Software Engineering

Session 6 – Main Theme
Detailed-Level Analysis and Design

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Presentation material partially based on textbook slides
Software Engineering: A Practitioner's Approach (7/e)
by Roger S. Pressman
Slides copyright © 1996, 2001, 2005, 2009

Agenda

1 Introduction

2 Requirements Analysis

3 Requirements Modeling

4 Design Concepts

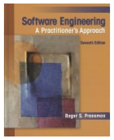
5 Sample Analysis and Design Exercise Using UML

6 Summary and Conclusion

What is the class about?

- Course description and syllabus:
  - » http://www.nyu.edu/classes/jcf/g22.2440-001/
  - » http://www.cs.nyu.edu/courses/spring13/G22.2440-001/

- Textbooks:
  - » ***Software Engineering: A Practitioner's Approach***



Roger S. Pressman
McGraw-Hill Higher International
ISBN-10: 0-0712-6782-4, ISBN-13: 978-00711267823, 7th Edition (04/09)

  - » http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0/
  - » http://highered.mcgraw-hill.com/sites/0073375977/information_center_view0/table_of_contents.html

Detailed-Level Analysis and Design in Brief

- Introduction
- Requirements Analysis
- Requirements Modeling
- Design Concepts
- Sample Analysis and Design Exercise (Using UML 1.x)
- Summary and Conclusion
  - Readings
  - Individual Assignment #2 (assigned)
  - Team Assignment #1 (ongoing)
  - Course Project (ongoing)

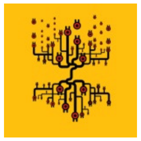
Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance



Alignment

Solution Approach

Agenda

1 **Introduction**

2 **Requirements Analysis**

3 **Requirements Modeling**

4 **Design Concepts**

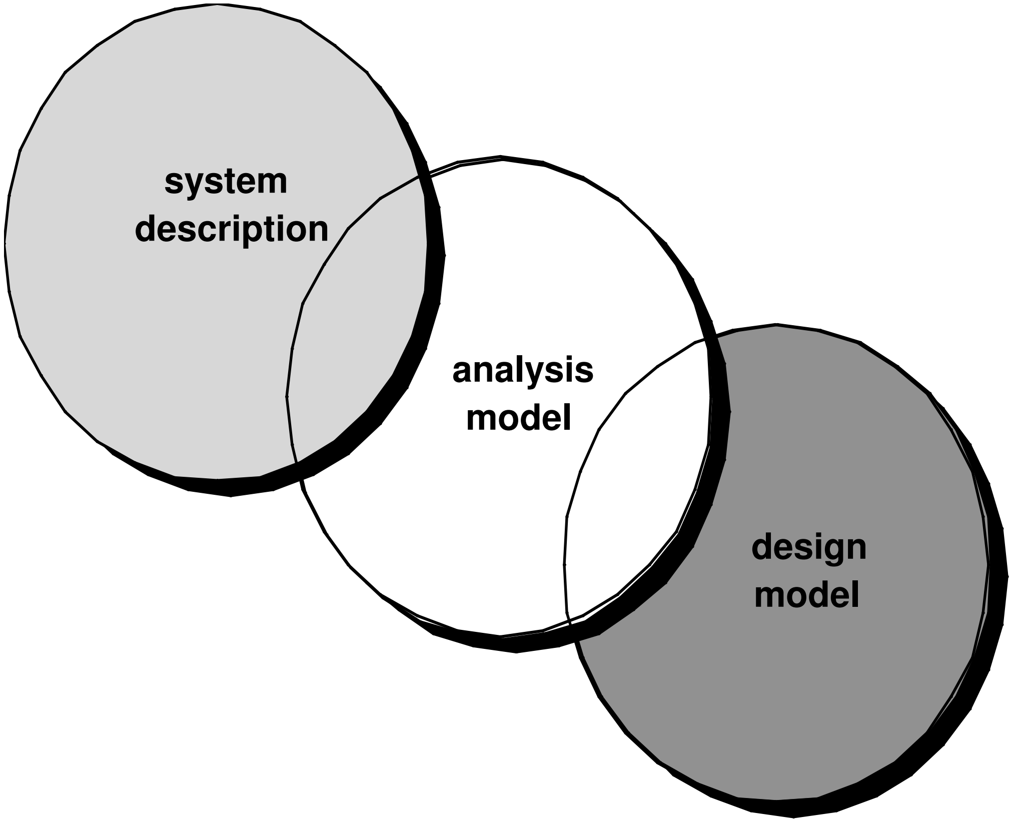
5 **Sample Analysis and Design Exercise Using UML**

6 **Summary and Conclusion**

Requirements Analysis

- Requirements analysis
  - specifies software’s operational characteristics
  - indicates software's interface with other system elements
  - establishes constraints that software must meet
- Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
  - elaborate on basic requirements established during earlier requirement engineering tasks
  - build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

A Bridge



Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain that the analysis model provides value to all stakeholders.
- Keep the model as simple as it can be.

