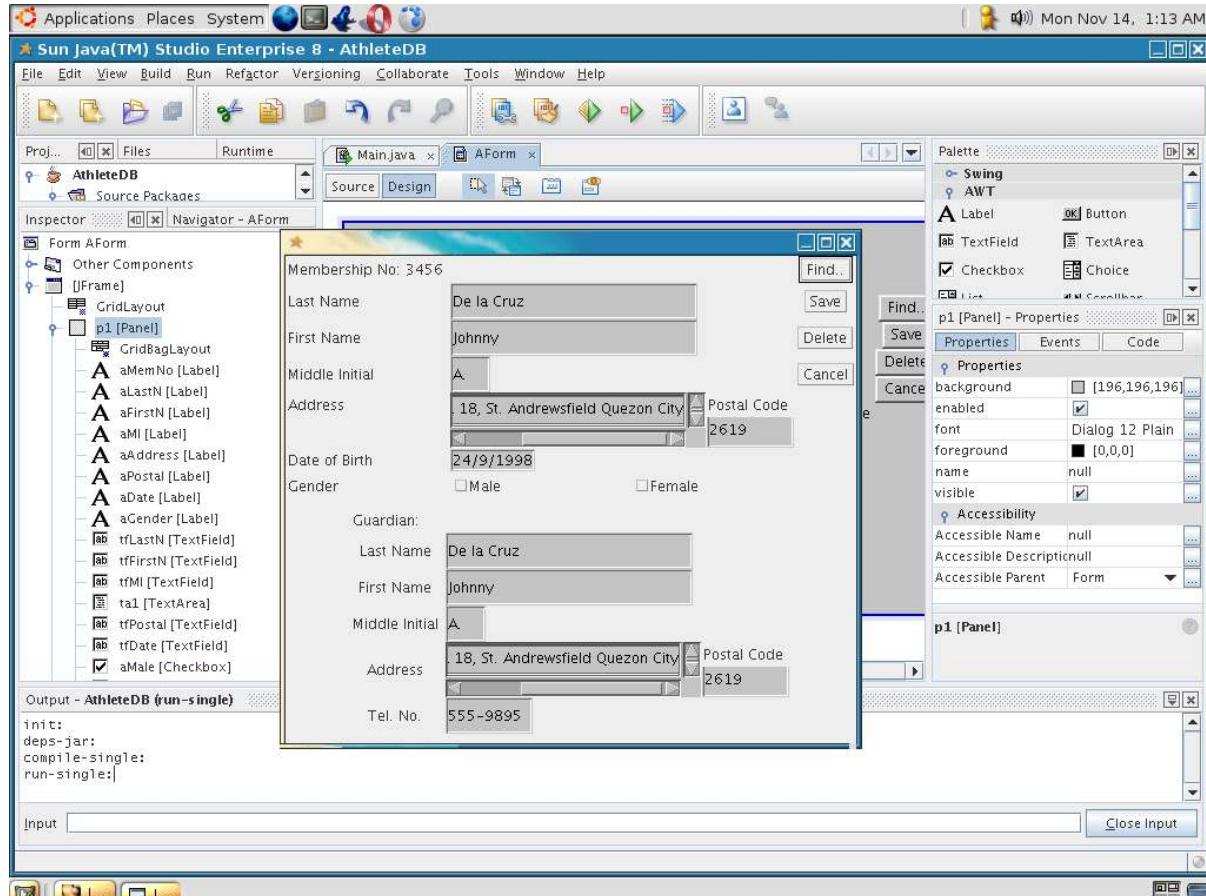


Figure 5.30 New Form Layout

12. To add dialogs to the form, go to **Component Inspector**, right-Click **Other Components**. Select **Swing -> JDialog**. Do the same process for the layouts.

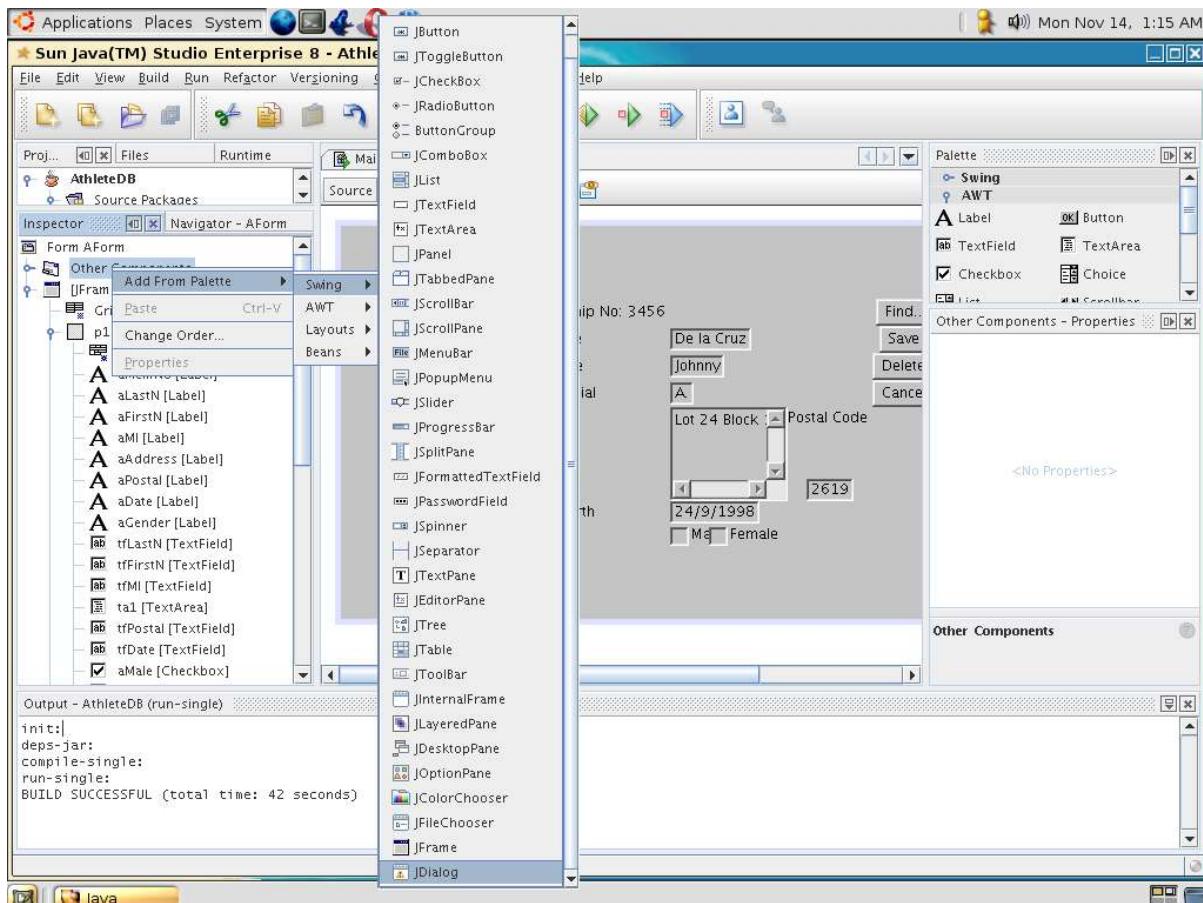


Figure 5.31 Adding a Dialog

13. To enable the event for the **Find** button in the **Athlete Form**, right-click its name at the **Component Inspector**. Select **Events -> Action -> actionPerformed**.

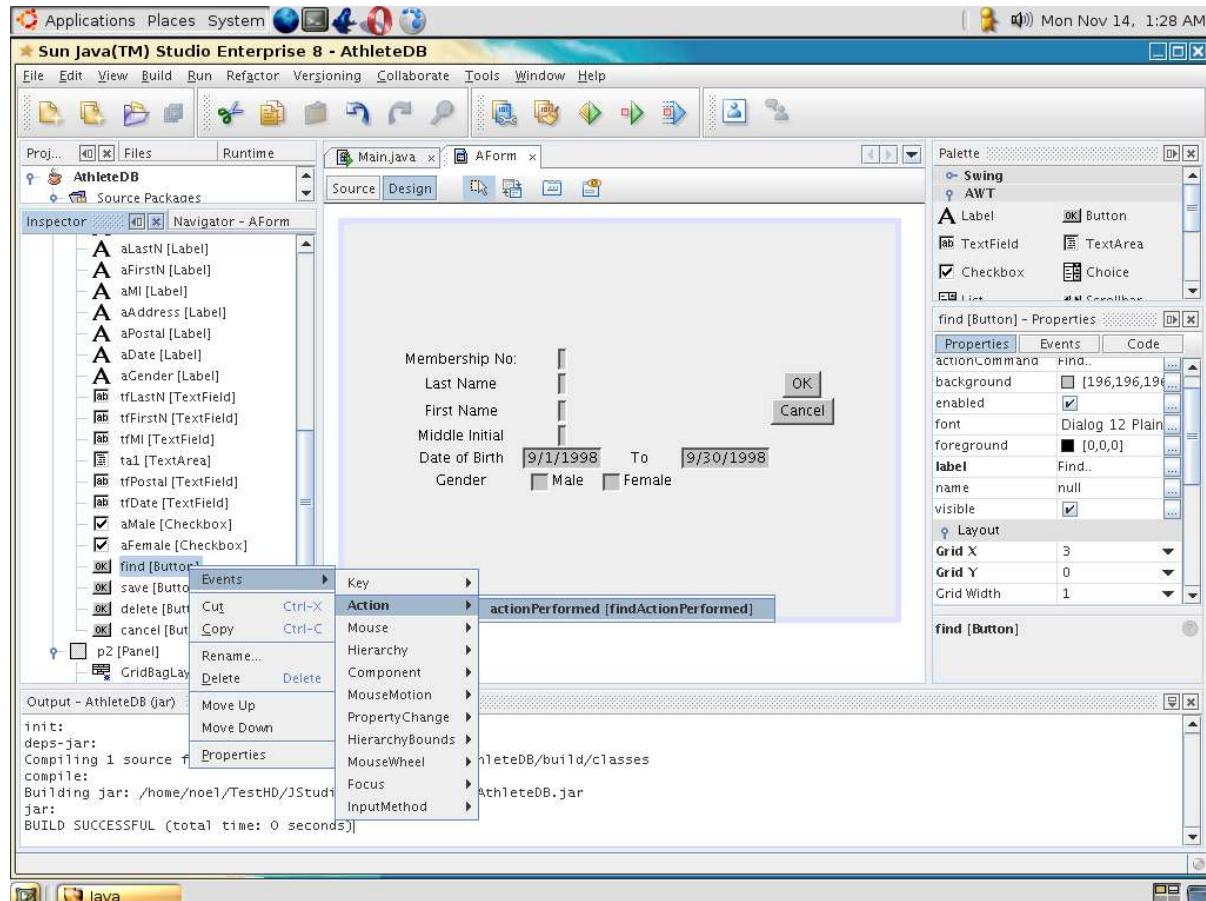


Figure 5.32 Enabling Events for the Form

This will select the **Source** tab beside the Design tab of the Design window. Modify the code in such a way that it will make the **Find An Athlete** Dialog visible. Do the same for the **OK** and **Cancel** Button in the Dialogs.

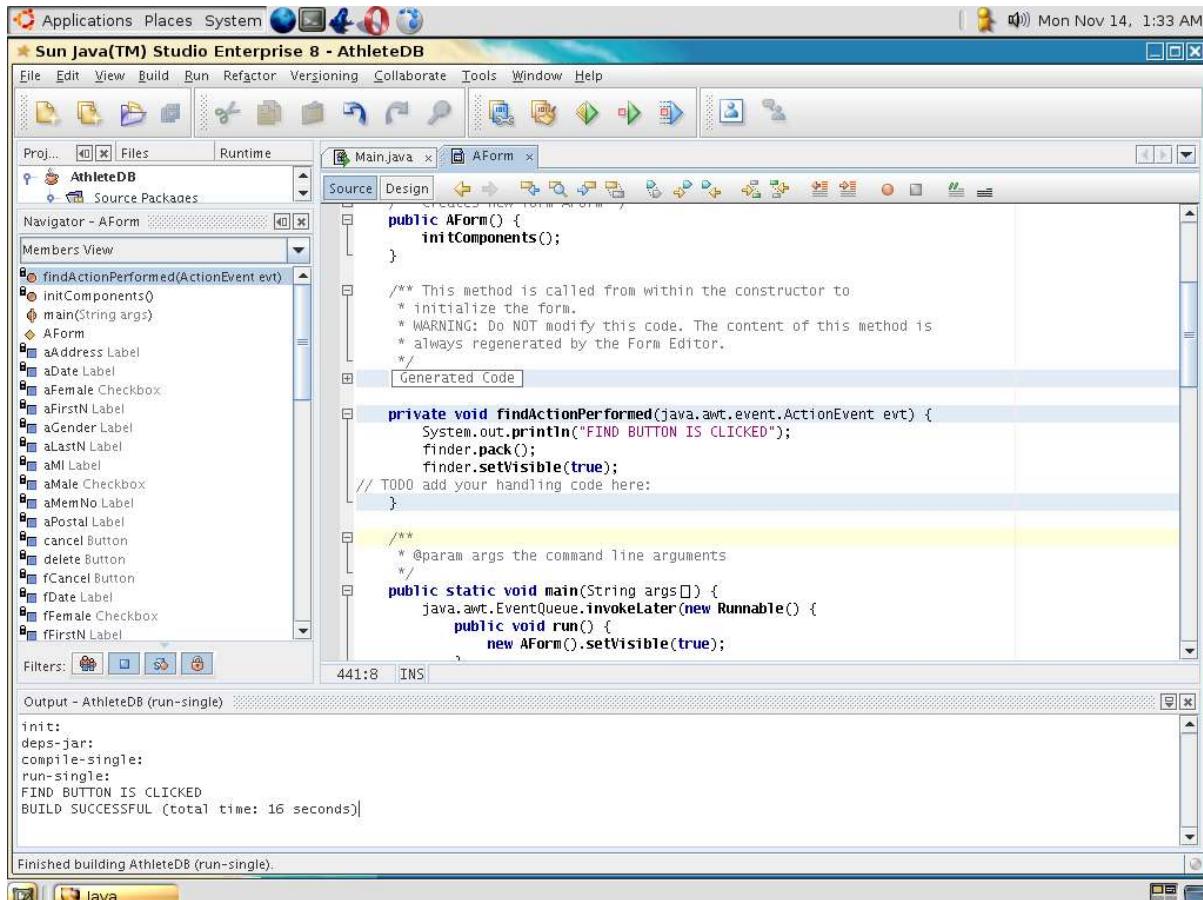


Figure 5.33 Setting Find Athlete Dialog Visible

14. Test the form. From the **Athlete** Form, click **Find** Button. It should display **Find An Athlete** Form. From this form, click **OK** button. It should display **Athlete List** Form.

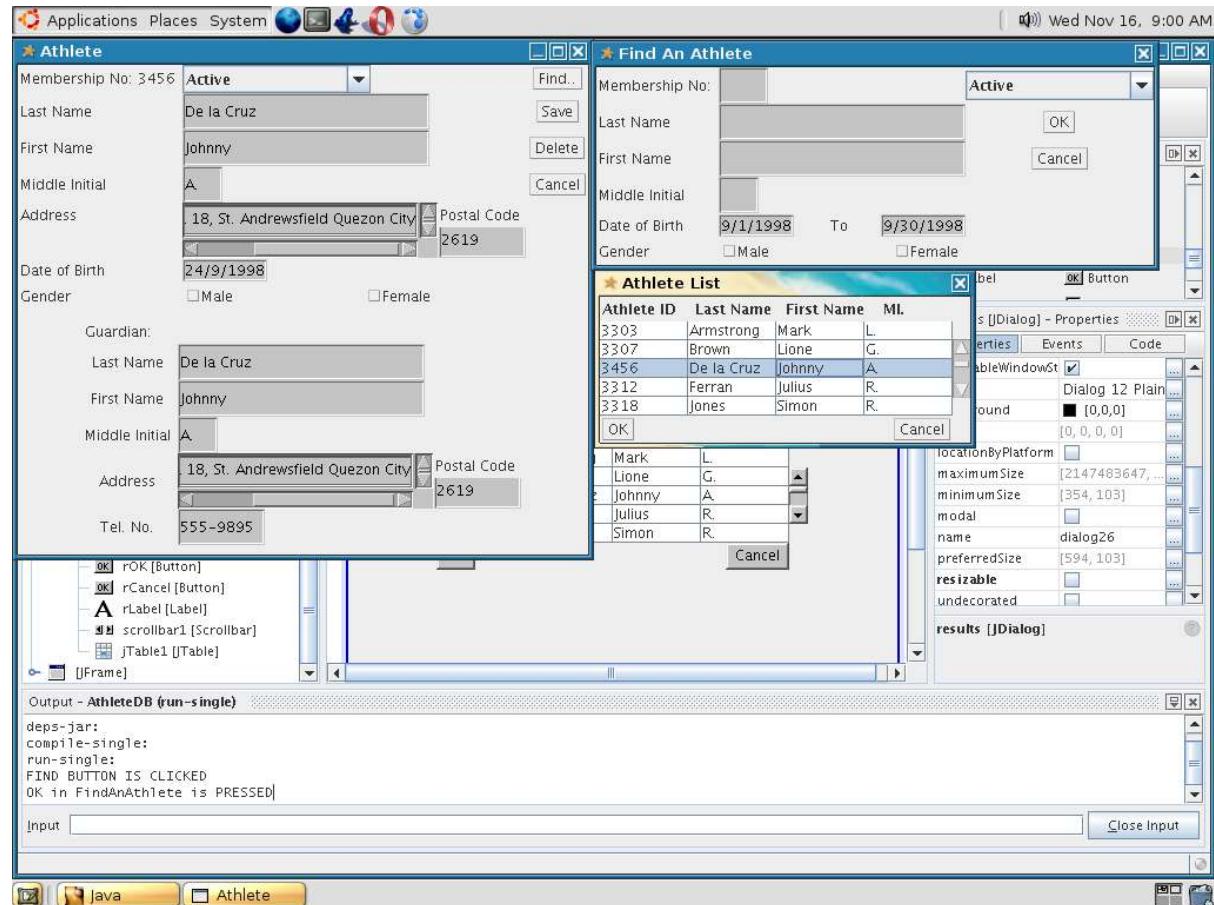


Figure 5.34 Testing all the forms

15. Rebuild and Run.

5.7 Controlling the Version of the Software

It is very difficult to control the development of the software without the aid of Versioning System. This section introduces the Concurrent Versioning System (CVS) and how it is used with Java™ Studio Enterprise 8. It assumes that there is a CVS server available for storing software packages.

