ADVANCED SYSTEMS DESIGN WITH JAVA UML AND MDA

KEVIN LANO

Advanced Systems Design with Java, UML and MDA

Kevin Lano

AMSTERDAM ● BOSTON ● HEIDELBERG ● LONDON ● NEW YORK ● OXFORD
PARIS ● SAN DIEGO ● SAN FRANCISCO ● SINGAPORE ● SYDNEY ● TOKYO

Elsevier Butterworth-Heinemann
Linacre House, Jordan Hill, Oxford OX2 8DP
30 Corporate Drive, Burlington MA 01803

First published 2005

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloguing in Publication Data
A catalogue record for this book is available from the Library of Congress

ISBN 0 7506 6496 7

Printed and bound in Great Britain

Contents

Preface vii

1 The Challenges of Software Design 1
- 1.1 Software development 1
- 1.2 Software development methods 3
- 1.3 Software development steps 7
- 1.4 Summary 13

2 The Unified Modelling Language 14
- 2.1 Introduction 14
- 2.2 Use case diagrams 15
- 2.3 Class diagrams 18
- 2.4 Object diagrams 30
- 2.5 Creating a platform-independent model 32
- 2.6 Exercises 36

3 The Object Constraint Language 40
- 3.1 Using OCL and LOCA Constraints 40
- 3.2 OCL operations 46
- 3.3 Navigation expressions 48
- 3.4 Quantifiers 52
- 3.5 Association constraints 53
- 3.6 Recursion 55
- 3.7 Identity attributes 58
- 3.8 Association classes 61
- 3.9 Implicit and explicit associations 62
- 3.10 Interfaces and implementation inheritance 63
- 3.11 Packages, Subsystems and Models 66
- 3.12 Other class diagram elements 68
- 3.13 Exercises 69

4 UML Dynamic Modelling Notations 72
- 4.1 Statecharts 72
- 4.2 Interaction diagrams 81
- 4.3 The Scrabble system revisited 83
- 4.4 Consistency and completeness checking of UML models 90
- 4.5 UML tools 94
- 4.6 Summary 95
- 4.7 Exercises 95

5 Platform-Independent Design 97
- 5.1 The design process 97
- 5.2 Model transformations 98
- 5.3 Design patterns 100
- 5.4 System and subsystem design 113
- 5.5 Detailed design 115
- 5.6 Constructing a design architecture 117
- 5.7 User-interface design 118
- 5.8 Data repository design 121
- 5.9 Exception handling 125
- 5.10 Transforming a PIM to a PSM 125
- 5.11 Exercises 128

6 From Platform-Specific Models to Executable Code 132
- 6.1 Production of a Java implementation 132
- 6.2 Synthesis of Java code 135
- 6.3 Synthesis case study: railway signalling system 138
- 6.4 Synthesis case study: inheritance example 142
- 6.5 Synthesis case study: derived attributes example 145
- 6.6 Data repository implementation: choosing a database 148
- 6.7 Production of an XML data repository 148
- 6.8 Summary 155
- 6.9 Exercises 155

7 Internet System Design 158
- 7.1 Fundamentals of internet systems 158
- 7.2 Design of internet systems 162
- 7.3 Design process for internet systems 209
- 7.4 Summary 210
- 7.5 Exercises 211

8 Web Services 215
- 8.1 Definitions of web services 215
- 8.2 J2EE 217
- 8.3 .Net 222
- 8.4 Communicating with web services 223
- 8.5 Examples of web services 225

8.6 Web service design guidelines 234
8.7 Implementing web services using J2EE 234
8.8 Mailing and Push technologies 239
8.9 The Semantic Web 243
8.10 Mobile computing and m-commerce 244
8.11 Summary 247
8.12 Exercises 247

9 Implementing the Model-Driven Architecture 249
9.1 MDA terminology 249
9.2 Model transformations 252
9.3 UML profiles 260
9.4 Transformations for internet systems 261
9.5 Implementing model transformations 263
9.6 MDA tools 266
9.7 Summary 267
9.8 Exercises 267