Domain Analysis

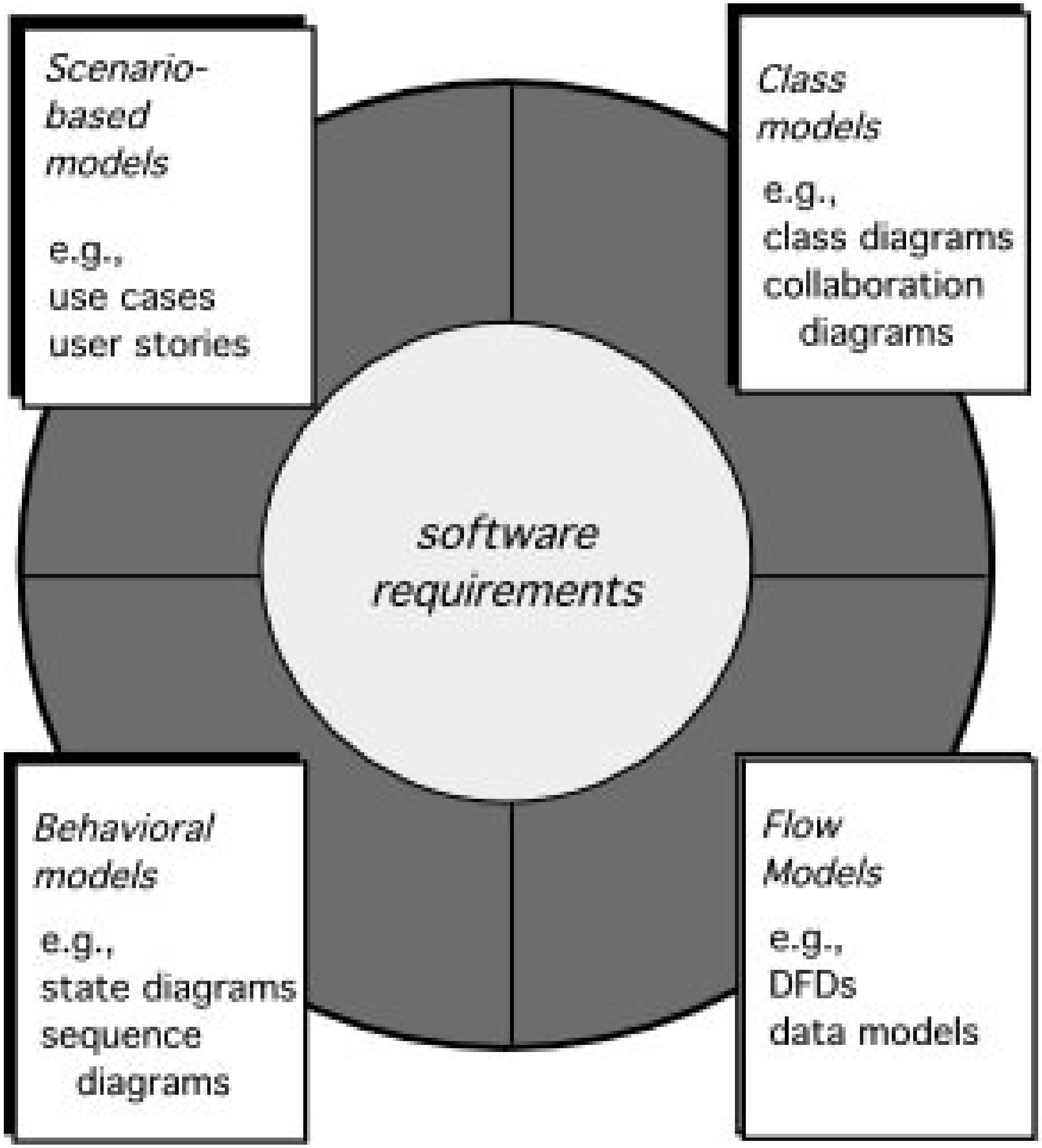
Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . .

Donald Firesmith

Domain Analysis

- Define the domain to be investigated.
- Collect a representative sample of applications in the domain.
- Analyze each application in the sample.
- Develop an analysis model for the objects.

Elements of Requirements Analysis



Scenario-based models

e.g.,
use cases
user stories

Class models

e.g.,
class diagrams
collaboration diagrams

software requirements

Behavioral models

e.g.,
state diagrams
sequence diagrams

Flow Models

e.g.,
DFDs
data models

Scenario-Based Modeling

“[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases).” Ivar Jacobson

(1) What should we write about?

(2) How much should we write about it?

(3) How detailed should we make our description?

(4) How should we organize the description?

What to Write About?