What is CVS?

CVS is an abbreviation of **C**oncurrent **V**ersioning **S**ystem. It is an open-source version control and collaboration system. Simply put, it is a software package that manages software development done by a team. Using it, you can record the history of sources files, and documents.

Collaboration with CVS

For teams to be able to collaborate and get updates of their work, the following should be done:

- *Setting up a CVS repository.* A repository is a persistent store that coordinates multi-user access to the resources being developed by a team.
- *Specify CVS repository location.*
- *Commit the work into that repository.*

After doing these steps, team members may now update the sources versions of the project they are doing.

Setting up CVS repository in Sun Java™ Studio Enterprise 8

Setting up CVS repository in Sun Java™ Studio Enterprise 8. Initializing the CVS repository in JSE8 is simple. Assume that you want to share the code shown in Text 13 to the other team members. Figure 5.2 shows this in the Java Studio Enterprise 8. To initialize, select **Versioning → Version Manager**. Click **Add** and the *New Generic VCS wizard* pops up that allows you to input the necessary versioning settings. In the **Version Control System Profile**, choose **CVS** and specify the repository path at the **Repository Path** field. Also, you should specify the **CVS Server Type**. This is important in order to let other team members get a hold of the sources in the project. The default value is local. You may require assistance from your repository administrator in order to fill in the necessary information. Figure 5.1 shows initialization of the CVS.

```
/*
 * Main.java
 *
 * Created on November 16, 2005, 2:54 PM
 *
 * To change this template, choose Tools
 * -> Options and locate the template under
 * the Source Creation and Management node.
 * Right-click the template and choose
 * Open. You can then make changes to the
 * template in the Source Editor.
 */

package welcome;
import javax.swing.JOptionPane;
/**
 *
 * @author Argel
 */
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(
            null, "Welcome\\nto\\nJEDI\\nProject");
        System.exit(0);
    }
}
```

Text 13 Sample CVS Code

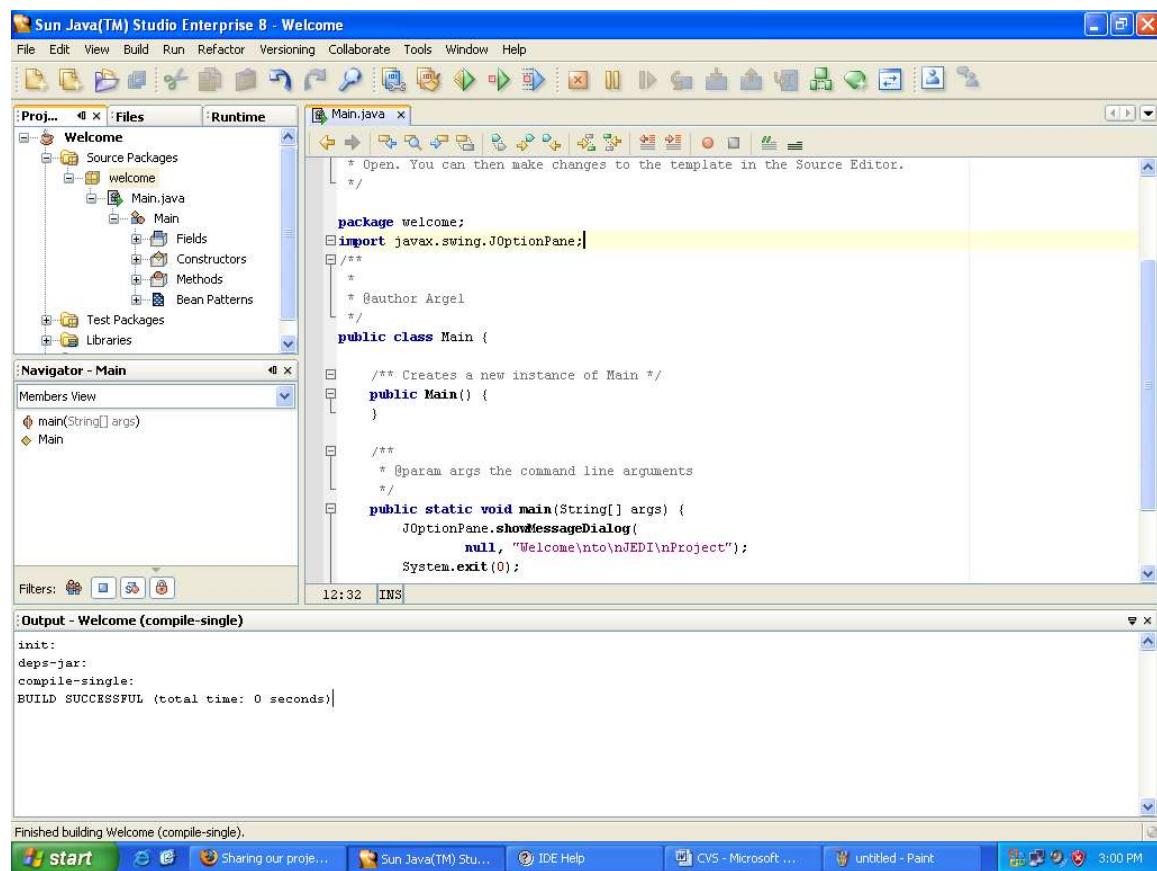


Figure 5.35 Sample Code to be Shared

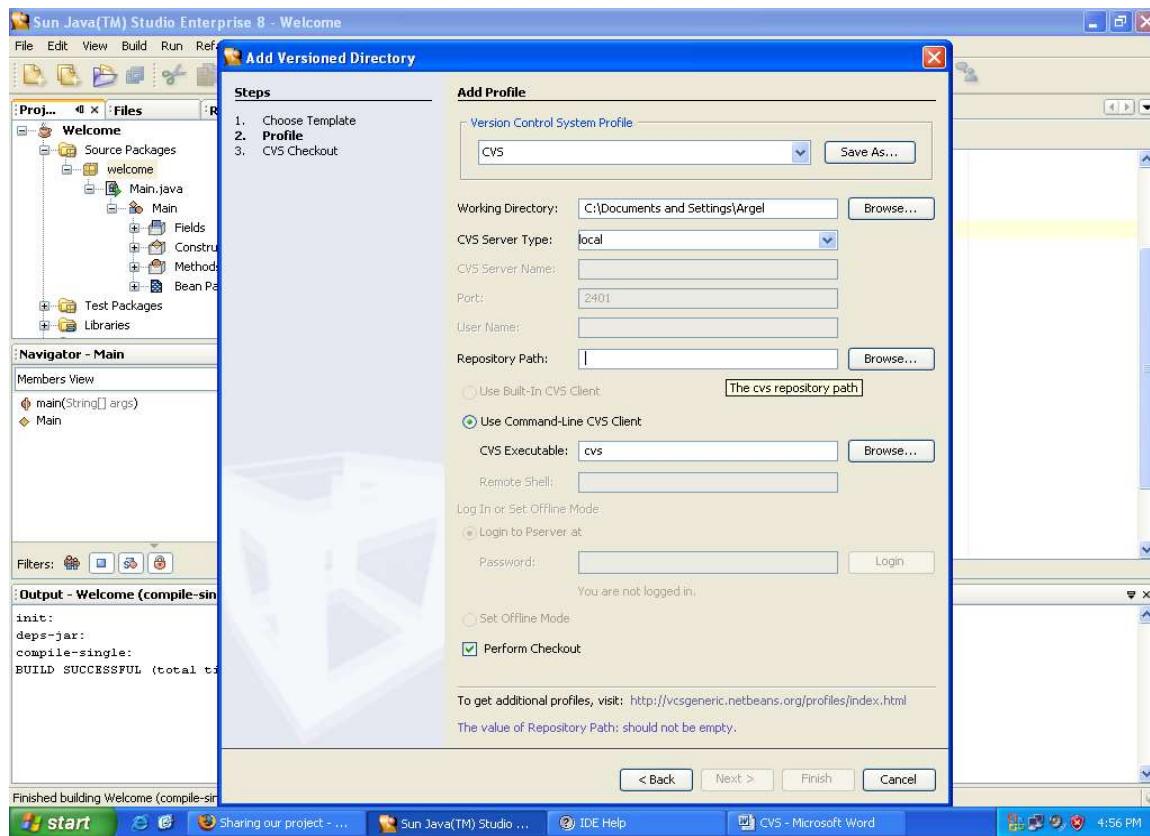


Figure 5.36 Initializing the Repository

CVS Checkout Repository

From the **CVS** option under **Versioning**, we can find the **CVS Checkout** option. This leads to the CVS Checkout Page, which may be used to specify the modules you want to check out. If you want to check out the entire repository select the **All** check box and to check out specific components, select the **Module(s)** check box and enter the path to the directory in the **Modules field**. Moreover, you can just click the Select button to choose from a list of all of the modules in the repository.

Updates and Collaboration

Team members may view the repository by selecting **Window→Versioning→Versioning**. This will open a window in the upper left corner that will show the repository's contents. At the upper right corner, you may add accounts for your teammates so that they may have access to the repository. Any updates and modification done by other team members can be seen here through the history log as shown in figure 3. Just right-click the repository and this men will show you what you need. This will window also shows tools that will help you test the modules modified by the other team members. By clicking on **Tools** after right-clicking the repository it will give you an option to create tests. (See *JUnit testing*)

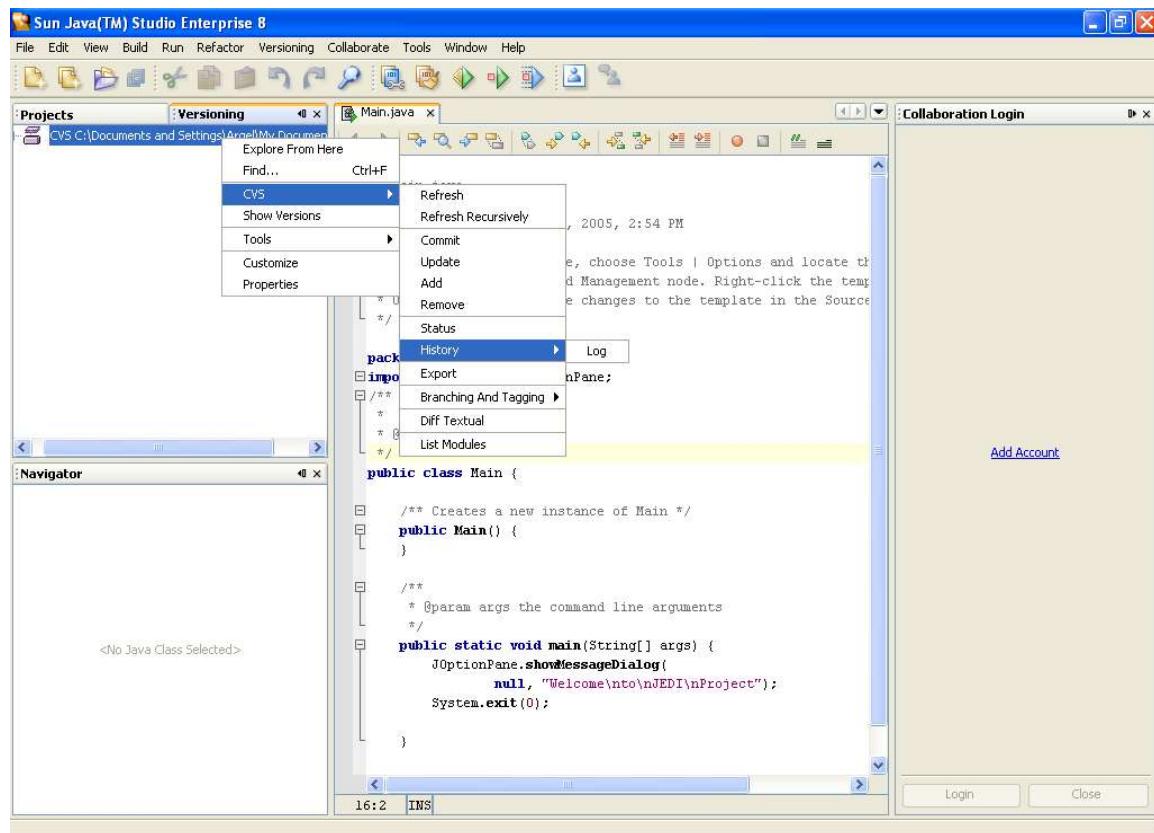


Figure 5.37 Versioning Window

CVS the Easy Way

Using the Sun Java™ Studio Enterprise 8 will surely make team collaboration in software projects easier. No extra software tool is needed: all you need is a working repository, team members that will help you in your project and Sun Microsystems' Java™ Studio 8!

5.8 Mapping Implementation Deliverables with the Requirements Traceability Matrix

Normally, no additional RTM elements are used for the implementation phase. Rather, we monitor the development of each software component defined under the Classes column. However, during the course of the development, additional classes may have been developed. They should be entered in this column under the appropriate package.

We can use the RTM to determine how many of the classes have been implemented, integrated and tested.

5.9 Implementation Metrics

Since the program characteristics must correspond to design characteristics, the metrics used for implementation are the metrics for the design. Other metrics considered at implementation phase are used for succeeding similar projects. They serve as history data that can be used for estimation for other projects.

1. *Lines of Code (LOC)*. This is the number of statement lines that was used. It can be for a program, component or entire software. It is basically used later on in managing similar projects.
2. *Number of Classes*. This is the total number of classes created.
3. *Number of Documentation Pages*. The total number of documentation produced.
4. *Cost*. The total cost of developing the software.
5. *Effort*. This is the total number of days or months the project was developed.

5.10 Exercises

5.10.1 Defining the Internal Documentation Format

1. Define the **Header Comment Block** that will be used for the Internal Documentation of the program. As a guide, the following are the recommended components:

- Component Name
- Author of the Component
- Date the Component was last created or modified
- Place where the component fits in the general system
- Details of the component's data structure, algorithm and control flow

Sometimes, it would help to place historical information about the program. The recommended components are:

- Who modified the component?
- When the component was modified?
- What was the modification?

The format should include how it will be written such as spacing, labels are in upper case etc. Provide a sample source file.

2. Using the Header Comment Block defined in number one for the Internal Documentation, implement the **AthleteRecordUI**, **FindAthleteRecordUI**, **AthleteListUI** and **FindAthleteRecord** designed in the previous chapter. If the database cannot be implemented at this point, all methods of the controller that use objects in the persistent and database layer are coded to display a message describing what the method is supposed to do. As an example, for the **public getAthleteRecord(String searchCriteria)** method the programmer simply displays a message dialog box to the user stating that this method returns a list of athlete records as specified by the value of **searchCriteria**.

5.11 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in this chapter. Particularly, they are:

1. Defining the header comment block
2. Tracking implementation using the Requirements Traceability Matrix

MAJOR WORK PRODUCTS:

1. Implemented Software
2. Installation and Configuration Files

6 Software Testing

Software is tested to detect defects or faults before they are given to the end-users. It is a known fact that it is very difficult to correct a fault once the software is in use. In this chapter, the methods and techniques to develop test cases will be discussed. Software testing strategies are also presented to help us understand the steps needed to plan a series of steps that will result in the successful construction of the software.

6.1 Introduction to Software Testing

Software is tested to demonstrate the existence of a fault or defect because the goal of software testing is to discover them. In a sense, a test is successful only when a fault is detected or is a result of the failure of the testing procedure. **Fault Identification** is the process of identifying the cause of failure; they may be several. **Fault Correction and Removal** is the process of making changes to the software and system to remove the fault.

Software testing encompasses a set of activities with the primary goal of discovering faults and defects. Specifically, it has the following objectives:

- To design a test case with high probability of finding as-yet undiscovered bugs
- To execute the program with the intent of finding bugs

Software Testing Principles

1. All tests should be traceable to the requirements.
2. Test should be planned long before testing begins.
3. The Pareto Principle applies to software testing. The Pareto Principle states that 80% of all errors uncovered during testing will likely be traceable to 20% of all program modules or classes.
4. Testing should begin "in small" and progress toward testing "in the large".
5. Exhaustive testing is not possible but there are testing strategies that allow us to have good chance of constructing software that is less likely to fail.
6. To be most effective, an independent third party (normally, a software testing group) should conduct the test.

Software Testing Strategies integrate software test case design methods into a well-planned series of steps that result in the successful implementation of the software. It is a road map that helps the end-users, software developers and quality assurance group conduct software testing. It has goals of ensuring that the software constructed implements a specific function (**verification**), and that the software is constructed traceable to a customer requirements (**validation**).

Software testing is performed by a variety of people. Software developers are responsible for testing individual program units before they perform the integration of these program units. However, they may have vested interest of demonstrating that the code is error-free. The *Quality Assurance group* may be tasked to perform the test. Their main goal is to uncover as many errors as possible. They ask the software developers to correct any errors that they have discovered.

Software testing begins from a single component using white-box and black-box

techniques to derive test cases. For object-oriented software development, building block of software is the class. Once the class is tested, integration of the class through communication and collaboration with other tested classes is tested. There may be instances that a subset of a test will be performed again to ensure that corrects of the fault does not generate additional or new errors. Finally, the software is tested as a whole system.

In order to have an organized way of testing the software, a test specification needs to be developed, normally, by the leader of the quality assurance group. If there is no quality assurance group, the project leader should formulate one. A **Test Specification** is an overall strategy that defines the specific tests that should be conducted. It also includes the procedures on how to conduct the test. It can be used to track the progress of the development team, and define a set of project milestones.

6.2

6.3 Software Test Case Design Methods

There are two ways software is tested. One approach is to test the internal workings of the software. It checks if internal operations perform as specified, and all internal components have been adequately tested. The other approach is to test the software as a whole, i.e., know how the software works and tests if it conforms to the specified functionality in the requirements. The first approach is known as white-box testing while the second approach is known as black-box testing.

6.3.1 White-Box Testing Techniques

White-Box Testing is also known as **glass-box testing**. It is a test case design technique that uses the internal control structure of the software component as defined by their methods to derive test cases. Its goal is to ensure that internal operations perform according to specification of the software component. It ensures that no logical errors, incorrect assumptions and typographical errors have been missed. It produces test cases that:

- Ensures that all independent paths with the component have been tested at least once;
- Tests all logical decisions on their true or false sides;
- Tests all loops at their boundaries and within their operational bounds; and
- Tests internal data structures for their validity.
- Tests paths within the components that are considered “out of the mainstream”.

There are several techniques that can be employed in defining test cases using the white-box approach. They are discussed in the succeeding section.

Basic Path Testing

Basis Path Testing is a white-box testing technique that enables the test case designer to derive a logical complexity measure based on the procedural specification of a software component. This complexity measure is used as a guide for defining a *basis set* of execution paths to be tested.