10 Case Studies of Web System Development 269
10.1 Property system specification 269
10.2 Web system generation 271
10.3 Interface components 273
10.4 Servlets 281
10.5 Database 287
10.6 JSP architecture example: Cat database 291
10.7 Summary 296

11 Catalogue of Model Transformations 297
11.1 Quality improvement transformations 297
11.2 Elaborations 310
11.3 Refinements 311
11.4 Design patterns 320

Bibliography 323

Appendices: 325

A Scrabble History and Rules 325
A.1 The history of Scrabble 325
A.2 Official Scrabble rules 325

B Web Application Development Support Package 329

C Using Standard OCL 331
C.1 Why use LOCA? 334

D Exercise Solutions 335
D.1 Solutions: Chapter 2 335
D.2 Solutions: Chapter 3 340
D.3 Solutions: Chapter 4 343
D.4 Solutions: Chapter 5 346
D.5 Solutions: Chapter 6 353
D.6 Solutions: Chapter 7 356
D.7 Solutions: Chapter 8 367
D.8 Solutions: Chapter 9 372

Index 373

Preface

The world of software development is experiencing dramatic growth and diversification, with a multitude of new languages and technologies continually being introduced and elaborated: XML, .Net, web services, mobile computing, etc. It therefore becomes increasingly difficult to keep up to date with even the technologies in one particular area.

At the same time, important steps towards unification and standardisation of notations and methods are taking place – the new UML 2.0 standard is the prime example of these, and provides a common notation and set of concepts which can be used in any object-oriented development for any kind of system. The MDA[1] (Model-driven Architecture) likewise provides a general strategy for separating platform-independent specifications of systems from their platform-specific implementations.

In this book we cover the key languages and techniques for object-oriented software design of a wide range of systems:

- UML 2.0: we introduce the core concepts and notations of UML, and show how they can be used in practice.
- MDA: we describe the key elements of this approach, and transformations of models using UML Profiles, XSLT, JMI and REI.
- Internet system design: how to use JavaScript, Flash, XHTML, JSP and Servlets to produce modular, usable, portable and accessable web systems.
- Web Services: including technologies such as J2EE, JavaMail, .Net, SOAP, WSDL and streaming.
- E-commerce: including the Semantic Web, FTP, WML and Bluetooth.

A catalogue of UML model transformations is also provided. The supporting website contains the UML-RSDS tool to support the MDA process, and the UML2Web tool for the synthesis of web applications from UML descriptions.

Examples of the use of these techniques are given throughout the book, with three large case studies being used:

[1]MDA is a registered trademark of the OMG.

- A system to play Scrabble.
- An internet jukebox using data streaming.
- An online estate-agent system.

Acknowledgements

Kelly Androutsopoulos, David Clark and Pauline Kan contributed to the program synthesis concepts used in the book. Runa Jesmin contributed the material on usability design of web systems. The internet jukebox case study is due to Ruth O'Dowd, Taherah Ansari contributed the case study of XSLT and REI transformations, and numerous other students at King's College have also helped in providing feedback on courses in which the material presented here has been taught.

Chapter 1

The Challenges of Software Design

This chapter surveys current issues in software construction, the problems caused by the pace of technological change, and the need for improved maintainability and portability of software. In particular we consider the need for software development to focus more resources on construction of platform-independent models to reduce the effort in recreating a system for a new platform or technology.

1.1 Software development

The purpose of software[1] remains the same today as it was at the beginning of computing in the 1940s: to automate the solution of complex problems, using computers. However the nature of the problems to be solved has changed dramatically, and so have the programming techniques employed: the first computers were used for highly critical and specialised tasks such as decryption, and 'programming' them meant reconfiguring the hardware (vacuum tubes or 'valves') of these huge and massively expensive devices.