Steps in Deriving Test Cases using Basis Path Testing

STEP 1. Use a procedural specification as input in deriving the basic set of execution path.

The procedural specification can be the design (pseudo-code or flowchart) or the source code itself. For classes, the operation or method procedural design will be used. As an example, assume that the following is pseudo-code of some method of some class.

```
while (condition1) do
    statement1;
    statement2;
    do case var1
        condition1:
            statement3
        condition2:
            statement4;
            statement5;
        condition3:
            if (condition2) then
                statement6;
            else
                statement7
                statement8
            endif
    endcase
    statement9;
endwhile
```

Text 14: Sample Design Code

STEP 2. Draw the flow graph of the procedural specification.

The **flow graph** depicts the logical control flow of the procedural specification. It uses nodes, edges and regions. The **nodes** represent one or more statements. It can be mapped to sequences or conditions. The **edges** are the links of the nodes. They represent flow of control similar to flow charts. Areas that are bounded by the edges and nodes are called **regions**. Figure 6.1 shows the flow graph of the sample procedural code. The graph does not normally shows the code. It is placed there for clarification purposes.

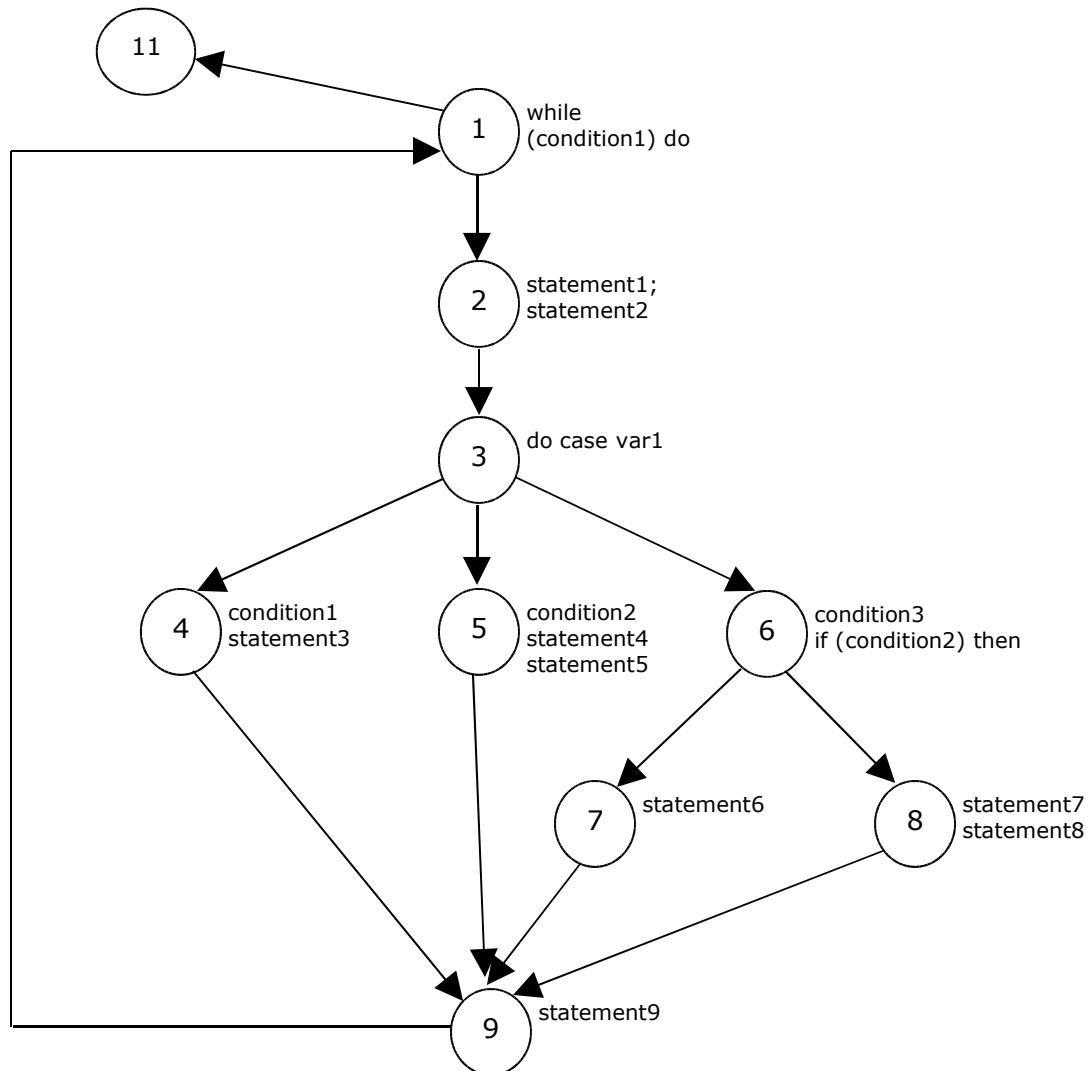


Figure 6.1 Flow Graph

STEP 3. Compute for the complexity of the code.

The **cyclomatic complexity** is used to determine the complexity of the procedural code. It is a number that specifies the independent paths in the code, which is the basis of the basic set. It can be computed in three ways.

1. The number of regions of the flow graph.
2. The number of predicate nodes plus one, i.e., $V(G) = P + 1$.
3. The number of edges minus the nodes plus 2, i.e., $V(G) = E - N + 2$

In the example:

The number of regions is 5.

The number of predicates is 4.

The number of edges is 13.

The number of nodes is 10.

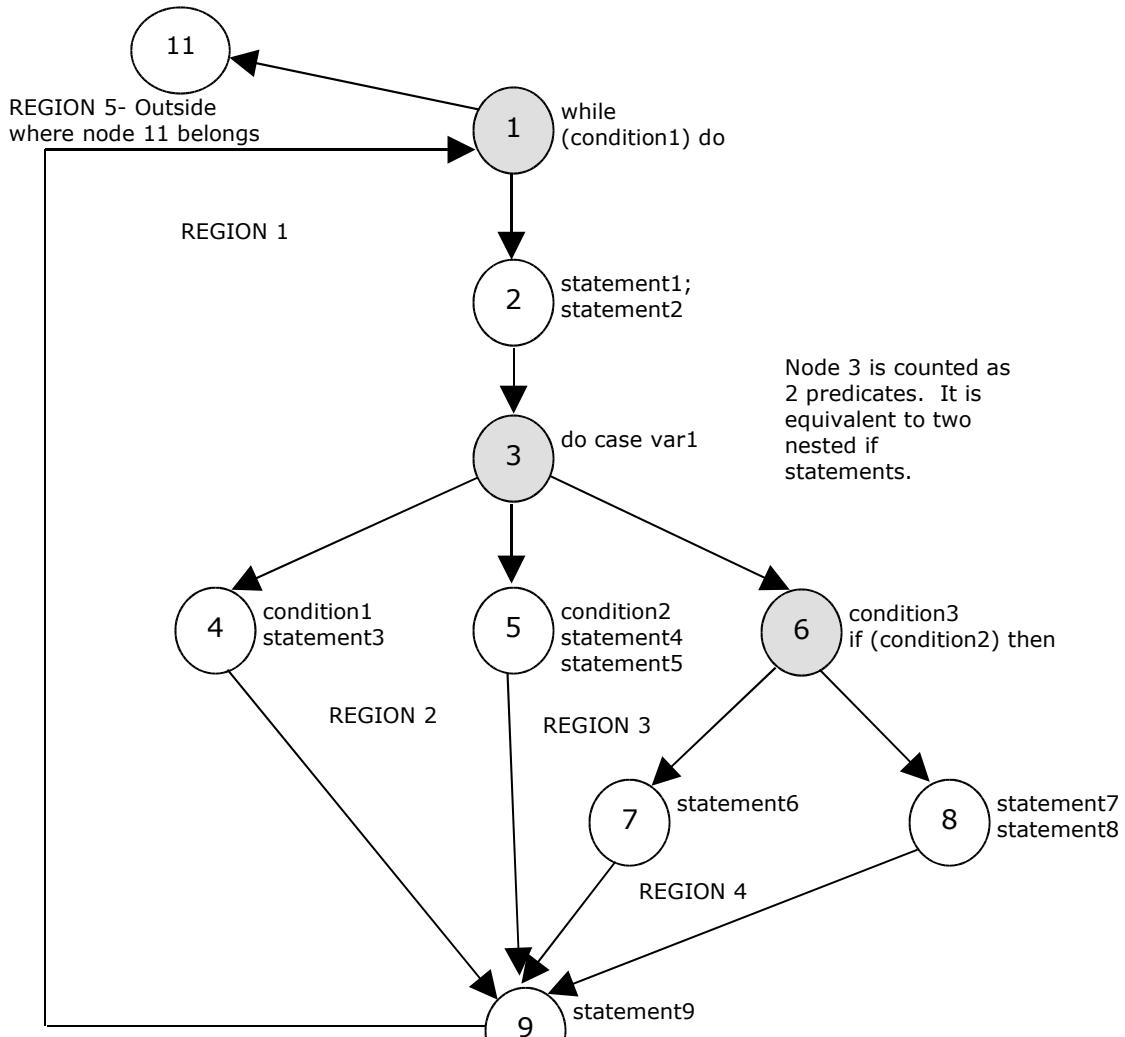


Figure 6.2 Cyclomatic Complexity Values

Using the formulae:

- $V(G) = 5$ (regions)
- $V(G) = 4$ (predicates) + 1 = 5
- $V(G) = 13$ (edges) - 10 (nodes) + 2 = 5

STEP 4. Determine the basic set of execution paths.

The cyclomatic complexity provides the number of the linearly independent paths through the code. In the above example, 5 linearly independent paths are identified.

- Path 1: Node-1, Node-2, Node-3, Node-4, Node-9
- Path 2: Node-1, Node-2, Node-3, Node-5, Node-9
- Path 3: Node-1, Node-2, Node-3, Node-6, Node-7, Node-9
- Path 4: Node-1, Node-2, Node-3, Node-6, Node-8, Node-9
- Path 5: Node-1, Node-11

STEP 5: Document the test cases based on the identified execution path.

Test cases are prepared to force execution of each path. Conditions on predicate nodes should be properly set. Each test case is executed and compared from the respected result. As an example:

PATH 1 Test Case:

For Node-1, condition2 should evaluate to TRUE (Identify the necessary values.)
For Node-3, evaluation of var1 should lead to Node-4 (Identify value of var1)
Expected Results: should produce necessary result for Node-9.

Control Structure Testing

Control Structure Testing is a white-box testing technique that test three types of program control, namely, condition testing, looping testing and data flow testing.

1. *Condition Testing.* It is a test case design method that test the logical conditions contained in a procedural specification. It focuses on testing each condition in the program by providing possible combination of values. As an example, consider the following condition of a program.

```
if ((result < 0) && (numberOfTest != 100))
```

The test cases that can be generated is shown in Table 26.

Test Case	result < 0	numberOfTest != 100	Compound
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Table 26: Control Structure Test Cases

2. *Looping Testing.* It is a test case design method that focuses exclusively on the validity of iterative constructs or repetition. There are four classes of iteration: simple, concatenated, nested and unstructured. They are shown in Table 27.

Simple Iteration:	Concatenated Iteration:
DO WHILE Statement ENDWHILE	DO WHILE Statement ENDWHILE DO WHILE Statement ENDWHILE
Nested Iteration:	Unstructured Iteration:
DO WHILE Statement DO WHILE Statement ENDWHILE ENDWHILE	DO WHILE Statement :Label1 DO WHILE Statement :Label2 IF <condition> GOTO Label1 ENDIF GOTO Label2 ENDWHILE ENDWHILE

Table 27: Iteration or Repetition Structure

For Simple Iteration, the test cases can be derived from the following possible execution of the iteration or repetition.

- Skip the loop entirely.
- Only one pass through the loop.
- Two passes through the loop.
- m passes through the loop where $m < n$
- $n - 1, n, n + 1$ passes through the loop

For Nested Iterations, the test cases can be derived from the following possible execution of the iteration or repetition.

- Start with the innermost loop.
- Conduct simple loop tests for the innermost loop while holding the outer loop at their minimum iteration parameter values. Add other test for out-of-range or excluded values.

- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
- Continue until all loops have been tested.

For Concatenated Iteration, the test cases can be derived from the following possible execution of the iteration or repetition

- If loops are independent, the test for simple loops can be used.
- If loops are dependent, the test for nested loops can be used.

For Unstructured Iteration, no test cases can be derived since it would be best to redesign the loop since it is not a good iteration or repetition constructs.

3. *Data Flow Testing.* It is a test case design method that selects test paths of a program according to the locations of the definitions and uses of variables in the program.

6.3.2 Black-Box Testing Techniques

Black-box testing is a test design technique that focuses on testing the functional aspect of the software whether it complies with functional requirements. Software engineers derive sets of input conditions that will fully test all functional requirements of the software. It defines a set of test cases that finds incorrect or missing functions, errors in interface, errors in data structure, errors in external database access, performance errors, and errors in initialization and termination.

Graph-based Testing

Graph-based Testing is a black-box testing technique that uses objects that are modeled in software and the relationships among these objects. Understanding the dynamics on how these objects communicate and collaborate with one another can derive test cases.

Developing Test Cases Using Graph-based Testing

STEP 1. Create a graph of software objects and identify the relationship of these objects.

Using nodes and edges, create a graph of software objects. Nodes represent the software objects. Properties can be used to describe the nodes and edges. For object-oriented software engineering, the collaboration diagram is a good input for the graph-based testing because you don’t need to create a graph. The collaboration diagram in Figure 6.3 is used as an example.

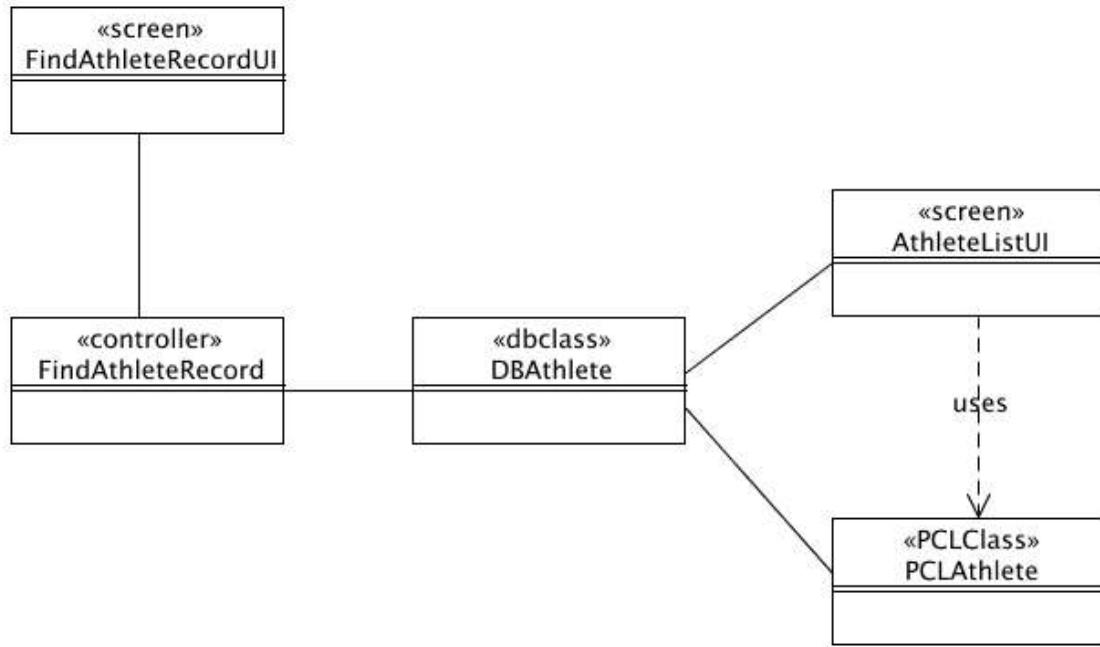


Figure 6.3: Finding an Athlete Collaboration Diagram

STEP 2. Traverse the graph to define test cases.

The derived test cases are:

Test Case 1:

- **FindAthleteUI** class sends a request to retrieve a list of athlete based on a search criteria. The request is sent to the **FindAthleteRecord**.
- The **FindAthleteRecord** sends a message to the **DBAthlete** to process the search criteria.
- The **DBAthlete** request the database server to execute the SELECT-statement. It populates the **PCLAthlete** class with the athlete information. It returns the reference of the **PCLAthlete** to the **AthleteListUI**.
- The **AthleteListUI** lists the names of the athletes.

Guidelines for Graph-based Testing

1. Identify the start and stop points of the graph. There should be an entry and exit nodes.
2. Name nodes and specify their properties.
3. Establish their relationship through the use of edges. Specify the properties.
4. Derive test cases and ensure that there is node and edge coverage.

Equivalence Testing

Equivalence Testing is a black-box testing technique that uses the input domain of the program. It divides the input domain into sets of data from which test cases can be derived. Derived test cases are used to uncover errors that reflect a class of errors. Thus, reduces the effort in testing the software. It makes use of **equivalence classes**, which are sets of valid and invalid states that an input may be in.

Guidelines in Identifying Equivalence Classes

1. Input Condition is specified as a range of value. The test case is one valid input, and two invalid equivalence classes.
2. Input Condition requires a specific value. The test case is one valid, and two invalid equivalence classes.
3. Input condition specifies a member of a set. The test case is one valid and one invalid.
4. Input condition is Boolean. The test case is one valid, and one invalid.

As an example, consider a text message code of registering a mobile number to a text service of getting traffic reports. Assume that the message requires the following structure:

Service Server Number (Number of the Server providing the service.)	Service Code (A unique service code that tells the service provider what service is asked.)	Mobile Number (The mobile number where the information is sent.)
765	234	09198764531

Input Conditions associated with each data element:

Service Server Number (specific value)	input condition 1: correct value input condition 2: incorrect value
Service Code (specific value)	input condition 1: correct value input condition 2: incorrect value
Mobile Number (specific value)	input condition 1: correct value input condition 2: missing number input condition 3: length not correct

Boundary Value Testing

Boundary Value Testing is a black-box testing technique that uses the boundaries of the input domain to derive test cases. Most errors occur at the boundary of the valid input values.

Guidelines in Deriving Test Cases Using Boundary Value Testing

1. If input condition specifies range bounded by n and m , the test cases that can be derived:
 - use values n and m
 - just above n and m
 - just below n and m
2. If input condition specifies number of values, the test cases that can be derived:
 - use the minimum
 - use the maximum
 - just above and below minimum
 - just above and below maximum.