Today, the variety of tasks for which computational power is used spans the whole range of business, social and creative endeavours. Increasingly, instead of performing some isolated computation, software systems form active or passive elements in a communicating network of services and agents, each new system depending essentially on existing capabilities of previously developed systems, whose services it uses.

Programming techniques have also advanced in the decades that followed the 1940s, through the use of languages of increasing power and abstraction: Assembly languages, FORTRAN, C, C++, and now Java and $C^{\#}$. Instead of manipulating instructions at the level of the hardware, programmers specify data and algorithms in terms of the problem domain. The rise of object-orientation as a language for problem description and solution is the prime

[1]The name 'software' for computer programs seems to have been used for the first time by John Tukey in the January 1958 edition of the *American Mathematical Monthly* journal [50].

present-day example of this progression. Object-oriented languages have become the dominant trend in software development, and even archaic languages such as FORTRAN, BASIC and COBOL have been extended with object-oriented facilities.

The software development activities undertaken by companies today have also changed. Instead of building new stand-alone systems, software development is often used to enhance a company's enterprise information capabilities by building new functions into an already existing infrastructure, or by gluing together existing systems to carry out new services. In general, the emphasis in software development is increasingly on systems as *components* in larger systems, components which interact and combine with each other, perhaps in ways that were not envisaged by their original developers.

For example, in the jukebox audio streaming case study of Chapter 8, existing web services specialised for streaming audio data need to be combined with a playlist database, control unit and a device for playing the downloaded data.

Another factor has arisen, of profound significance for the software industry, which is the *pace of change and introduction of new technologies.* Although the rate of progress in the computer industry has always been rapid compared to other fields, since the advent of the internet, the dissemination and uptake of new languages and techniques has reached a level that seems to put any system more than a year old in danger of obsolescence, and requires continual revision in education and training of developers.

For example, it took over a decade for object-orientation to emerge from research and niche application areas to become the pervasive feature of modern programming languages, yet more recently XML [53] has made the same scale of transition in under five years.

The rapid change and introduction of new software technologies and languages, whilst obviously bringing benefits in enhanced capabilities, has also resulted in expense and disruption to companies using software, due to the need to continually upgrade and migrate applications.

The concept of *Model-driven Architecture* (MDA) [28] aims to alleviate this problem by focusing developer effort at higher levels of abstraction, in the creation of *platform-independent models* (PIMs) from which versions of the system appropriate to particular technologies/languages can be generated, semi-automatically, using *platform-specific models* (PSMs). This means that companies can retain the key elements of their software, especially the *business rules* or logical decision-making code, in a form that is independent of changes in technology, and that can be used to generate, at relatively low cost, new implemented systems for particular technologies.

The MDA approach (Figure 1.1) can be seen as a continuation of the trend towards greater abstraction in programming languages. In this case, the 'programming' is intended to take place at the level of diagrammatic UML models and constraints on these models. Executable versions of the system are then generated, as an extension of the compilation process.

This book will describe one approach for making this vision of reusable

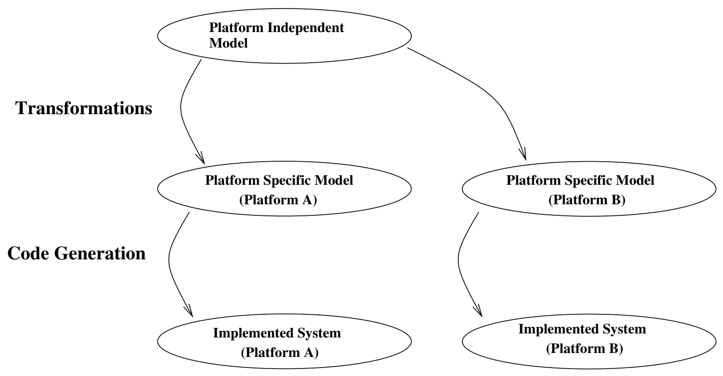
Platform Independent Model
Transformations
Platform Specific Model (Platform A)
Platform Specific Model (Platform B)
Code Generation
Implemented System (Platform A)
Implemented System (Platform B)

Figure 1.1: The MDA process

software a practical reality. In the remainder of the chapter we survey existing software development methods and the key concepts of software development processes, and discuss how these relate to the MDA.

1.2 Software development methods

A *software development method* is a systematic means of organising the process and products of a software construction project. A software development method typically consists of:

- *Notations* or languages to describe the artifacts being produced, for example UML [51] can be used to describe requirements, analysis or design models.
- A *process*, defining the sequence of steps taken to construct and validate artifacts in the method.
- *Tools* to support the method.

Some popular development methods are the *spiral model*, the *waterfall model*, the *rational unified process*, and *extreme programming.*

1.2.1 The Waterfall model

In the waterfall model [41], the stages of development such as requirements definition, design and implementation are separated into complete tasks which must be fully carried out and 'signed off' before the next task can be started (Figure 1.2). Each stage produces a deliverable (eg, a complete requirements specification, complete system design) which is intended to not need much further revision.

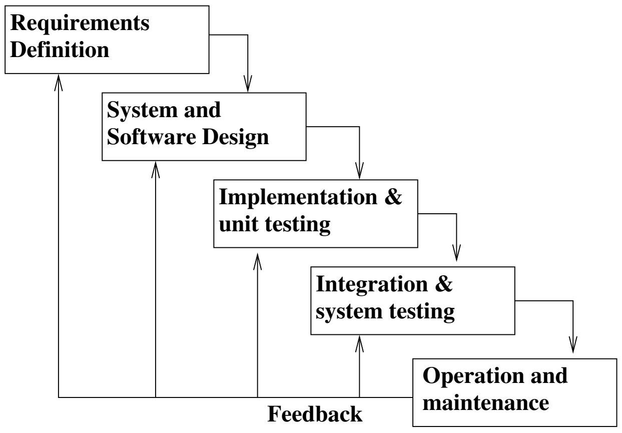
Requirements Definition
System and Software Design
Implementation & unit testing
Integration & system testing
Operation and maintenance
Feedback

Figure 1.2: Waterfall model process

1.2.2 The Spiral model

The spiral model [5] is based on an iteration of incremental development steps (Figure 1.3). Each iteration (cycle of the spiral) produces a deliverable, such as an enhanced set of models of a system, or an enhanced prototype of a system. In each iteration a review stage is carried out to check that the development is progressing correctly and that the deliverables meet their requirements. Unlike the waterfall model, the deliverables produced may be partial and incomplete, and can be refined by further iterations.

1.2.3 The Rational Unified Process (RUP)

The Rational Unified Process [43] defines four phases in the development process:

1. *Inception*: definition of project scope, goals of the system, evaluation of its feasibility and general estimates of project cost and time. A prototype may be built to assist in checking if the project is viable. The main process in this phase is requirements definition, producing initial use cases in agreement with the users.
2. *Elaboration*: requirements analysis and definition of architectural design. The use cases are elaborated and structured, forming a starting point for the design of the overall architecture of the system.
3. *Construction*: design and implementation through the development of prototypes, culminating in the delivery of a beta version of the system to user sites.
4. *Transition*: testing, correction of defects detected by users, rollout of new versions until a production version is reached.