6.4 Testing your Programs

Software testing can be done in several tasks or phases. Program testing is concerned with testing individual programs (unit testing) and their relationship with one another (integration testing). This section discusses concepts and methods that allows one to test programs.

Unit Testing

Unit testing is the basic level of testing. It has an intention to test the smaller building blocks of a program. It is the process of executing each module to confirm that each performs its assigned function. It involves testing the interface, local data structures, boundary conditions, independent paths and error handling paths.

The environment to which unit tests can be performed is shown in Figure 6.4. To test the module, a driver and stub is used. A **Driver** is a program that accepts test case data, passes data to the component to be tested, and prints relevant results. A **Stub** is a program that performs support activities such as data manipulation, queries of states of the component being tested and prints verification of entry. If the driver and stub require a lot of effort to develop, unit testing may be delayed until integration testing.

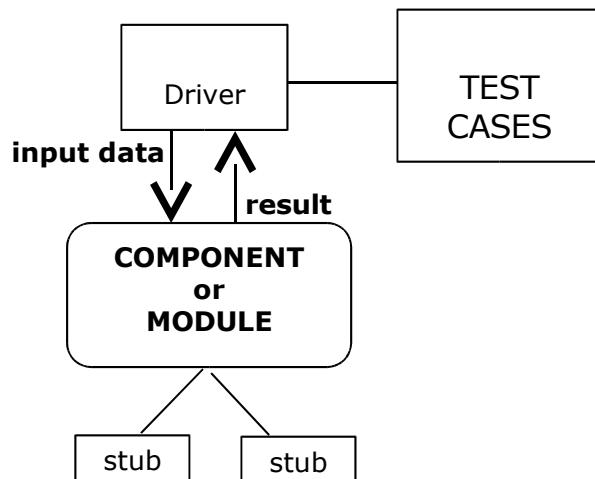


Figure 6.4 Unit Test Environment

To create effective unit tests, one needs to understand the behavior of the unit of software that one is testing. This is usually done by decomposing the software requirements into simple testable behaviors. It is important that software requirements can be translated into tests.

In object-oriented software engineering, the concept of encapsulation is used to define a class, i.e., the data (attributes) and functions (methods or operation) are grouped together in a class. The smallest testable units are the operations defined in the class. As opposed to conventional way of testing, operations defined in class cannot be tested separately. It should be tested in the context to which objects of the class are instantiated. As an example, consider the class diagram shown in Figure 6.5.

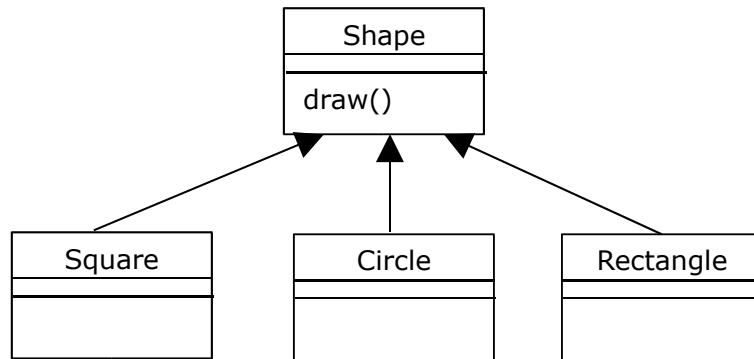


Figure 6.5 Sample Class Hierarchy

The `draw()` method is defined in **Shape** class. All subclasses of this base class inherit this operation. When a subclass uses the `draw()` method, it is being executed together with the private attributes and other methods within the context of the subclass. The context in which the method is used varies depending on what subclass executes it. Therefore, it is necessary to test `draw()` method in all subclasses that uses it. As the number of subclasses is defined for a base class, the more testing is required for the base class.

Integration Testing

After unit testing, all classes must be tested for integration. **Integration testing** verifies that each component performs correctly within collaboration and that each interface is correct. Object-oriented software does not have an obvious hierarchy of control structure which is characteristic of conventional way of developing software. Traditional way of integration testing (top-down and bottom-up strategies) has little meaning in such software. However, there are two approaches that can be employed in performing integration testing.

Thread-based Testing Approach

Integration Testing is based on a group of classes that collaborate or interact when one input needs to be processed or one event has been triggered. A **thread** is a path of communication among classes that needs to process a single input or respond to an event. All possible threads should be tested. The sequence diagrams and collaboration diagrams can be used as the basis for this test. As an example, consider the sequence diagram that retrieves an athlete record as shown in Figure 6.6. In integration testing, just follow the order of the interaction of the objects through their messages.

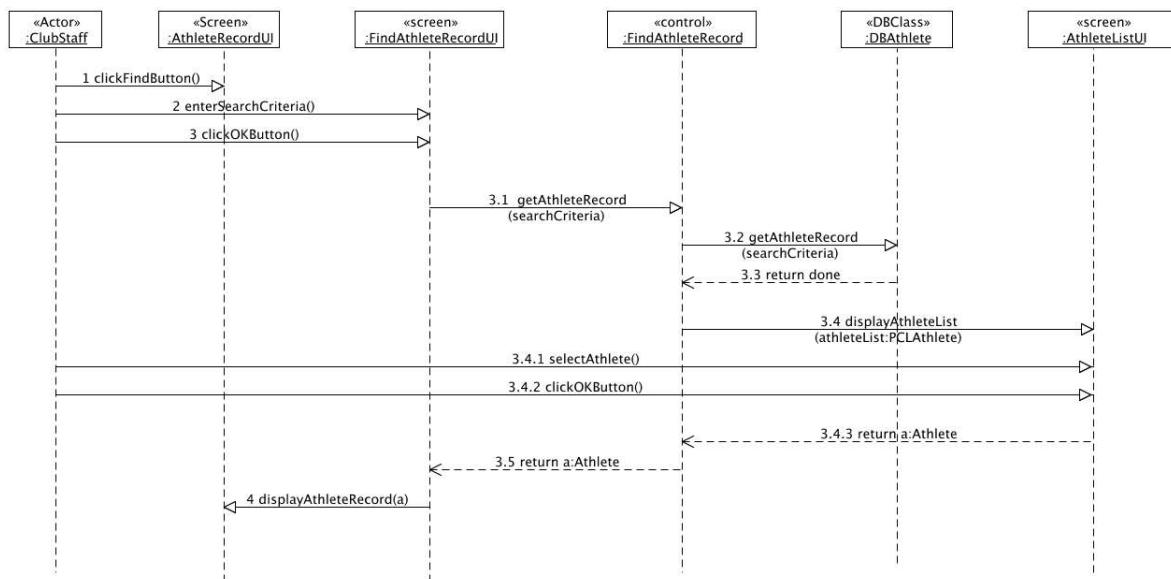


Figure 6.6 Retrieving an Athlete Record

The sequence of the test would involve the following steps:

1. The club staff will click the **Find Button** of the **Athlete Record User Interface**.
2. The **FindAthleteRecordUI** Screen will be displayed.
3. The club staff enters the search criteria.
4. The club staff clicks the **OK button** which will instantiate the **FindAthleteRecord** Controller.
5. The **FindAthleteRecord** Controller communicates with the **DBAthlete** to populate the **PCLAthleteList**.
6. The **AthleteListUI** Screen will be displayed to show a list of athletes that satisfy the search criteria.
7. The club staff highlights the athlete he wants to retrieve.
8. The club staff clicks the **OK button** or press enters.
9. The **AthleteRecordUI** Screen should display the athlete information on the appropriate text field.

Use-based Testing Approach

Integration Testing starts by identifying independent classes. **Independent classes** are those classes that do not use or use few server classes. Independent classes are tested first. After testing these classes, the next set of classes called **dependent classes** are tested. Dependent classes are those classes that use the independent classes. As an example, consider the classes shown in Figure 6.2. The **Athlete** and **PCLAthlete** classes can be considered as independent classes. Their methods can be tested independently. If tests are successful, the **DBAthlete** class can be tested.

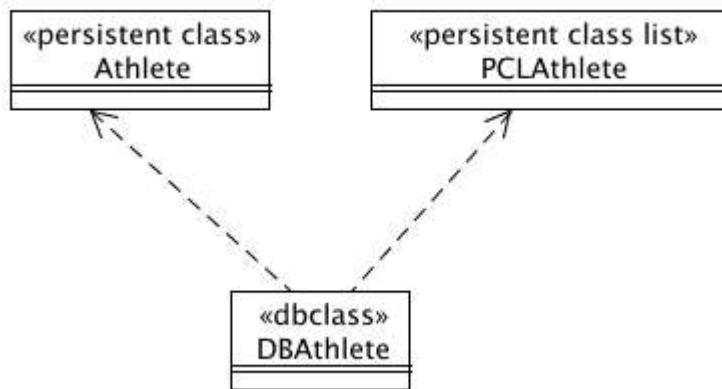


Figure 6.7 DBAthlete Cluster

To help manage integration, clustering can be used. **Clustering** is the process of defining a set of classes that are integrated and tested together. They are called clusters because they are considered as one unit. In the examples, these three classes combined together can be known as the **DBAthlete** cluster.

Regression Testing

Sometimes when errors are corrected, it is necessary to re-test the software components. **Regression testing** is re-execution of some subset of tests to ensure that changes have not produced unintended side effects. Re-testing occurs for the software function that was affected, software component that has been changed and software components are likely to be affected by the change.

6.5 Test-driven Development Methodology

Test-driven Development (TDD) is a method of developing software that adopts a test-first approach plus refactoring. The **test-first approach** is a programming technique, involving analysis, design, coding and testing all together. Two basic steps are involved in this approach:

- Write just enough test to describe the next increment of behavior.
- Write just enough production code to pass the test.

As the name implies, one writes the test first.

Refactoring, on the other hand, means improving the design of the existing code. It is a systematic way of improving the design of the code by removing redundant or duplicate codes. It is a process of changing code so that the internal structure of the code is improved without changing the behavior of the program. It is cleaning up bad code. Most developers refactor code on a regular basis using common sense.

In test-driven development, each new test provides feedback into the design process which helps in driving design decisions. Often, one doesn't realize a design has problems until it has been used. Test allows us to uncover design problems.

Benefits of Test-driven Development

There are many benefits. Some of them are enumerated below.

1. *It allows simple incremental development.* One of the immediate benefits of TDD is that one has a working software system almost immediately. The first iteration of the software is very simple and may not have much functionality. But, the functionality will improve as the development continues. This is less risky compared from building the whole system and hope that the pieces will work together.
2. *It involves a simpler development process.* Developers are more focused because the only thing that they need to worry about is passing the next test. He focuses his attention on a small piece of software, gets it to work, and moves on.
3. *It provides constant regression testing.* The domino effect is very prevalent in software development. A change to one module may have unforeseen consequences throughout the rest of the project. This is why regression testing is needed. TDD runs the full set of unit tests every time a change is made. This means that any change to the code that has an undesired side-effect will be detected immediately and be corrected. The other benefit of constant regression testing is that you always have a fully working system at every iteration of the development. This allows you to stop the development at any time and quickly respond to any changes in the requirements.
4. *It improves communication.* The unit tests serve as a common language that can be used to communicate the exact behavior of a software component without ambiguities.
5. *It improves understanding of required software behavior.* Writing unit tests before writing the code helps the developer focus on understanding the required behavior of the software. As a developer is writing unit test he is

adding pass/fail criteria for the behavior of the software. Each of these pass/fail criteria adds to the knowledge of how the software must behave. As more unit tests are added because of new features or correction, the set of unit tests becomes a representative of a set of required behaviors of the software.

6. *It centralizes knowledge.* The unit tests serve as a repository that provides information about the design decisions that went into the design of the module. The unit tests provide a list of requirements while the source code provides the implementation of the requirements. Using these two sources of information makes it a lot easier for developers to understand the module and make changes that won't introduce bugs.
7. *It improves software design.* Requiring the developer to write the unit tests for the module before the code helps them define the behavior of the software better and helps them to think about the software from a users point of view.

6.5.1 Test-driven Development Steps

Test-driven development basically is composed of the following steps:

1. *Write a test that defines how the software should behave.* As an example, suppose that "Ang Bulilit Liga" has the following requirements, "The software must be able to compute the shooting average of every athlete by game and for the season." There are a number of behaviors the software needs to have in order to implement this requirement such as:
 - The software should identify an athlete.
 - There must be a way to input the data needed to calculate the shooting average.
 - There must be a way to identify the game.
 - There must be a way to get a athlete's shooting average for a game.
 - There must be a way to get a athlete's shooting average for the entire season.

After making some assumptions like athletes will be identified by their ids and games by dates, you could write the test case. As an example, a fraction of the unit test code is shown in Text 15.

```
Athlete timmy = PCLAthleteList.getAthlete(3355);
AthleteStatistics timmyStatistics =
    new AthleteStatistics(timmy);

//Add shooting average statistics shooting 1 for 9 attempts
// last 8/9/2004 game.
timmyStatistics.addShootingAverage("8/9/2004",1,9);

//Add shooting average statistics shooting 4 for 10 attempts
// last 8/16/2004 game.
timmyStatistics.addShootingAverage("8/16/2004",4,10);

//Add shooting average statistics shooting 7 for 10 attempts
// last 8/25/2004 game.
timmyStatistics.addShootingAverage("8/25/2004",7,10);

//Compute for the shooting average for game 8/16
float gameShootingAverage =
    timmyStatistics.getGameShootingAverage("8/16/2005");

//Compute for the shooting average for season
float seasonShootingAverage =
    timmyStatistics.getSeasonShootingAverage();

//Check game shooting average
assertEquals(200, gameShootingAverage);

//Check game shooting average
assertEquals(214, seasonShootingAverage);
```

Text 15 Sample Unit Test for Computing Shooting Average

2. *Make the test run as easily and quickly as possible.* You don't need to be concerned about the design of the source code; just get it to work. In the case of the example, the source code we are referring to is the class **AthleteStatistics** which contains the methods that compute the shooting average of an athlete by game and season. Once the unit test is written, it is executed to verify for execution failure. The code fails because the code to implement the behavior has not yet been written but this is an important step because this will verify that unit test is working correctly. Once the unit test has been verified, the code to implement the behavior is written and the unit test is run against it to make sure that the software works as expected. An initial **AthleteStatistics** class is shown below.

```
public class AthleteStatistics{
    private Athlete a;
    private Statistics stats;

    //constructor with Athlete parameter
    public AthleteStatistics(Athlete theAthlete){
        //place here code of initializing statistics of an athlete.
        a = theAthlete;
        stats = new Statistics();
    }

    public void addShootingAverage(String dateOfGame,
        int shots, int attempts){
        //place here code of adding statistics for the game.
    }

    public float getGameShootingAverage(String dateOfGame){
        //place computation here.
    }

    public float getSeasonShootingAverage(){
        //place computation here.
    }
}
```

Text 16 Fraction of AthleteStatistics Codes

3. *Clean up the code.* Now that the code is working correctly, take a step back and refactor the codes to remove any duplication or any other problem that was introduced to get the test running. As developers, you know that the first pass at writing a piece of code is usually not the best implementation. Do not be afraid to refactor your code. If you break your code, the unit test will let you know immediately.

6.5.2 Testing Java Classes with JUnit

JUnit is a program used to perform unit testing of softwares. It is used for writing and organizing test cases for Java programs. Test cases could be written without the help of JUnit, but JUnit tests, aside from being easier to code, gives the programmer flexibility in grouping small tests allowing simultaneous running of tests and easier program debugging. This section discusses how JUnit is used within the Java Studio™ Enterprise 8.

Unit Testing with JUnit

Unit testing with JUnit can be accomplished by doing the following:

1. Build a subclass of the class **TestCase**.
2. Create a method named **setUp()** if some things have to be set before testing.
3. Write as many test methods as you deem necessary.
4. Create a method named **tearDown()** if some things have to be cleared after testing.

To run several tests at once, just create a test suite. JUnit provides an object,

TestSuite, which runs multiple test cases and test suites together.

Using JUnit in Sun Java™ Studio Enterprise 8

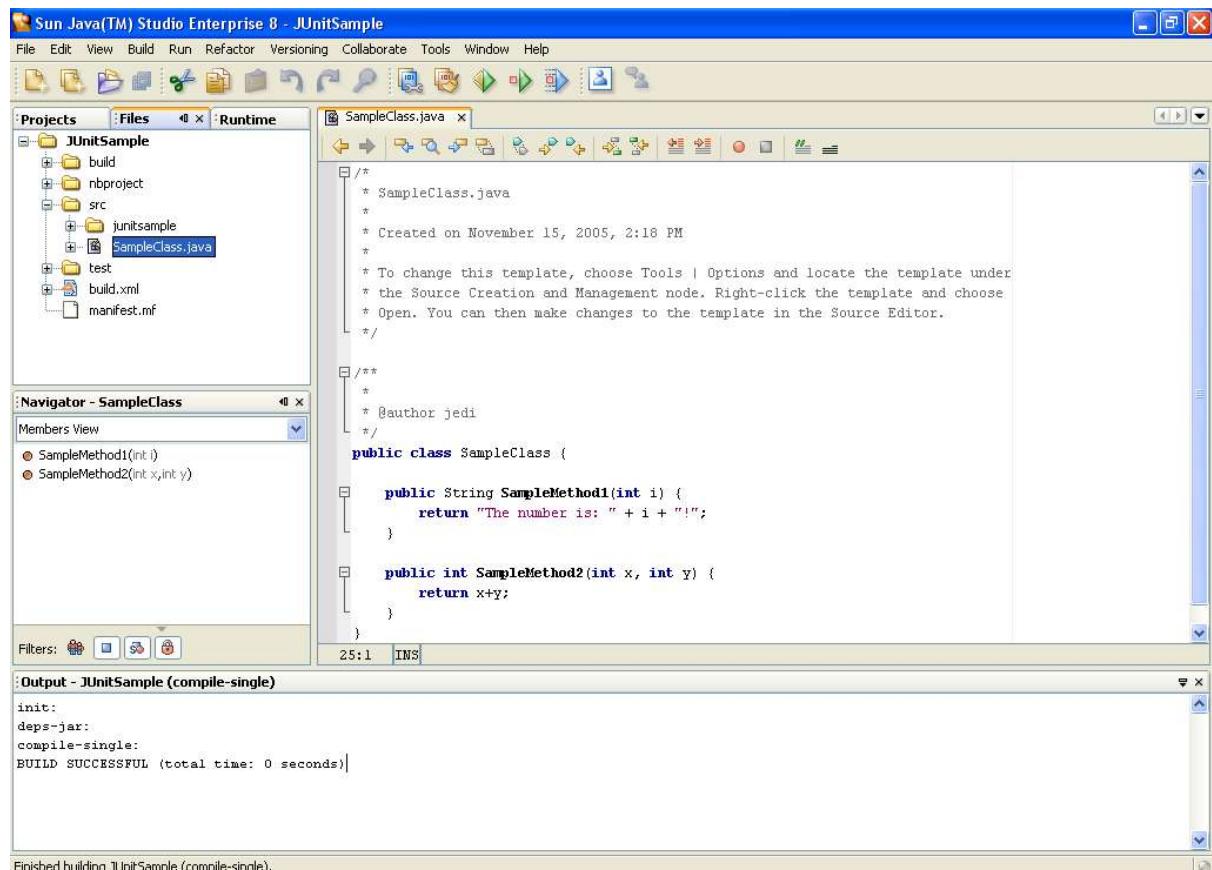


Figure 6.8 Sample Code to be Tested

Creating tests in Sun Java™ Studio Enterprise 8 is very easy. Using the sample shown in Figure 6.8 as the class to be tested, create a test by selecting **Tools** → **JUnit Tests** → **Create Tests** from the toolbar. Make sure that the class that you want to test is highlighted in the file's node. Alternately, tests for a class can also be created by right-clicking the file's node and choosing **Tools** → **JUnit Tests** → **Create Tests** from the contextual menu.