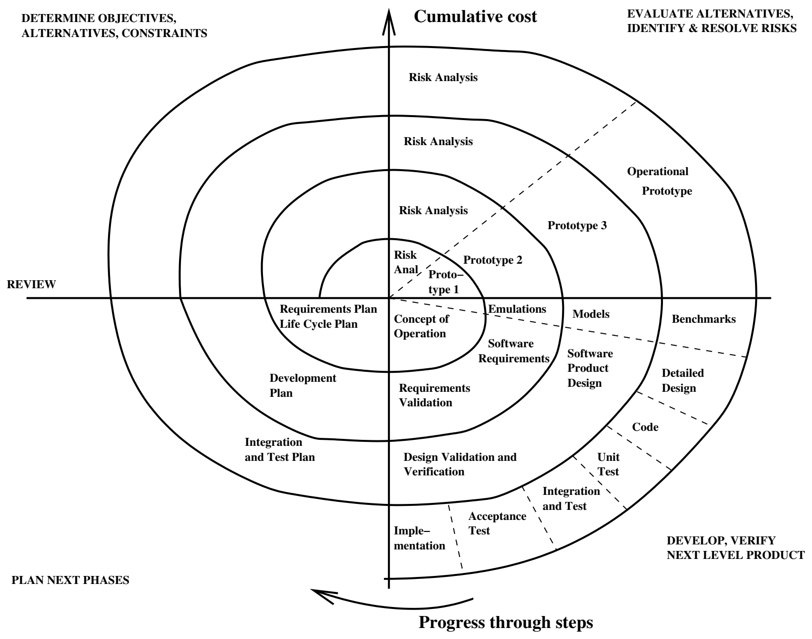

Figure 1.3: Spiral model process

The major milestones in development are the progression from one phase to another: the decision to commit to the project in the *Inception* → *Elaboration* progression, an accepted first revision of the requirements document in the *Elaboration* → *Construction* progression, and a beta release in the *Construction* → *Transition* progression.

The process is *use-case driven*: each design module should identify what use cases it implements, and every use case must be implemented by one or more modules.

1.2.4 Extreme Programming (XP)

Extreme programming tries to minimise the complexity of following a particular development process [4]. Instead of prescribing a rigid set of development steps and milestones, it emphasises a number of *practices*:

- *Realistic planning*: customers make the business decisions about a system, the developers make the technical decisions. Plans are reviewed and revised as necessary.
- *Small releases*: release a useful system quickly, and release updates at frequent intervals.
- *Metaphor*: all programmers should share an understanding of the purpose and global strategy of the system being developed.

- *Simplicity*: design everything to be as simple as possible instead of preparing for future complexity.
- *Testing*: both customers and programmers write tests. The system should be frequently tested.
- *Refactoring*: the system should be restructured whenever necessary to improve the code and eliminate duplication.
- *Pair programming*: put programmers together in pairs, each pair writes code on the same computer.
- *Collective ownership*: all programmers have permission to change any code as necessary.
- *Continuous integration*: whenever a task is completed, build a complete system containing this part and test it.
- *40-hour week*: don't work extreme hours to try to compensate for planning errors.
- *On-site customer*: an actual customer of the system should be available at all times.
- *Coding standards*: follow standards for self-documenting code.

XP is a lightweight development approach which aims to avoid the time-consuming documentation and structures of other methods. Code can be written immediately after definition of use cases (together with test plans for each use case). The code produced can then be refactored and restructured to produce a more efficient and maintainable implementation.

1.2.5 Which method to choose?

The waterfall model has often been criticised for its rigid progression from one task to the next, which can lead to a progressive build-up of delays, as one task cannot begin until its predecessor has completed. On the other hand, it imposes greater discipline than more evolutionary incremental approaches, where 'completion' of a task is left open-ended, and indeed, may not end.

The cumulative iterations and reviews of the spiral model can lead to earlier detections of errors, and therefore lower costs and reduced risks of failure than monolithic methods such as the waterfall model.

The Rational Unified Process, like the waterfall model, also has an emphasis on development phases with milestones as the progressions between them. Some software engineers believe that, in contrast, an *architecture-driven* approach is necessary for object-oriented development, whereby global phases are replaced as the focus by the modular construction of systems, component by component and layer by layer. For each layer there are separate specification, design and implementation activities. XP may be a better fit for this approach than phase-centred methods.