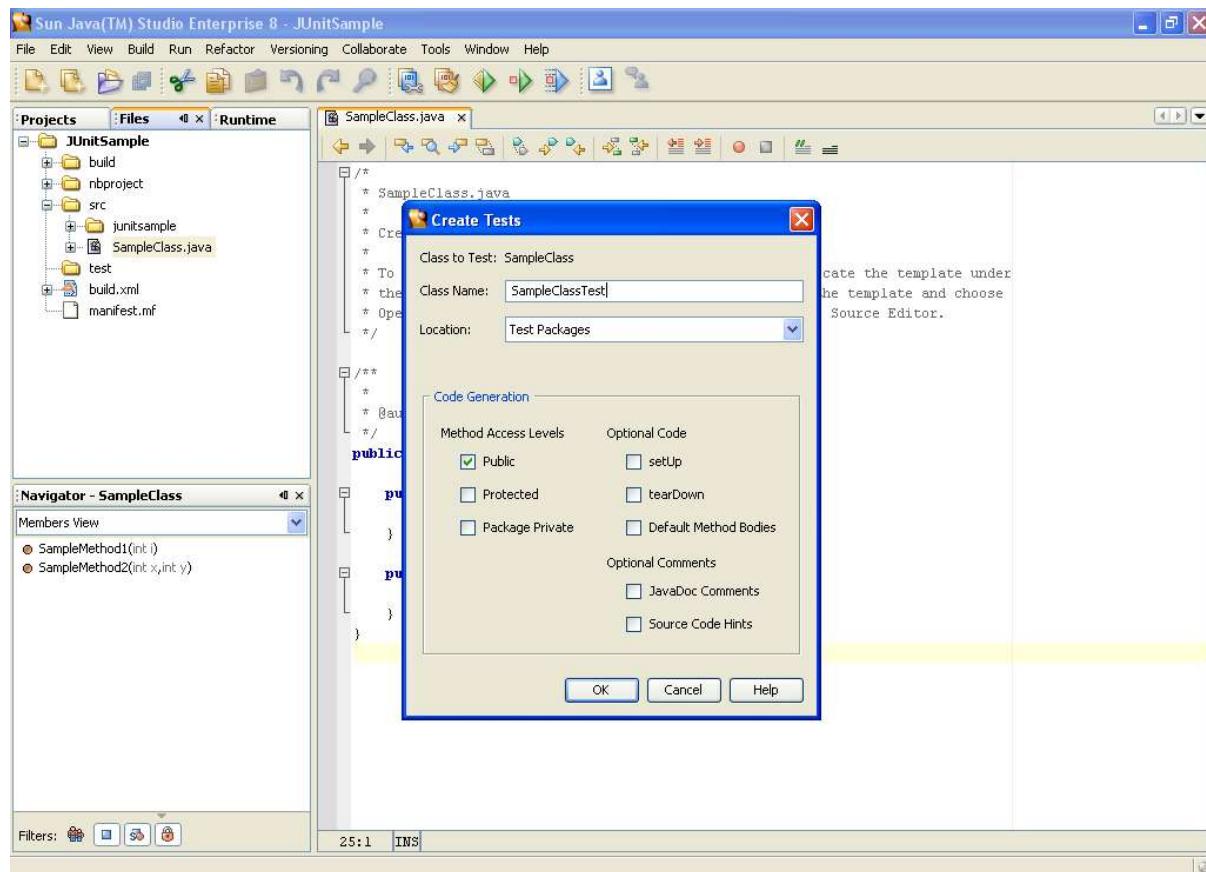


Figure 6.9 JUnit Create Test Dialog Box

In the **Create Tests** dialog box, select the **Code Generation** options the user requires for the tests. The IDE will generate a template with the specified options. For this case, everything is unchecked except the **Public** option. The class name for the test can also be renamed from the default in this dialog box (Figure 6.9). JUnit settings can be changed in **JUnit Module Settings** (in the **Options** window) or the **Create Tests** dialog box. Clicking **OK** will generate a skeleton test case from the sample code's (to be tested) structure (Figure 6.10).

Methods can be written using test expressions from the Assert class (Class `junit.framework.Assert`). Figure 6.11 shows the test methods written to test the sample code in Figure 1. To run the test, choose **Run → Test "project-name"** from the toolbar. The test output will then be generated by the IDE.

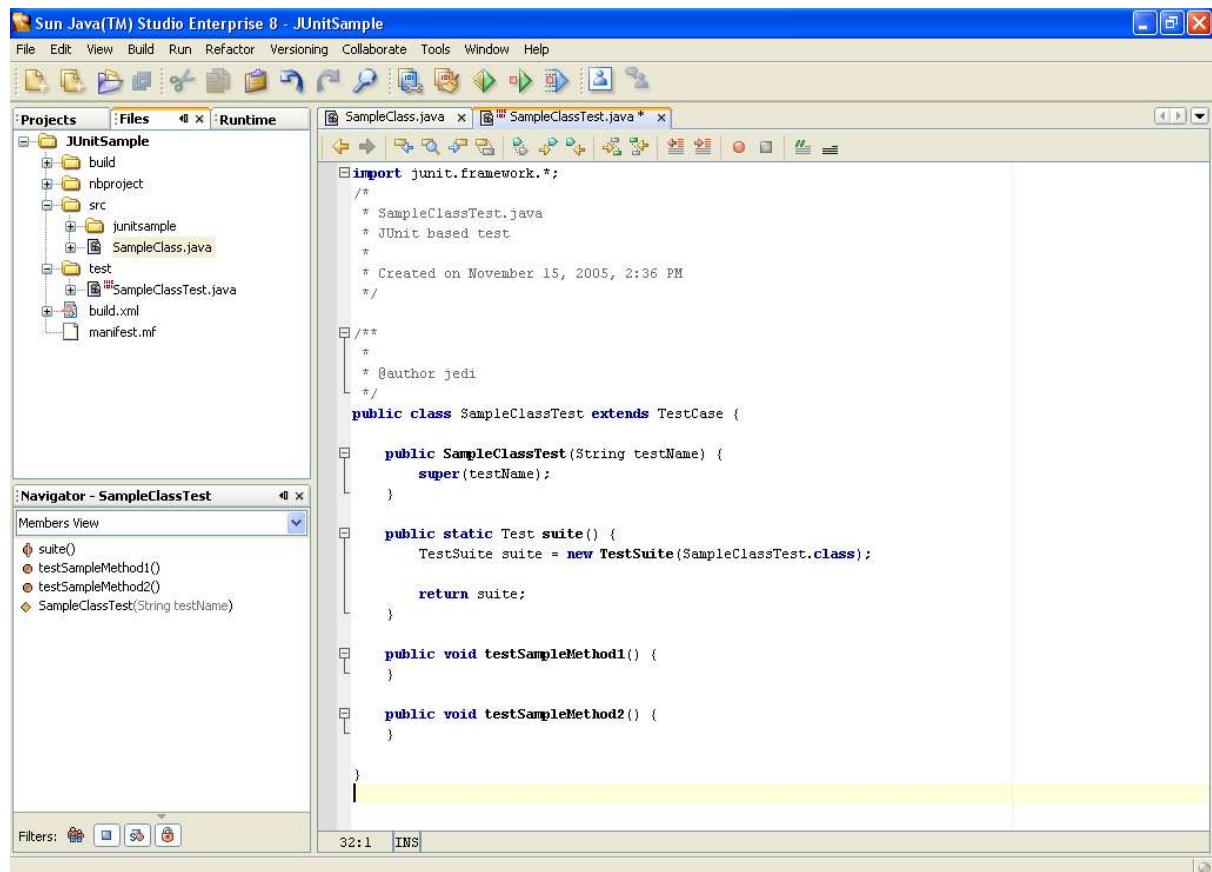


Figure 6.10 IDE Generated Test

Failed test cases are specified in the output box followed by the reason for their failure. For the sample code's case, `testSampleMethod1` failed because the expected string was different from the obtained string. The second test case `testSampleMethod2` did not fail, therefore, no error messages were displayed.

The test suite can also be edited to add additional test cases outside the file. Other test suites can also be added in the test suite. Other methods like `setUp()` and `tearDown()` can also be included in the test case. These methods are often called test fixtures since they are fixed or set before or after the tests.

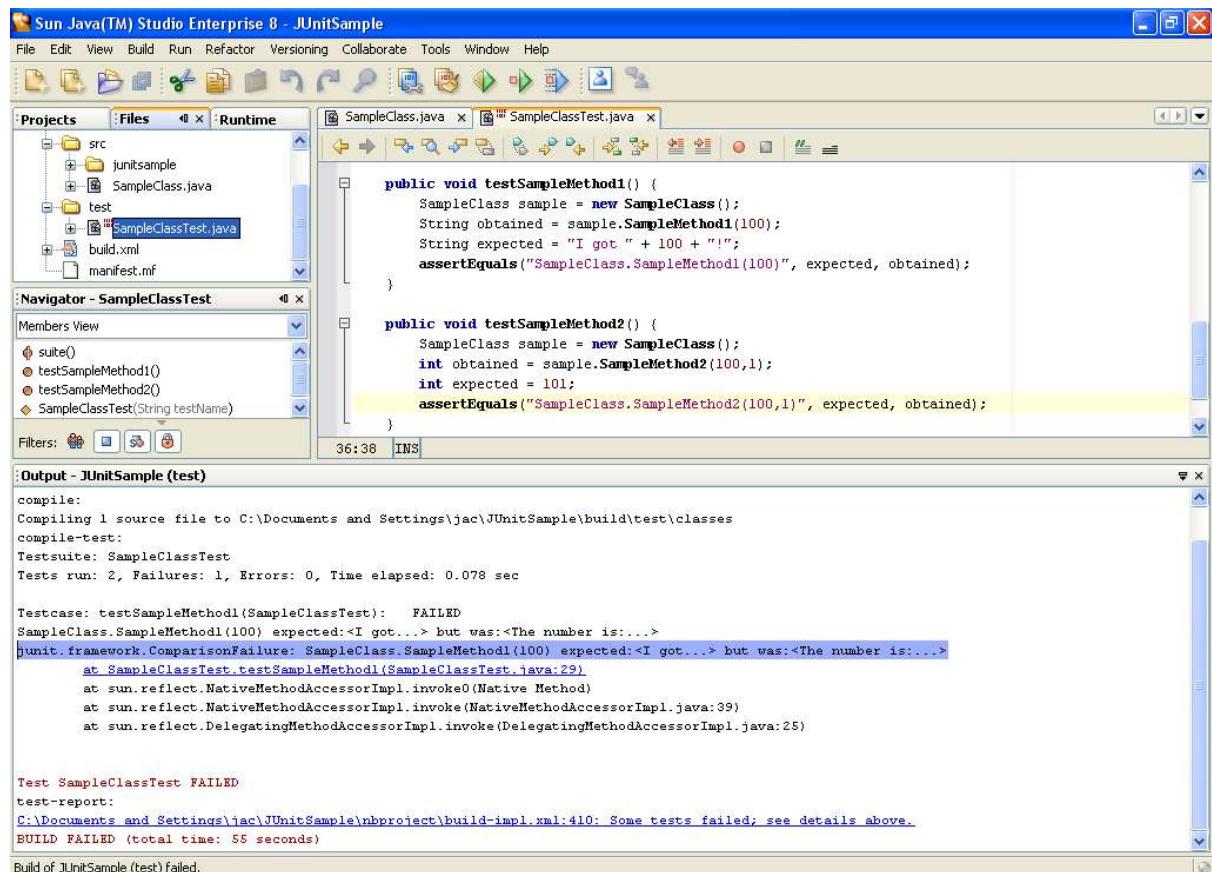


Figure 6.11 JUnit Test Case Using assertEquals()

Samples of the program is shown in Text 16 and Text 14.

```
/*
 * SampleClass.java
 *
 * Created on November 17, 2005, 4:03 PM
 *
 * To change this template, choose Tools | Options and locate
the template under
 * the Source Creation and Management node. Right-click the
template and choose
 * Open. You can then make changes to the template in the
Source Editor.
 */

/**
 *
 * @author Jaclyn Cua
 */
public class SampleClass {

    public String SampleMethod1(int i) {
        return "The number is: " + i + "!";
    }

    public int SampleMethod2(int x, int y) {
        return x+y;
    }
}
```

Text 17 SampleClass.java

```
import junit.framework.*;
/*
 * SampleClassTest.java
 * JUnit based test
 *
 * Created on November 17, 2005, 4:06 PM
 */

/**
 *
 * @author Jaclyn Cua
 */
public class SampleClassTest extends TestCase {

    public SampleClassTest(String testName) {
        super(testName);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(SampleClassTest.class);

        return suite;
    }

    public void testSampleMethod1() {
        SampleClass sample = new SampleClass();
        String obtained = sample.SampleMethod1(100);
        String expected = "I got " + 100 + "!";
        assertEquals("SampleClass.SampleMethod1(100)", expected, obtained);
    }

    public void testSampleMethod2() {
        SampleClass sample = new SampleClass();
        int obtained = sample.SampleMethod2(100,1);
        int expected = 101;
        assertEquals("SampleClass.SampleMethod2(100,1)", expected, obtained);
    }
}
```

Text 18 SampleClassTest.java

6.6 Testing the System

System testing is concerned with testing an entire system based on its specification. The software under test is compared with its intended functional specification. After integration testing, the system is tested as a whole for functionality and fitness of use. This may include any tests such as performance testing which test the response time and resource utilization, stress testing which test the software under abnormal quantity, frequency and volume of data, requests or processes, security testing which test the protective mechanism of the software, recovery testing which test the recovery mechanism of the software when it fails etc.

In the context of object-oriented software engineering, test requirements are derived from the analysis artifacts or work products that use UML such as the class diagrams, sequence diagrams and collaboration diagrams. To derive system test cases, the use case diagram is a good source. They represent a high level of functionalities provided by the system to the user. Conceptually, functionality can be viewed as a set of processes that run horizontally through the system, and the objects as sub-system components that stand vertically. Each functionality does not use every object, however, each object may be used by many functional requirements. This transition from a functional to an object point-of-view is accomplished with use cases and scenarios.

However, use cases are not independent. They not only have the extend and include dependencies, they also have sequential dependencies which came from the logic of the business process the system supports. When planning test cases, we need to identify possible execution sequences as they may trigger different failures.

A scenario is an instance of a use case, or a complete path through the use case. End users of the completed system can go down many paths as they execute the functionality specified in the use case. Each use case would have a basic flow and alternative flow. The basic flow covers the normal flow of the use case. The alternative flows cover the behavior of an optional or exceptional character relative to normal behavior.

The succeeding section gives the steps in deriving the test cases using use cases. It is adapted from Heumann⁹.

Generating System Test Cases

STEP 1: Using the RTM, prioritize use cases.

Many software project development may be constrained to deliver the software soon with a relatively small team of developers. In this case, prioritizing which system function should be developed and tested first is very crucial. Importance is gauged based on the frequency with which each function of the system is used. The RTM is a good tool in determining which system functions are developed and tested first.

⁹ Heuman, J. *Generating Test Cases for Use Cases..* The Rational Edge, Rational Rose Software, 2001.

RTM ID	Requirement	Notes	Requestor	Date	Priority
Use Case 1.0	The system should be able to maintain an athlete's personal information.		RJCS	06/12/05	Should Have
Use Case 2.0	The system should be able to allow the coach to change the status of an athlete.		RJCS	06/12/05	Should Have
Use Case 3.0	The system should be able to view the athlete's personal information.		RJCS	06/12/05	Should Have
Use Case 1.1	The system should be able to add an athlete's record.		RJCS	06/13/05	Should Have
Use Case 1.2	The system should be able to edit an athlete's record.		RJCS	06/13/05	Should Have
Use Case 1.3	The system should be able to remove an athlete's record		RJCS	06/13/05	Should Have

Table 28: Sample Initial RTM for Club Membership Maintenance

Consider the RTM formulated during the analysis as shown in Table 28. Users may decide to do Use Case 1.0 and its sub-use cases first rather than the rest.

STEP 2: For each use case, create use case scenarios.

For each use case starting from higher to lower priority, create a sufficient set of scenarios. Having a detailed description of each scenario such as the inputs, preconditions and postconditions can be very helpful in the later stages of test generation. The list of scenario should include the basic flow and at least one alternative flow. The more alternative flows mean more comprehensive modeling and consequently more thorough testing. As an example, consider the list of scenarios for adding an athlete to the system in Table 29.

Scenario ID	Scenario Name
SCN-01	Adding an Athlete Data Successfully
SCN-02	Athlete Data is Incomplete
SCN-03	Athlete Data is Invalid
SCN-04	Duplicate Entry
SCN-05	System Unavailable
SCN-06	System Crashes during Input

Table 29: Scenario Set for Adding an Athlete Record

STEP 3: For each scenario, take at least one test case and identify the conditions that will make it execute.

The preconditions and flow of events of each scenario can be examined to identify the input variables and the constraints that bring the system to a specific state as represented by the post conditions. A matrix format is suitable for clearly documenting the test cases for each scenario. As an example, consider the matrix as shown in Table 30.