In principle, any of the above development methods can be used in a MDA development, although the end product of the process may be a PIM or a PIM plus PSMs, instead of executable code. With the MDA, the progression from specification to design, and from design to implementation, may be partly automated. This enables developer effort to be concentrated on producing high-quality and correct PIMs. Refactoring transformations and design patterns are of particular importance in this respect, and we describe how these can be used for PIM to PIM mapping and PIM to PSM mapping in Chapter 5.

1.3 Software development steps

A number of stages are typically present in any software development project, regardless of the development model and methods chosen. We will illustrate these stages for two different projects: (i) a system to play Scrabble against human players, and (ii) a system to stream music from a server to a home entertainment system.

1.3.1 Feasibility analysis

This stage evaluates the technical options for implementing the system, determining if it is feasible to implement it, and if so, if there is a cost-effective implementation. Background research may be needed to investigate similar systems that already exist, and what services are available to carry out parts of the required functionality. Trial implementation/prototyping could also be carried out, or mathematical estimation (eg, of bandwidth or data storage requirements).

This stage can be carried out as part of the risk analysis phase in the earliest iteration of the spiral model, or as the first step in the waterfall model. In the spiral model a developer could recheck the feasibility of the system after each iteration as part of the review task.

For example, in the case of the Scrabble playing system, this stage would investigate the rules of Scrabble, existing AI techniques and programs such as Maven [46] and the feasibility of different memory storage and lookup schemes for dictionaries.

For the Jukebox project, this stage would involve investigating the state of available music streaming technologies, for example, streaming servers such as Helix (http://www.realnetworks.com/) and checking that these support the required functionality of the system.

1.3.2 Requirements definition

Provided that it has been determined that there is some feasible and cost-effective way of constructing the system, the development process can advance to elicit in detail the required functionality of the system.

This stage systematically records the requirements that the customer(s) of the system have for it, and the constraints imposed on the system by existing systems that it is required to operate with, including existing work practices.

For the Scrabble system, the list of requirements could include:

1. The system must enable between one and three human players to play, together with the computer player.
2. Standard Scrabble rules for moves and the validity of moves should be enforced.
3. The system should check all words formed in each move, and only accept the move if all the words appear in its dictionary.
4. The system should keep a history of all valid moves made in a game: the letters played, the player who moved, and the score of the move.

For the Jukebox, the requirements could include:

1. The server should be capable of storing 100 tracks.
2. The user can add a track to the server from a CD, by inserting the CD into their computer and following on-screen instructions from the system.
3. The user interface can be on a separate device to the server. The UI provides an index of tracks, a way to select tracks and playback control (play, stop, fast forward, rewind).

Use-case models in UML can be drawn for either system, to make clear what users or external systems are involved in each functional requirement. Figure 1.4 shows some use cases for the Scrabble system, and Figure 1.5 those for the jukebox.