Test Case ID	Scenario	Athlete Data	Not Already in the Athlete Database	Expected Result
TC-01	Adding an Athlete Data Successfully	V	V	Display athlete added. message record
TC-02	Athlete Data is Incomplete	I	V	Error message: Incomplete Athlete Data.
TC-03	Athlete Data is Invalid	I	V	Error message: Invalid data
TC-04	Duplicate Entry	V	I	Error message: Athlete record already exists.
TC-05	System Unavailable	V	I	Error Message: System not available.
TC-06	System Crashes during Input	V	I	System crashes but should be able to recover.

Table 30: Sample Test Case Matrix for Adding An Athlete Record

The test matrix represents a framework of testing without involving any specific data

values. The V indicates valid, I indicates invalid and N/A indicates not applicable. This matrix is an intermediate step and provides a good way to document the conditions that are being tested.

STEP 4: For each test case, determine the data values.

Once all test have been identified they should be completed, reviewed, and validated to ensure accuracy and identify redundant or missing test cases. If this is ensured then the next step is to plug in the data values for each V's and I's. Before deploying the application, it should be tested using real data to see if some other issues like performance pop up. Table 31 shows sample values.

Test Case ID	Scenario	Athlete Data	Not Already in the Athlete Database	Expected Result
TC-01	Adding an Athlete Data Successfully	Johnny De la Cruz Lot 24 block 18 St. Andrewsfield, Quezon City 24/9/1998 Male Status Guardian: Johnny De la Cruz, Sr. -same address- 555-9895	V	Display message athlete record added.
TC-02	Athlete Data is Incomplete	No guardian	V	Error message: Incomplete Athlete Data.
TC-03	Athlete Data is Invalid	Missing Bday	V	Error message: Invalid data
TC-04	Duplicate Entry	Johnny De la Cruz Lot 24 block 18 St. Andrewsfield, Quezon City 24/9/1998 Male Status Guardian: Johnny De la Cruz, Sr. -same address- 555-9895	I	Error message: Athlete record already exists.

Table 31: Sample Test Case Matrix for Adding An Athlete Record with Values

Validation testing starts after the culmination of system testing. It consists of a series of black-box tests cases that demonstrate conformity with the requirements. The software is completely packaged as a system and that interface errors among software components have been uncovered and corrected. It focuses on user-visible actions and user-recognizable output.

A **Validation Criteria** is a document containing all user-visible attributes of the software. It is the basis for validation testing. If the software exhibits these visible attributes, then, the software complies with the requirements.

Alpha & Beta Testing

Alpha & Beta Testing is a series of acceptance tests to enable customer or end-user to validate all requirements. It can range from informal test drive to planned and systematically executed tests. This test allows end-users to uncover errors and defects that only them can find. **Alpha testing** is conducted at a controlled environment, normally, at the developer's site. End-users are asked to use the system as if it is being used naturally while the developers are recording errors and usage problems. **Beta testing**, on the other hand, is conducted at one or more customer sites and developers are not present. At this time, end-users are the ones recording all errors and usage problems. They report them to the developer for fixes.

6.7 Mapping the Software Testing Deliverable to the RTM

Once a component is implemented, the test specification and test cases must be developed. We need to keep track of it, and at the same time, assure that each test is traced to a requirement. Additional RTM components are added. The following list the recommended elements for software testing in the RTM.

RTM Software Testing Components	Description
Test Specification	The filename of the document that contains the plan on how to test the software component.
Test Cases	The filename that contains the test cases to be performed as part of the test specification.

Table 32: Software Testing RTM Elements

These components should be related to a software component defined in the RTM.

6.8 Test Metrics

The metrics used for object-oriented design quality mentioned in Design Engineering can also provide an indication of the amount of testing effort is need to test object-oriented software. Additionally, other metrics can be considered for encapsulation and inheritance. Example of such metrics are listed below.

1. *Lack of Cohesion in Methods (LCOM)*. If the value of LCOM is high, the more states of the class needs to be tested.
2. *Percent Public and Protected (PAP)*. This is a measure of the percentage of class attributes that are public or protected. If the value of PAP is high, it increases the likelihood of side effects among classes because it leads to high coupling.
3. *Public Access To Data Members (PAD)*. This is a measure of the number of classes or methods that an access another class's attributes. If the value of PAD is high, it could lead to many side effects.
4. *Number of Root Classes (NOR)*. As the number of root classes increases, testing effort also increases.
5. *Number of Children (NOC) and Depth of the Inheritance Tree (DIT)*. Superclass methods should be re-tested for every subclass.

6.9 Exercises

6.9.1 Specifying a Test Case

1. Using the Basic Path Testing, write the test cases that must be done for the method retrieving an athlete record. Determine the cyclomatic complexity of the code to determine the number of independent paths. Each of this path becomes one test case.

6.9.2 Specifying System Test Cases

1. Write the system test cases for the Coach Information Maintenance System.
2. Write the system test cases for the Squad and Team Maintenance System.

6.10 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in this chapter. Particularly, they are:

1. Defining the test cases.
2. Defining the system test cases.
3. Tracking the revision and testing using the Requirements Traceability Matrix.

MAJOR WORK PRODUCTS:

1. Test Cases
2. System Test Cases
3. Test Results

7 Introduction to Software Project Management

Building a computer software include a variety of phases, activities and tasks. Some of these activities are done iteratively in order for the computer software be completed and delivered to the end-users for their operation. However, before the development can proceed, end-users and developers need estimates on how long the project will proceed and how much it will cost. It is not the intention of this chapter to give a full-blown discussion of software project management. That is reserved for a Software Project Management Course. The discussion in this chapter would revolve around project management concepts that will help the group manage class-based software development projects.

7.1 Software Project Management

Software project management is defined as the process of managing, allocating, and timing resources to develop computer software that meets requirements. It is a systematic integration of technical, human and financial resources to achieve software development goals and objectives done in an efficient and expedient manner. Figure 7.1 shows the project management process.

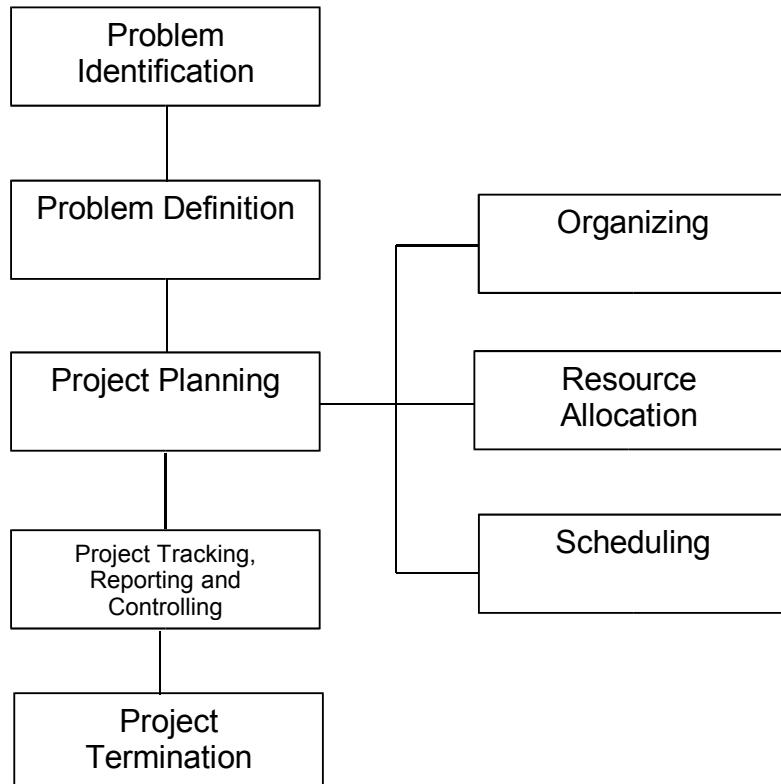


Figure 7.1 Project Management Process

It consists of eight (8) tasks.

Problem Identification

It is a task where a proposed project is identified, defined and justified. The proposed project may be a new product, implementation of a new process or an improvement of existing facilities.

Problem Definition

It is a task where the purpose of the project is clarified. The mission statement of the software project is the main output of this task. It should specify how project management may be used to avoid missed deadlines, poor scheduling, inadequate resource allocation, lack of coordination, poor quality and conflicting priorities.

Project Planning

It is a task that defines the ***project plan***. The project plan outlines a series of actions or steps needed to develop a work product or accomplish a goal. It specifies how to initiate a project and execute its objectives. Common components of the project plan include project objectives, project definition, team organization and performance criteria or validation criteria.

Project Organization

It is a task that specifies how to integrate the functions of the personnel in a project. It is usually done concurrently with project planning. It requires the skill of directing, guiding and supervising the project personnel. It is necessary that clear expectations for each personnel and how their job functions contribute in the overall goals of the project is identified.

Resource Allocation

It is the task of allocating resources such as money, people, equipment, tools, facilities, information, skills etc. to the project in order to achieve software goals and objectives.

Project Scheduling

It is the task of allocating resources so that overall project objectives are achieved within a reasonable time span. It involves the assignment of time periods to specific tasks within the work schedule. It uses techniques such as resource availability analysis (human, material, money) and scheduling techniques (PERT, CPM, Gantt Charts).

Tracking, Reporting and Controlling

It is a task that involves checking whether or not project results conform to project plans and performance specification. Controlling involves identifying and implementing proper actions to correct unacceptable deviations from expected performance. It involves the process of measuring the relationship between planned performance and actual performance with respect to project objectives. The variables to be measured, the measurement scales, and the measuring approaches should be clearly specified during planning activity. Corrective action may be rescheduling, reallocation of resources or expedition of task performance.

Project Termination

It is that task that involves submission of final report, the power-on of new equipment, or the signing of a release order. It may trigger follow-ups or spin-off projects.

7.2 Problem Identification and Definition

The development of a project starts with its initiation and definition. The purpose of this task is to make informed decisions when approving, rejecting and prioritizing projects. It requires that a product study request and proposal be given to provide a clear picture of the proposed project and the rationale behind it.

The main work product is the **project proposal**. It typically comes from a variety of sources such as client or user requirements and organization engineering, regulatory or legal requirements, market research, and new technologies. It should address background information, initial project boundaries and scope, technical feasibility, cost and benefit, and risk. It does not include details of the requirements, planning and design. Creating a project proposal requires a series of steps.

1. Define the need.
2. Identify alternative approaches to meet the need.
3. Recommend at least two alternatives.
4. Obtain approval

A proposal team can be established to do the project proposal. It should include expertise from the following functional area:

- Software/Hardware
- Network Support
- Data Processing Centers
- Data Security
- Database Administration
- Clients and Users
- Internal and External Auditors
- Other affective or support groups

All of the project proposal deliverables should be documented in the development plan or project file. The project proposal outline can be seen below.

- I. Executive Summary
 - A. Scope of the Project
 - B. Business and Information System Objectives
 - i. Business Objective
 - a) Primary Objective
 - b) Secondary Objective
 - ii. Information System Objectives
 - a) Primary Objective
 - b) Secondary Objective
- II. Success Criteria
- III. Alternatives
- IV. Schedule
- V. Costs
- VI. Benefits
- VII. Risk
- VIII. Recommendation

Executive Summary

It provides an introduction to the project which includes background information leading to the project's initiation and summary of work to be performed. It answers the questions why is the project important, what is the business opportunity or need, and what is the pertinent background information.

Scope

It defines the boundaries of the project. It describes the functions and process that are involved in the project. It answers the questions what is the ultimate achievement expected from this proposal, what will be included in the project, what will not be included in the project, who is the client, and who requested the project development effort.

Business and Information System Objectives

It provides specific targets and defines measurable results. When defining objectives, SMART (specific, measurable, attainable, result-oriented and time-oriented) and M&M's (measurable and manageable) principles are used. Measurements and results are in terms of time, cost and performance. It answers the following question: what are the desired results or benefits of this project, and what strategic, tactical or operational objectives are supported by this project.

Success Criteria

It identifies the high-level deliverables that must be in place in order for the project to be completed. It provides specific measurement used to judge success. It should support the project objectives. It answers the questions how will we know when the project is completed, who will judge the success of the project, and how will they judge the project's success.

Alternatives

It defines the alternative solutions to the business problem or need. It may be "make or buy" alternative or technical approach such as traditional coding, object-oriented development or integrated CASE, integration issues, business process reengineering.

Each alternative should be evaluated to determine the schedule. It answers the questions what are the major deliverables, what are the logical relationships between the major deliverables, what assumptions were used to develop the schedule, what is a reasonable schedule range based on size, effort, dependencies and assumptions.

Costs

It represents the estimated cost of developing the software. It is represented as a range of values. It answers the questions what are the costs associated with the effort required to produce each major deliverables, what are the costs associated with hardware and software, what other non-labor costs need to be considered, and what operating costs need to be considered and identified.

Benefits

It may be short or long term benefits. It includes potential savings, potential revenue, company or product positioning, improved quantity or quality, productivity improvements, and other tangible and intangible benefits.

Risks

It identifies risks of each alternative, and how the risks can be mitigated. It should be analyzed based on likelihood of occurrence and impact.

Recommendation

The proposal team should work together to select best alternative that best balances project schedule, cost and benefit, and risk. It should meet the project objectives in the best way.

7.3 Project Organization

The **project organization** integrates the functions of the personnel in a project. It is necessary that clear expectations for each personnel and how their job functions contribute in the overall goals and objectives of the project be communicated for smooth collaboration. This task needs an understanding on the systems development organization structure. Figure 7.2 illustrates this structure.

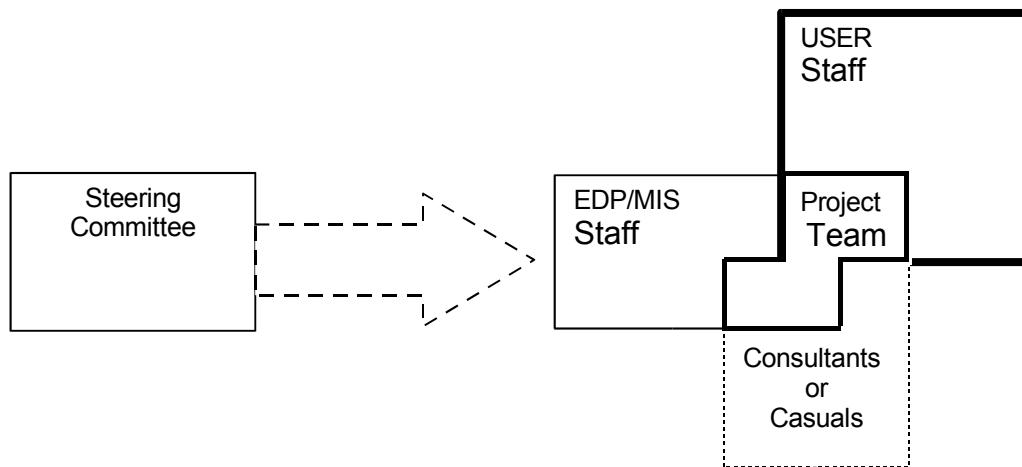


Figure 7.2 Systems Development Organization Structure

Steering Committee

The **Steering Committee** defines some business issues that have a great effect on the development of the project. It is composed of top management (the President, CEO or Managing Director or Partner), Line Management (Functional Vice-presidents or Division Heads), Staff Management (Corporate Planning, Comptroller), MIS or EDP Management and optionally, Consultants. Their functions consists of:

- formalizing the project team composition,
- reviewing and approving project plans and recommendations,
- approving and allocating resources for the project,
- synchronizing the system development effort with the rest of the organization's activities,
- regularly assessing progress of the project,
- resolving conflict situations,
- approving critical system change request, and
- releasing operational system to user.

Project Team

The **project team** is composed of two sets of groups, namely, developers and end-user. Developers consist of the project manager, analysts, designers, programmers, and quality assurance members. The end-user group consists of EDP or MIS Staff, user staff, casuals and consultants. It is a small group of with a leader or manager. They regularly interact to fulfill a goal. It requires team spirit, contribution and coordination.

7.3.1 The Project Team Structure

The team structure can be organized as democratic decentralized, controlled decentralized and controlled centralized.

Democratic Decentralized (DD)

The **Democratic Decentralized (DD)** Team Structure has no permanent leader. Rather, there are task coordinators appointed for a short duration who are replaced later on by others who may need to coordinate other task. Decisions are solved by group consensus. The communication is highly horizontal.

Controlled Decentralized (CD)

The **Controlled Decentralized (CD)** Team Structure has a permanent leader who primarily coordinates tasks. It has a secondary leader responsible for subtasks. Decisions are solved by group consensus but the implementation is done by subgroups. The communication during implementation is horizontal but control is vertical.

Controlled Centralized (CC)

The **Controlled Centralized (CC)** Team Structure has a team leader responsible for team coordination and top-level decision-making. The communication between leader and team members is vertical.

The decision on what team structure to employ would depend on the project characteristics. Use Table 33 to determine the team structure to be used for a project.

Team Type Project Characteristics	DD	CD	CC
Level of Problem Difficulty			
High	X		
Low		X	X
Size of Software (Lines of Code or Function Points)			
Large		X	X
Small	X		
Team Duration			
Short		X	X
Long	X		
<i>Modularity of the Program:</i>			
High		X	X
Low	X		
System Reliability Requirement			
High	X	X	
Low			X
Delivery Date Rigidity			
Strict			X
Lax	X	X	
Sociability Degree			
High	X		
Low		X	X

Table 33: Project Characteristics vs Team Structure

7.3.2 Project Responsibility Chart

The **Project Responsibility Chart** is a matrix consisting of columns of individual or functional departments, and rows of required actions. It supports the need to communicate expectations and responsibilities among the personnel involved in the development of the software. It avoids problems encountered with communication and obligations from being neglected. It answers the following questions:

- Who is to do what?
- How long will it take?
- Who is to inform whom of what?
- Whose approval is needed for what?
- Who is responsible for which results?
- What personnel interfaces are required?
- What support is needed from whom and when?

A sample project responsibility matrix is shown in Table 34.

	P r e s i d e n t	P r o j e c t	A n i e s t	P r o g r a m m e r	N O V 0 5	N O V 0 6	N O V 0 7	N O V 0 8	
1. Pre-joint Meeting Task									
1.1 Write product request.	R				D				
1.2 Set pre-joint meeting.	R	R			D				
1.2.1 Set time and date.	I	R	R		D				
1.2.2 Set place.	I	R	R		D				
1.2.3 Identify facilitator and participants.	C	I	R		O				
1.3 Invite participants.			R		O				
1.4 Distribute product request.	C		R						

Responsibility Code: R-Responsible I-Inform C-Consult S-Support

Task Code: D-Done O-On Track B-Delayed

Table 34: Project Responsibility Matrix

7.4 Project Scheduling

Project Scheduling is the task that describes the software development process for a particular project. It enumerates phases or stages of the projects, breaks each into discrete tasks or activities to be done, portrays the interactions among these pieces of work and estimates the time that each task or activity will take. It is a time-phased sequencing of activities subject to precedence relationships, time constraints, and resource limitations to accomplish specific objectives.

It is a team process that gives the start point of formulating a work program. It is iterative process and must be flexible to accommodate changes. There are certain basic principles used in project scheduling. Some of them are enumerated below.

1. *Compartmentalization.* The product and process are decomposed into manageable activities and tasks.
2. *Interdependency.* The interdependency of each compartmentalized activity or task must be determined. Tasks can occur in sequence or parallel. Tasks can occur independently.
3. *Time Allocation.* Each task should be allocated some number of work unit (person-days or man-days of effort). Each task must have a start and end date subject to interdependency and people responsible for the task (part-time or full-time).
4. *Effort Validation.* No more than the allocated number of people has been allocated at any given time.
5. *Define Responsibility.* Each task must have an owner. It should be a team member.
6. *Define Outcome.* Each task must have a defined outcome. Work products are combined in deliverables.
7. *Define Milestones.* Each task or group of tasks should be associated with a project milestone. Project milestones are reviewed for quality and approved by project sponsor.

Project Scheduling identifies activities or task sets, milestones and deliverables.

1. *Activities or Task Set.* An **activity or task set** is a collection of software engineering work tasks, milestones and deliverables that must be accomplished to complete a particular project. It is part of a project that takes place over a period of time. It is written as a verb-noun phrase.
2. *Milestones.* A **milestone** is an indication that something has been completed. It references a particular moment of time. It signifies points of accomplishments within the project schedule. It is not duration of work. Examples of project milestones are user signoff, approved system design, and system turnover.
3. *Deliverables.* A **deliverable** is a list of items that a customer expects to see during the development of the project. It can include documents, demonstrations of functions, demonstrations of subsystems, demonstrations of accuracy and demonstration of reliability, security or speed.

7.4.1 Project Work Breakdown Structure (WBS)

The **Project Work Breakdown Structure (WBS)** is a tool that allows project managers to define task sets, milestones and deliverables. It is a systematic analysis approach of depicting a project as a set of discrete pieces of work. The tasks and milestones can be used in a project to track development or maintenance. Two methods are used in defining the work breakdown structure, namely, the Work Breakdown Analysis and WBS Top-down Decomposition and Bottom-up Integration.

Work Breakdown Analysis

The Work Breakdown Analysis consists of the following steps:

1. Break the project into blocks of related activities.
2. Arrange the blocks into a logical hierarchy.
 - Analysis starts by identifying the major phases and the major deliverables that each produces.
 - For each activity, break them to define sub-activities and the work products produced by these sub-activities.
 - Continue to subdivide an activity until you get to an activity that cannot be subdivided anymore. This atomic activity is called the work unit or package.
3. Define the Work Unit or Package. The work unit or package is the responsibility of one person. It should be executed until it finishes without any interruption. Its duration and cost can be measured, and requires the continuous use of a resource group.

WBS Top-down Decomposition and Bottom-up Integration Process

This method defines two major activities as specified in its name, specifically, the top-down decomposition and bottom-up integration.

1. Top-down Decomposition
 - Identify 4-7 major components of work. Do not worry about the sequence.
 - Identify intermediate and final deliverables for each grouping.
 - Perform functional decomposition until a task has one owner, clear deliverables, credible estimates and can be tracked.
 - Use verb-object at lowest task level. Recommended number of levels is four (4).
 - Multiple iterations are required.
2. Bottom-up Integration
 - Brainstorm all possible tasks.
 - Organize task into 4-7 major groupings reflecting how the project will be managed.

7.4.2 Work Breakdown Schedule Format

There are two common formats that can be used to represent the WBS, namely, graphical or hierarchy chart and outline format.

Graphical or Hierarchy Chart

An example of a hierarchy chart of Pre-joint Meeting Task of Requirements Engineering is shown Figure 7.3.

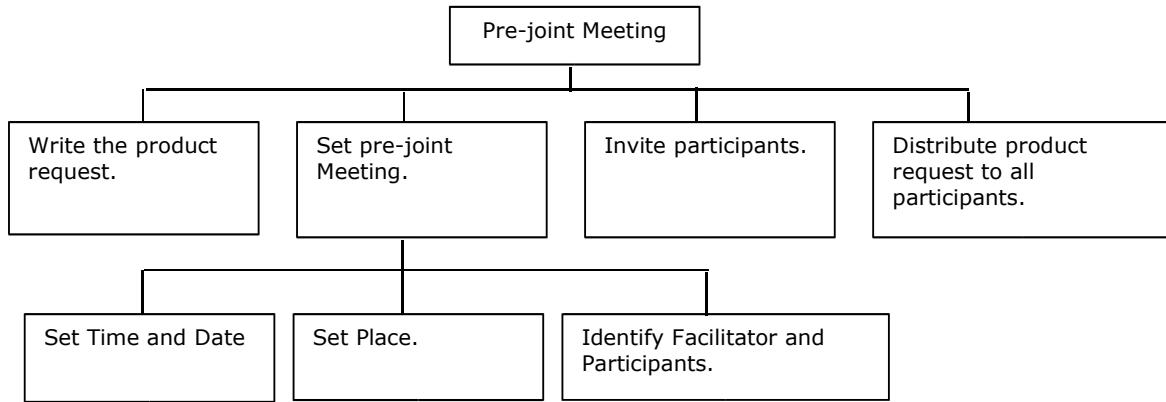


Figure 7.3 Hierarchy Chart

Outline Format

A sample outline format is shown in below.

1. Pre-joint Meeting Task
 - 1.1 Write product request.
 - 1.2 Set pre-joint meeting.
 - 1.2.1 Set time and date.
 - 1.2.2 Set place.
 - 1.2.3 Identify facilitator and participants
 - 1.3 Invite participants
 - 1.4 Distribute product request to all participants.

GANTT Chart

It is an easy way of scheduling tasks. It is a chart that consists of bars that indicates the length of the tasks. The horizontal dimension indicates the time while the vertical dimension indicates the task. Table 35 shows an example of a GANTT Chart.

	1	2	3	4	5	6
1. Pre-joint Meeting Task						
1.1 Write product request.						
1.2 Set pre-joint meeting.						
1.2.1 Set time and date.						
1.2.2 Set place.						
1.2.3 Identify facilitator and participants.						
1.3 Invite participants.						
1.4 Distribute product request.						

Table 35: Sample GANTT Chart

7.5 Project Resource Allocation

The **project resource allocation** is the process of allocating or assigning money, people, equipment, tools, facilities, information, skills etc to the tasks identified in the project. They are needed in order to achieve software goals and objectives. There are three major project constraints that are always considered in a software development project. They are:

- Time Constraints
- Resource Constraints
- Performance Constraints

7.5.1 Resource Availability Data Base

The project manager must use the resource availability database to manage resources allocated for each task set. The Resource Availability Database specifies what resources are required against the resources that are available. Table 36 shows an example.

Resource Type	Resource ID	Skills	Date Needed	Duration	Number of Resources
Type 1	Project Manager	Planning Managing	1/1/XX	10 months	1
Type 2	Analyst	Developing Analysis Model	12/25/XX	Indefinite	2
Type 3	Designer	Developing Software Architecture and Components	Now	36 Months	2
...
...
...
Type n-1	Operator	Machining	Immediate	Indefinite	4
Type n	Programmer	Software Tools	9/2/XX	12 months	5

Table 36: Resource Availability Database

There are some factors to be considered when allocating resources. Some of them are listed below.

- Limitations on resource availability
- Precedence restrictions
- Activity-splitting restrictions
- Non-pre-emption of activities
- Project Deadlines
- Resource Substitutes
- Partial Resource Assignments

- Mutually exclusive activities
- Variable resource availability
- Variable activity durations

Allocating Resources to the Project Task Set.

Project Task Set	Resources
1 1. Pre-joint Meeting Task	
1.1 Write product request.	End-user who will write the product request. Computer Office Supplies
1.2 Set pre-joint meeting.	
1.2.1 Set time and date.	Project Leader
1.2.2 Set place.	Project Leader
1.2.3 Identify facilitator and participants.	Project Leader
1.3 Invite participants.	Administrative staff E-mail Office supplies
1.4 Distribute product request.	Project Leader Computer Office supplies

7.6 Software Metrics

Software Metrics refers to a variety of measure used in computer software. Measurements can be applied to the software process with the intent of continuing the improvement of the process. It can be used throughout a software project to assist in estimation, quality control, productivity assessment and project control. It can be used by software engineers to help assess the quality of work products and to assist in the tactical decision-making as project proceeds.

Software Metrics are used as indicators to provide insight into the software process, project or the software product. It allows software engineers to make decisions on how to progress within the development effort.

Categories of Measurement

There are roughly two categories of measurements:

1. *Direct Measures.* In software process, it can be the cost and effort applied. In software, it can be the lines of codes (LOC) produced, number of classes, execution speed, memory size and defects reported over a period of time
2. *Indirect Measures.* It can be product measures such as functionality, quality, complexity, efficiency, reliability, maintainability etc.

Examples of metrics that are applied to software projects are briefly enumerated below.

1. *Productivity Metrics.* It refers to the measure on the output of software process.
2. *Quality Metrics.* It refers to the measure that provides indication of how closely software conforms to the requirements. It is also known as the software's fitness for use.
3. *Technical Metrics.* It refers to the measure on the characteristics of the product such as complexity, degree of collaboration and modularity etc.
4. *Size-Oriented Metrics.* It is used to collect direct measures of software output and quality based on the line of codes (LOC) produced.
5. *Function-Oriented Metrics.* It is used to collect direct measures of software output and quality based on the program's functionality or utility.
6. *Human-Oriented Metrics.* It provides measures collected about the manner in which people develop computer software and human perception about the effectiveness of tools and methods.

Different phases or activities of the development projects required different metrics. Each chapter provides the metrics that are normally used to help software engineers the quality of the software and the process.

7.6.1 Size-oriented Metrics- Lines of Codes (LOC)

A common direct measure used for software metrics is the Lines of Code (LOC). It is a size-oriented metrics used to estimate effort and cost. It is important that the metrics of other projects are placed within a table to serve as reference during estimation. Table 37 shows an example of this.

Project	Effort in perso n- month s	Cost (in P)	KLOC (thous and lines of code)	Pages (of Doc.)	Error s	Peopl e
Project One	24	120	12.1	365	29	3
Project Two	62	450	27.2	1224	86	5
Project Three	43	213	20.2	1050	64	6
Project Four	36	30	15	17	10	5
...
...
...

Table 37: Software Metrics History

In this table, effort in person-month specifies the number of months the project was developed, cost is the money spent producing the software, KLOC is the lines of codes produced, pages is the number of pages of documentation, errors is the number of errors reported after the software was delivered to the customer, and people is the number of people who developed the software.

One can derive metrics from this table. Example computations are shown below.

$$\text{Productivity} = \text{KLOC} / \text{person-month}$$

$$\text{Quality} = \text{errors} / \text{KLOC}$$

$$\text{Cost} = \text{Cost} / \text{KLOC}$$

$$\text{Documentation} = \text{Pages} / \text{KLOC}$$

7.6.2 Function-Oriented Metrics: Function Points (FP)

Another metric that is commonly used is the Function Point (FP) Metrics. It focuses on the program functionality or utility. It uses an empirical relationship based on countable measures of software's information domain and assessments of software complexity.

Computing the Function Point

STEP 1. Determine the value of the information domain by using the table shown in Table 38.

Fill-up the count column and multiply the value with the chosen weight factor. As an example, if the number of input is 3 and the input data are simple, chose the weight factor of 3, i.e., $3 \times 3 = 9$.

Information Domain		Weighting Factor			
Measurement Parameter	Count	Simple	Average	Complex	
Number of User Inputs		x 3	4	6	=
Number of User Outputs		x 4	5	7	=
Number of User Inquiries		x 3	4	6	=
Number of Files		x 7	10	15	=
Number of External Interfaces		x 5	7	10	=
Count-Total					

Table 38: Computation of Information Domain

Find the sum (Count-Total).

STEP 2. Determine the complexity adjustment value.

To determine the complexity adjustment value, use the scale in Table 39 in answering the questions specified below.

0	1	2	3	4	5
No Influence	Incidental	Moderate	Average	Significant	Essential

Table 39: Complexity Adjustment Scale

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

STEP 3. Compute for the function point.

Use the formula indicated below.

$$\text{FunctionPoint} = \text{Count-Total} \times [0.65 + 0.01 \times \text{sum } (F_i)]$$

One can also derive metrics similar to size-oriented metrics. Examples of that are listed below.

$$\text{Productivity} = \text{FP} / \text{person-month}$$

$$\text{Quality} = \text{Defects} / \text{FP}$$

$$\text{Cost} = \text{Cost} / \text{FP}$$

$$\text{Documentation} = \text{Pages of Documentation} / \text{FP}$$

7.6.3 Reconciling LOC and FP Metrics

2 To reconcile LOC and FP Metrics, we use the Table 40¹⁰.

Programming Language	LOC/FP (Average)
Assembly Language	337
COBOL	77
Java	63
Perl	60
Ada	154
Visual Basic	47
SQL	40
C	162
C++	66

Table 40: Reconciliation of LOC and FP

¹⁰ Pressman, Software Engineering: A Practitioner's Approach, p. 657)

7.7 Project Estimations

Project estimations are necessary to help software engineers determine the effort and cost required in building the software. To do estimation, it would require experience, access to good historical data, and commitment to quantitative measures when qualitative data are all that exists.

One can use LOC and LP to provide cost and effort estimates. They provide estimate variables to "size" each element of the software. They provide a baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

As an example, consider the project estimates for the club membership maintenance system. To use LOC and FP to determine the cost and effort estimate, we computer first the FP.

Information Domain		3	Weighting Factor			
4 Measurement Parameter	5 Count	6 Simple	Average	Complex		
7 Number of User Inputs	8 2	9 x 3	4	6	6	
10 Number of User Outputs	115	12 x 4	5	7	20	
13 Number of User Inquiries	143	15 x 3	4	6	9	
16 Number of Files	172	18 x 7	10	15	14	
19 Number of External Interfaces	202	21 x 5	7	10	10	
						Count-Total 59

1. Does the system require reliable backup and recovery? 3
2. Are data communications required? 3
3. Are there distributed processing functions? 1
4. Is performance critical? 2
5. Will the system run in an existing, heavily utilized operational environment? 3
6. Does the system require on-line data entry? 4
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 2
8. Are the master files updated on-line? 2
9. Are the inputs, outputs, files, or inquiries complex? 1
10. Is the internal processing complex? 1
11. Is the code designed to be reusable? 4
12. Are conversion and installation included in the design? 3
13. Is the system designed for multiple installations in different organizations? 1
14. Is the application designed to facilitate change and ease of use by the user? 1

Total= 31

$$FP = 59 \times [0.65 + 0.01 \times 31] = 56.64 \text{ FP}$$

The project heavily uses JAVA. The estimated LOC is computed as follows:

$$\text{LOC} = 56.64 * 63 = 3568.32 \text{ LOC}$$

Derived Estimates: Assume that the Club Membership Maintenance is similar to Project Four.

Cost Estimates are computed as follows:

Cost of Producing a single LOC (From the Database): KLOC / Cost

$$\text{Cost/KLOC} = \$30,000 / 15,000 \text{ LOC} = \$2.00 / \text{LOC}$$

$$\text{Cost of the project} = 3568.32 * 2.00 = \$7136.64$$

Effort Estimates are computed as follows:

Number of LOC produced in a month (From the Database): KLOC/Effort

$$\text{KLOC/Effort} = 15,000 \text{ LOC} / 8 \text{ months} = 1875 \text{ LOC} / \text{month}$$

$$\text{Effort of the project} = 3568.32 / 1875 = 1.9 \text{ or } 2 \text{ months}$$

7.8 Writing the Project Plan

The project plan is a document that contains the plan for developing the software. It need not be lengthy or complex. Its purpose is to help establish the viability of the software development effort. The plan concentrates on the general statements of "what" and a specific statement of how much and how long. The details will be presented as the development progresses. The project plan would have the following outline.

- I. Introduction
 - A. Background of the Study
 - B. Project Scope and Objectives
 - C. Major Functions and Data
 - D. Performances Issues
 - E. Constraints and Limitations
- II. Project Estimates
 - A. Effort Estimates
 - B. Cost Estimates
- III. Project Schedule
- IV. Project Resources
- V. Project Organization

7.9 Risk Management

Risk Management consists of steps that help the software engineers to understand and manage uncertainty within the software development process. A **risk** is considered a potential problem, i.e., it may happen or it may not. It is better to identify it, assess its probability of occurring, estimate its impact and establish a contingency plan if it does happen. Many things can happen during the progress of the development of the software. It is for this reason that understanding risks and having measures that avoid them influences how well the project is managed.

Two important characteristics of risk are always analyzed and identified in risk management. Risk always involved uncertainty. This means that risks may or may not happen. If a risk becomes a reality, bad consequences and losses will occur. The **level of uncertainty** and **degree of loss** associated for each risk are identified and documented.

There are different categories of risks that the software engineer needs to understand. They should be considered as potential problems may arise from them, and understanding them also allows the creation of contingency plan with minimal impact to the software development project.

1. *Project Risks.* These are risks associated with the project plan. If such risk becomes a reality, chances are there would be delays, budgetary and resource problems.
2. *Technical Risks.* These are risks associated with the implementation aspects of the software. If such risk becomes a reality, it would be very difficult for us to implement the project. This may include design and interface problems, component problems, database and infrastructure problems.
3. *Business Risks.* These are risks associated with the viability of the software. If such risk becomes a reality, the software will become a problem in a business setting. This may include building a software that no one needs, building a software that cannot be market and losing budgetary and personnel commitment.
4. *Known Risks.* These are risks that can be detected with careful evaluation of the project plan, business environment analysis, and other good informational sources.
5. *Predictable Risks.* These are risks that came from past similar projects that are expected to happen with the current one.
6. *Unpredictable Risks.* These are risks that are not detected.

7.9.1 The Risk Table

The **Risk Table** is a tool that allows software engineer to identify and manage risks within the software development project. Table 41 shows an example of a risk table.

Risks Identified	Category	Probability	Impact	RMMM
Project Size Estimates may be too low.	PS	60%	2	
A large number of group of end-users are not identified.	PS	30%	3	
No components are reused.	PS	70%	2	
Members of the steering committee are not interested in the project.	BU	40%	3	
Project deadline is not negotiable.	BU	50%	2	
Staff turnover will be high	ST	60%	1	
...