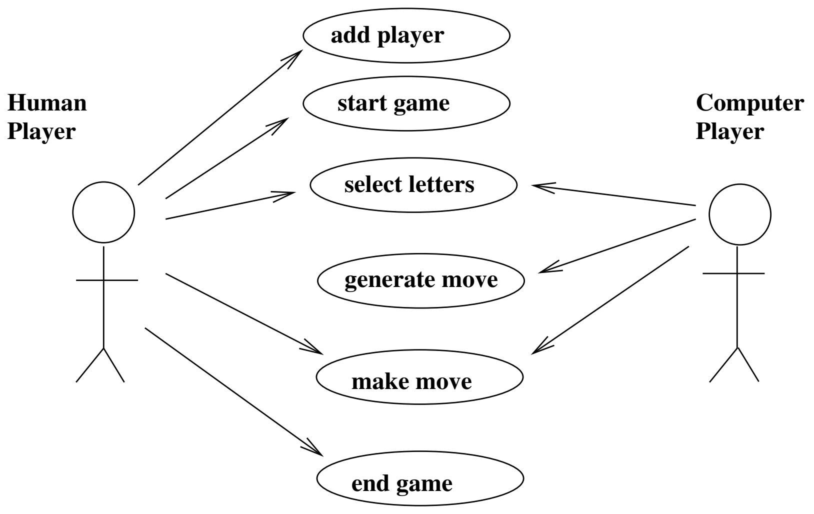

Figure 1.4: Use cases for Scrabble player

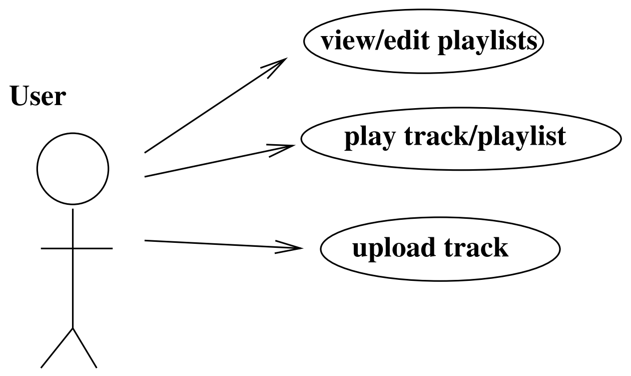

Figure 1.5: Use cases for internet jukebox

1.3.3 Analysis

This stage builds precise models of the system, using suitable graphical or mathematical notations to represent its state and behaviour in an abstract, platform-independent manner. In a critical system, such as a railway signalling system, formal validation that the analysis models satisfy required properties would be carried out.

For the Scrabble system a fragment of the class diagram is shown in Figure 1.6. A fundamental property is that the total number of letters in the game is always 100, these may be distributed between the player's racks, the bag, or the board. Hence the annotation in the top corner of the *Letter* class.

For the Jukebox, the core data model is much simpler (Figure 1.7). There will also be other web forms associated with the system, for logging in, viewing and creating playlists, etc.

1.3.4 Design

This stage defines an architecture and structure for the implementation of the system, typically dividing it into subsystems and modules which have responsibility for carrying out specific parts of its functionality and managing parts of its data. The activities in design include:

1. *Architectural design*: define the global architecture of the system, as a set of major subsystems, and indicate the dependencies between them.
   For example, partitioning a system into a GUI, functional core, and data repository. The GUI depends on the core because it invokes operations of the core, and the core may depend on the GUI (if it invokes GUI operations to display the result of computations, for example). The core depends on the repository. There may be no other connections. Such an architecture is termed a *three tier architecture*. Figure 1.8 shows such a design architecture for the Scrabble system.