Table 41: Sample Risk Table

First column lists down all possible risk including those that are most likely not to happen. Each risk are then categorized. Categorization identifies the type of risks, and they are briefly enumerated below.

1. *Product Size.* These are risks that are related to the overall size of the software that needs to be developed.
2. *Business Impact.* These are risks that are related to the constraints imposed by management or market.
3. *Customer Characteristics.* These are risks that are related to end-user's requirements and the ability of the development team to understand and communicate with them.
4. *Process Definition.* These are risks that are related to the software process and how the people involved in the development effort are organized.
5. *Development Environment.* These are risks that are related to the availability and quality of the tools that are used to develop the software.
6. *Technology.* These are risks that are related to the complexity of the system and the degree of newness that is part of the software.
7. *Staff Size and Experience.* These are risk that are related to the overall technical and project experience of the software engineers.

The probability value of the risk is identified as a percentage value of it occurring. Next, the impact of the risk when it occurs is identified. It is assessed using the following values.

- 1- *Catastrophic.* It would result in failure to meet the requirements and non-acceptance of the software.
- 2- *Critical.* It would result in failure to meet the requirements with system

performance degradation to the point of mission success is questionable.

3- *Marginal*. It would result in failure to meet the requirements would result in degradation of secondary mission.

4- *Negligible*. It would result in inconvenience or usage problems.

The last column contains the RMMM (Risk Mitigation, Monitoring and Management) plan for each risk. It contains the following components:

1. *Risk ID*. It is a unique identification number that is assigned to the risk.
2. *Description*. It is a brief description of the risk.
3. *Risk Context*. It defines the condition by which the risk may be realized.
4. *Risk Mitigation and Monitoring*. It defines the steps that need to be done in order to mitigate and monitor the risk.
5. *Contingency Plan*. It defines the steps that need to be done when the risk is realized.

7.9.2 Risk Identification Checklist

One can use the following to determine risks within the software development project.

Product Size Risks

The product size is directly proportional to the project risk. As software engineers, we take a look at the following:

- How large is the estimated size of the software in terms of lines-of-code and function points?
- How many is number of programs, files and transactions?
- What is the database size?
- How many is the estimated number of users?
- How many components are reused?

Business Impact Risks

- What is the business value of the software to the company or organization?
- What is the visibility of the software to senior management?
- How reasonable is the delivery date?
- How many is the customers?
- How many are the products or systems that interoperate with the software?
- How large is the project documentation?
- Does the product documentation have quality?
- What are the governmental constraints applied to the development of the project?
- What is the cost associated with late delivery?
- What is the Cost associated with a defective product?

Customer Related Risks

- What is the customer working relationship?
- What is the level of the customer's ability to state their requirements?
- Are customers willing to spend time for requirements gathering?
- Are customers willing to participate in formal technical reviews?
- How knowledgeable are the customers in terms of the technological aspects of the software?
- Do customer understand of the software development process and their role?

Process Risks-Process Issues

- Does senior management show full support or commitment to the software development effort?
- Do the organization have a developed and written description of the software process to be used on this project?
- Do the staff members know the software development process?
- Is the software process used before in other projects with the same group of people?
- Has your organization developed or acquired a series of software engineering training courses for managers and technical staff?
- Are there any published software engineering standards that will be used by software developer and software manager?
- Are there any developed document outlines and examples of deliverables?
- Are formal technical reviews of the requirements specification, design and code done regularly? Do they have defined procedures?
- Are formal technical reviews of test procedures, test cases done regularly? Do they have defined procedures?
- Are the results of each formal technical review documented, including errors found and resources used? Do they have defined procedures?
- Do they have procedures that ensure that work conducted on a project conforms to software engineering standards?
- Do they use configuration management to maintain consistency among system/software requirements, design, code, and test cases?
- Do they have documents for the statement of work, software requirements specification, and software development plan for each subcontract?

Process Risks-Technical Issues

- Are there any mechanism that facilitates clear communication between customer and developer?
- Are there any specific methods used for software analysis?
- Are there any specific methods used for component design?

- Are there any specific method for data and architectural design?
- Are codes written in a high level language? How much is the percentage?
- Are documentation and coding standards followed?
- Are there any specific methods for test case design?
- Are software tools used to support planning and tracking activities?
- Are configuration management software tools used to control and track change activity throughout the software process?
- Are CASE tools used?
- Are quality metrics collected for all software projects?
- Are there any productivity metrics collected for previous software projects?

Technology Risks

- Is the organization new to the technology?
- Do we need to create new algorithms, input or output technology?
- Do the software need to use new or unsupported hardware?
- Do the software need to use a database system not tested in the application area?
- Do we need to define a specialized user interface?
- Do we need to use new analysis, design, or testing methods?
- Do we need to use unconventional software development methods, such as formal methods, AI-based approaches, and artificial neural networks?

Development Environment Risks

- Are there any software project management tools employed?
- Are there any tools for analysis and design?
- Are there any testing tools?
- Are there any software configuration management tools?
- Are all software tools integrated with one another?
- Do we need to train the members of the development team to use the tools?
- Are there any on-line help and support groups?

Risk Associated with Staff Size and Experience

- Is the staff turnover high?
- Do we have skilled team members?
- Is there enough number of people in the development team?
- Will the people be required to work overtime?
- Will there any problems with the communications?

7.10 Software Configuration Management

Software Configuration Management is an umbrella activity that supports the software throughout its life cycle. It can occur any time as work products are developed and maintained. It focused on managing change within the software development process. It consists of identifying change, controlling change, ensuring that change is being properly implemented, and reporting the change to others who may be affected by it. It manages items called **software configuration items (SCIs) or units**, which may be computer programs (both source-level and executable forms), documents (both technical or user) and data.

The possible work products that can serve as software configuration items or units are listed below. They are characterized based on the software engineering phase or activities they are produced.

1. Project Planning Phase
 - a) Software Project Plan
2. Requirements Engineering
 - a) System Specifications
 - b) Requirements Model
 - c) Analysis Model
 - d) Screen Prototypes
3. Design Engineering
 - a) Database Design
 - b) Dialogue and Screen Design
 - c) Report Design
 - d) Forms Design
 - e) Software Architecture
 - f) Component Design
 - g) Deployment Design
4. Implementation
 - a) Source Code Listing
 - b) Executable or Binary Code
 - c) Linked Modules
5. Testing
 - a) Test Specifications
 - b) Test Cases
6. Manuals
 - a) User Manuals

b) Technical Manuals

c) Installation and Procedures for Software Engineering

7. Standards and Procedures for Software Engineering

7.10.1 Baseline

Managing SCIs uses the concept of baseline. A **baseline** is a software configuration management concept that helps software engineers control change without seriously impeding justifiable change. It is a specification or product that has been formally reviewed and agreed upon to be the basis for further development. It can only be changed through a formal change procedure. Figure 7.4 shows the dynamics of managing the baseline of SCIs.

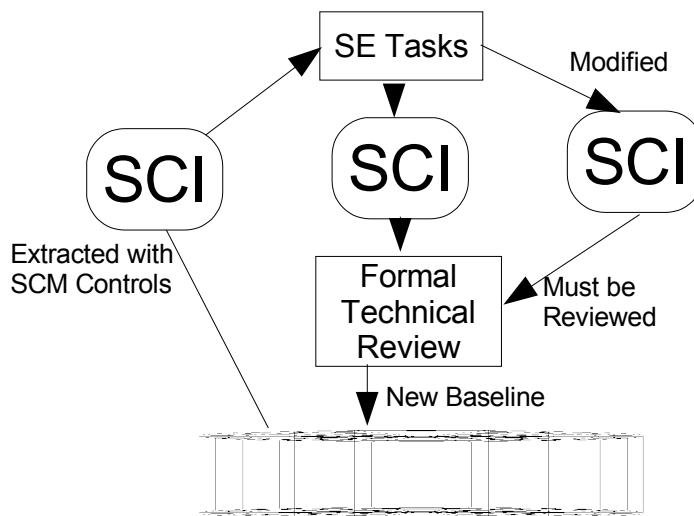


Figure 7.4 SCIs Baseline Dynamics

A Software Engineering Task develops a work product that becomes a software configuration item or unit. It goes through a formal technical review to detect faults and errors. Once the work product passes or is acceptable, it becomes a baseline and is saved in a project database. Sometimes, this work product is extracted from the project database as input to some other software engineering task. Notices that "checking out" a SCI involves controls and procedure. During the execution of the SE task, it is possible that the SCI has been modified. In this case, the modified SCI will pass through another formal technical review. If it passed or acceptable, the modified version becomes the new baseline.

7.10.2 Software Configuration Tasks

Software configuration management has the responsibility to control change. It identifies software configuration items and various version of the software. It also audits the software configuration items to ensure that they have been properly developed and that the reporting of changes is applied to the configuration. Five tasks are done, namely,

change identification, version control, change control, configuration audit and reporting.

Change Identification

It is the process of identifying items that change throughout the software development process. It specifically identifies software configuration items (SCIs) and uniquely gives it an identifier and name.

Version Control

It is the procedures and tools to manage the different version of configuration objects that are created during the software development process. A version is a collection of SCIs that can be used almost independently.

Change Control

It consists of human procedures and automated tools to provide a mechanism for the control of change. It has the following subset of task:

- A change request is submitted and evaluated to assess technical merit, potential side effects, overall impact on other SCIs.
- A change report is created to identify the final decision on the status and priority of the change.
- The Engineering Change Order (ECO) is created that describes the change to be made, the constraints, and the criteria for review and audit.
- The SCIs is checked out for modification.
- It is, then, checked in after the review and subject to version control.

Configuration Audit

It is the process of assessing a SCIs for characteristics that are generally not considered during a formal technical review. It answers the following questions:

- Was the change specified in the ECO done? Are there any additional modification made?
- Did the formal technical review assessed the technical correctness of the work product?
- Did the software engineering procedures been followed?
- Did the SCM procedures been followed?
- Have all appropriate SCIs been updated?

Status Reporting

It is the process of informing people that are affected by the change. It answers the following questions:

- What happened?
- Who did it?
- When did it happen?
- What else will be affected?

7.11 Project Assignment

The objective of the project assignment is to reinforce the knowledge and skills gained in Chapter 8. Particularly, they are:

1. Developing the Project Plan

GROUP TASK:

1. The group should create the project plan for their project as an exercise.

MAJOR WORK PRODUCT:

Software Project Plan

8 Software Development Tools

The implementation of software requires a variety of software tools. Ensuring that these are available in compatible versions and with sufficient licences is part of project management. Many of these tools have been designed and implemented to make the work of the software engineer easier and organized. In this chapter, software development tools are presented.

8.1 Case Tools

Computer-aided Software Engineering (CASE) tools are software that allows the software analysts create models of the system and software. There are now several tools that support the Unified Modeling Language (UML). The main feature of such software is the project repository which allows the maintenance of the links among the textual and structured descriptions of every class, attributes, operation, state, etc, to the diagrammatic representation.

Other CASE Tools supports code generators. This ensures that the implementation reflects the design diagrams. There are tools that generate Java, Visual Basic, C++ and SQL codes. Examples of CASE Tools are ArgoUML, umlet, Rational Rose, and Dia.

8.2 Compilers, Interpreters and Run-time Support

All source code must be translated into an executable code. In Java, the source code is compiled into a intermediary bytecode format that is interpreted by the Java Virtual Machine (JVM). This environment is supported by most browsers; it means that you can run java programs using simply a browser.

8.3 Visual Editors

Creating graphical user interfaces can be difficult when done manually. Visual Editors allows the creation of these interface by simply dragging and dropping the components onto the forms and setting the parameters that control their appearance in a properties window. NetBeans supports this kind of visual editing.

8.4 Integrated Development Environment (IDE)

Integrated Development Environment (IDE) incorporates a multi-window editor, mechanisms for managing files that make up the projects, links to the compiler so that codes can be compiled from within the IDE, and a debugger to help the programmer step through the code to fine errors. Some IDEs have visual editors. This type of software development tools enable a software engineer to manage source codes in a single interface, and at the same time, can perform some testing and debugging. NetBeans is an example of an IDE.

8.5 Configuration Management

Configuration Management Tools are tools that support the tracking of changes and dependencies between components and versions of source codes and resource files that are used to produce a particular release of a software package. This type of tool supports the Software Configuration Management process. Example of this tool is Concurrent Version System(CVS).

8.6 Database Management Tools

A considerable amount of software is needed for a large-scale database management system. If support for client-server mode of operation is needed, a separate client and server components should be installed. Such as when ODBC or JDBC is used, additional special libraries or Java packages (specific DriverManager of a vendor) should be installed or available during run-time. ObjectStore PSE Pro includes a post-processor that is used to process Java class files to make them persistent.

8.7 Testing Tools

These are tools that identifies test methods and generates test cases. Automated testing tools are available for some environment such as NetBeans' support for JUnit. What is more likely is that programmers will develop their own tools to provide harnesses within which to test classes and subsystems.

8.8 Installation Tools

This tool allows the creation of installation executable files that when run automates the creation of directories, extraction of files from the archives and the setting up of parameters or registry entries.

8.9 Conversion Tools

This type of tool is needed when an existing system needs to transfer its data to another system. Nowadays, new software replaces existing computerized system. Packages like Data Junction provide automated tools to extract data from a wide range of systems and format it for a new system.

8.10 Document Generator

These are tools that allow the generation of technical or user documentation. Java has javadoc that processes Java source files and generates HTML documentation in the style of the API documentation from special comments with embedded tags in the source code.

8.11 Software Project Management

Software Project Management tools supports project management. It allows the project manager to set the project schedule, manage resource such as time and people, track progress. MS Project is an example of this tool.

Appendix A – Installing Java Studio Enterprise 8

Installing Java Studio Enterprise 8 in a Linux Environment

Pre-installation Notes

Collaboration Runtime

The Collaboration Runtime uses port 5222 by default. If you have Java Studio Enterprise 7 Collaboration Runtime and want to use both Java Studio Enterprise 7 and 8 Collaboration Runtimes at the same time, start the Java Studio Enterprise 7 Collaboration Runtime before starting version 7's IDE installation. Doing this ensures the installer use a different port for the Collaboration Runtime.

Installation Log File and Temporary Directory

If you want a log file for installation reports, you must start the installer from a terminal window using the following commands.

```
> cd /installer-location  
> ./jstudio_ent8_linux-jds.bin -is:log log-file-name
```

If you want to specify a temporary directory for your installation, use the following commands in the command line dialog.

```
> cd /installer-location  
> ./jstudio_ent8_linux-jds.bin -is:tempdir temp-directory
```

Installation of the Java Studio Enterprise Software

There are two ways to install the Java Studio Enterprise 8 Software in your Linux machine – from media or a download file.

Media Installation

1. Insert the Java Studio Enterprise 8 media in your CD/DVD drive.
2. Type in the following commands in a terminal window:

```
> cd /media-drive/volume-label  
> ./installer.sh
```

3. Follow the instructions in the **Next Installation Steps** subsection.

Download File Installation

1. Download the Java Studio Enterprise installer file.
2. Type in the following commands in a terminal window:

```
> chmod +x installer-label  
> ./installer-label
```

3. Follow the instructions in the **Next Installation Steps** subsection.

Next Installation Steps

1. Select **Next** at the welcome page of the installation wizard.
2. Read the license agreement and click **Next**.
3. For the *Select Features* page, use the following table to guide you on selecting features to install.

If	Then
You want to use your installation of <i>Sun Java System Application Server 8.1 Platform Edition, Update Release 2</i> with this IDE installation	Untick the <i>Sun Java(tm) System Application Server 8.1 PE UR2</i> checkbox, and click Next . Provide the necessary information about your application server in the <i>Application Server Selection</i> page, and click Next .
You do not want to install the Collaboration Runtime	Untick the <i>Collaboration Runtime</i> checkbox, and click Next .
You want to install all the features	Click Next .

4. Choose which Java Development Kit (JDK) you want to use for the installation of this IDE at the *Java Development Kit Selection* page and click **Next**.
5. Verify the settings at the *Start Copying Files* page and click **Next** to begin installation.
6. Tick the *Start Sun Java(TM) Studio Enterprise 8* checkbox if you want to launch the IDE after the installation wizard finishes.
7. Click **Finish**.

Post-installation Notes

The installed IDE can be run using a typing a run command in a terminal window.

```
> cd /location-of-IDE-executable  
> ./runide.sh
```

Reference

http://docs.sun.com/source/819-2807/chap_linux_installing.html

References

Florence Balagtas, JEDI Course: Introduction to Programming I, 2005

Rebecca Ong, JEDI Course: Introduction to Programming II, 2005

Roger S. Pressman, Software Engineering: A Practitioner's Approach 6th Edition, McGraw-Hill International Edition, 2005

Shari Lawrence Pfleeger, Software Engineering Theory and Practice, Prentice-Hall International, Inc., 1999

Ian Sommersville, Software Engineering 6th Edition, Pearson Education Limited, Inc., 2001

Simon Bennett, Steve Robb and Ray Farmer, Object-oriented System Analysis and Design Using UML 2nd Edition, McGraw-Hill Companies, 2002

Object-oriented Analysis and Design Using the UML Student Manual, Rational Software Corporation, 2000

Hans-Erik Erikson and Magnus Penker, Business Modeling with UML, Business Patterns at Work, Wiley Computer Publishing, 2000

Chidamber, S.R., and C.F. Kemerer, A Metrics Suite for Object-oriented Design, IEEE Trans. Software Engineering, vol SE-20, no. 6, June 1994

Floyd Marinescu, EJB Design Patterns, The Middleware Company, 2002

Designing Enterprise Applications 2nd Edition,
http://java.sun.com/blueprints/guidelines/designing_enterprise_application_2e, Last Accessed: November 2005

J2EE Design Patterns, <http://java.sun.com/blueprints/patterns/index.html>, Last Accessed: November 2005

Lionel Briand and Yvan Labiche, A UML-based Approach to System Testing, Carleton University TR-SCE-01-01 Version 4, June 2002

Manish Nilawar, A UML-based Approach for Testing Web Applications, University of Nevada, August 2003

Wellie Chao, Testing Java Classes with Junit, <http://www.theserverside.com>, Last Accessed: 2005

Beck, Kent and Erich Gamma, Junit Cookbook,
<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

Junit, <http://www.devdaily.com/java/junit/node11.shtml>, Last Accessed: November 2005

Simpson, Blaine, JUnit Howto, <http://www.admc.com/blaine/howtos/junit/junit.html>, Last Accessed: November 2005

Wong, Stephen and Dung Nguyen, Unit Testing with JUnit in DrJava,
<http://cnx.rice.edu/content/m11707/latest/>