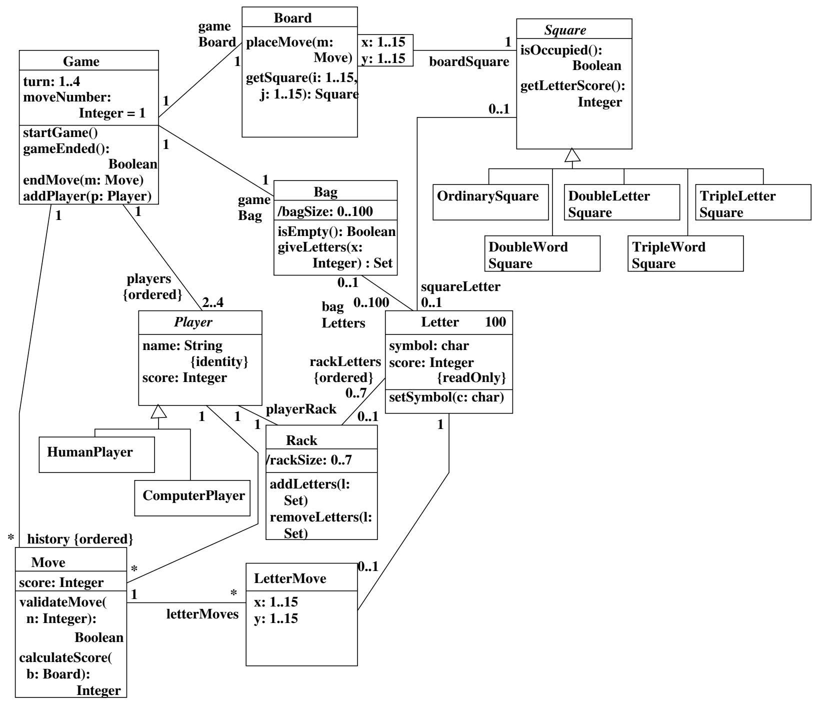

Figure 1.6: Extract from analysis model of Scrabble player

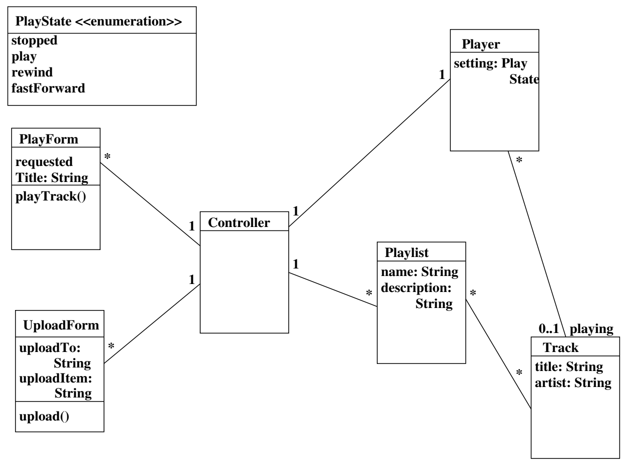

Figure 1.7: Analysis model of jukebox

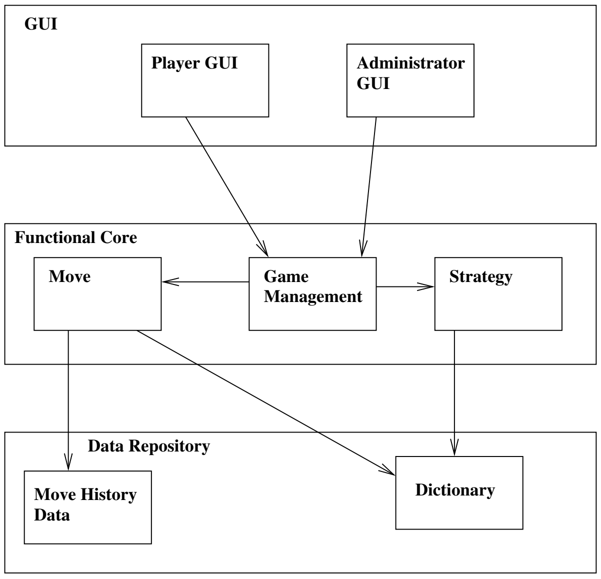

Figure 1.8: Architecture of Scrabble system

MDA may be applied only to the functional core of a system, which contains the business rules of the system, or to all tiers, provided suitable models of these tiers can be defined.

2. *Subsystem design*: decomposition of these global subsystems into smaller subsystems which each handle some well-defined subset of its responsibilities. This process continues until clearly identified *modules* emerge at the bottom of the subsystem hierarchy. A module typically consists of a single entity or group of closely related entities, and operations on instances of these entities.
3. *Module design*: define each of the modules, in terms of:

   (a) the data it encapsulates – eg: a list of attributes and their types or structures;

   (b) the properties (invariants or constraints) it is responsible for maintaining (ie, for ensuring that they are true whenever an operation is not in progress);

   (c) the operations it provides (external services) – eg: their names, input and output data, and specifications. This is called the *interface* of the module.

4. *Detailed design*: for each operation of the module, identify the steps of its processing.

Specialised forms of design include *interface design*, to plan the appearance and sequencing of dialogs and other graphical interface elements for user in-