- Inception and elicitation—provide you with the information you'll need to begin writing use cases.
- Requirements gathering meetings, QFD, and other requirements engineering mechanisms are used to
  - identify stakeholders
  - define the scope of the problem
  - specify overall operational goals
  - establish priorities
  - outline all known functional requirements, and
  - describe the things (objects) that will be manipulated by the system.
- To begin developing a set of use cases, list the functions or activities performed by a specific actor.

How Much to Write About?

- As further conversations with the stakeholders progress, the requirements gathering team develops use cases for each of the functions noted.
- In general, use cases are written first in an informal narrative fashion.
- If more formality is required, the same use case is rewritten using a structured format similar to the one proposed.

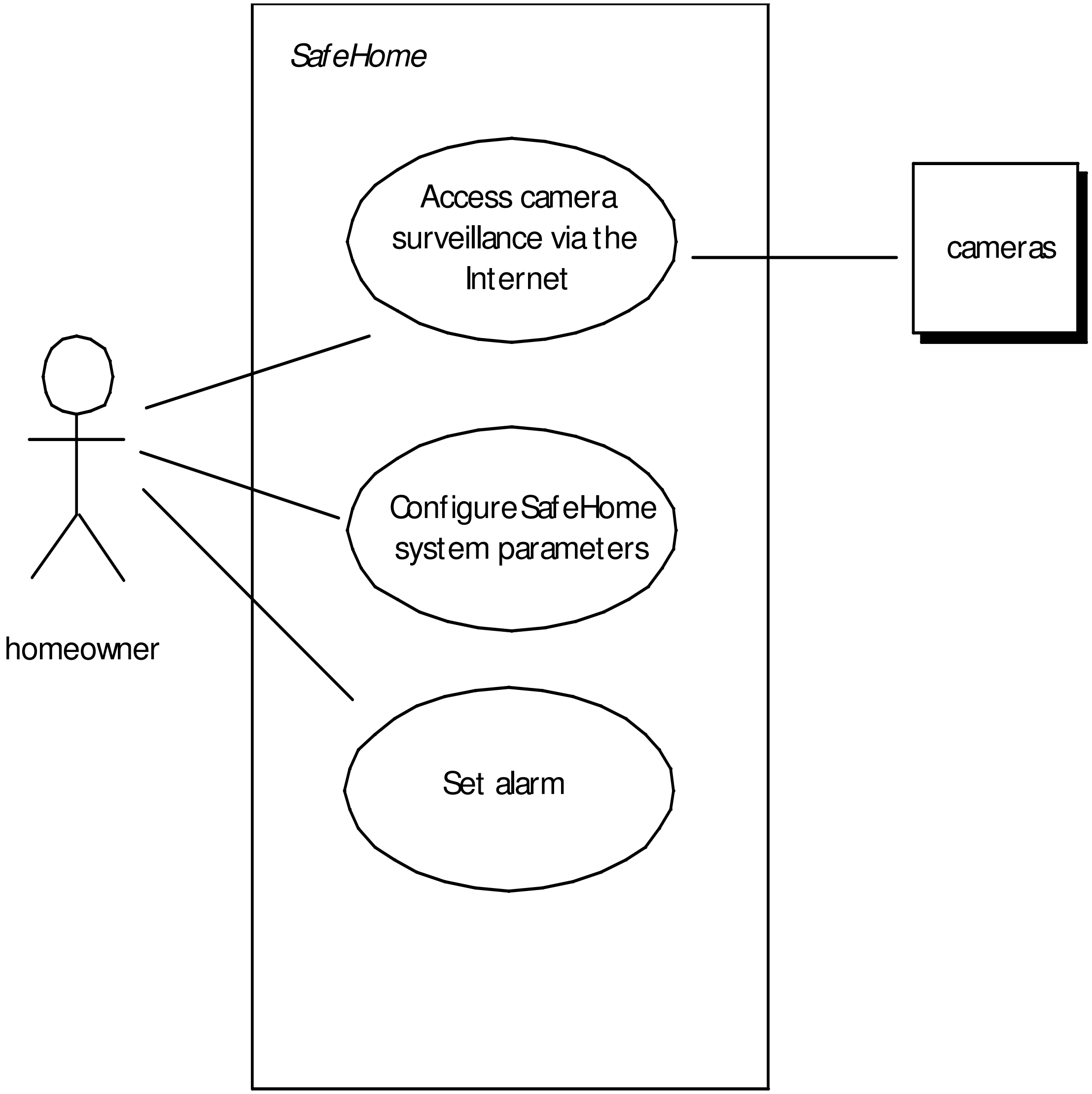
Use-Cases

- a scenario that describes a “thread of usage” for a system
- *actors* represent roles people or devices play as the system functions
- *users* can play a number of different roles for a given scenario

Developing a Use-Case

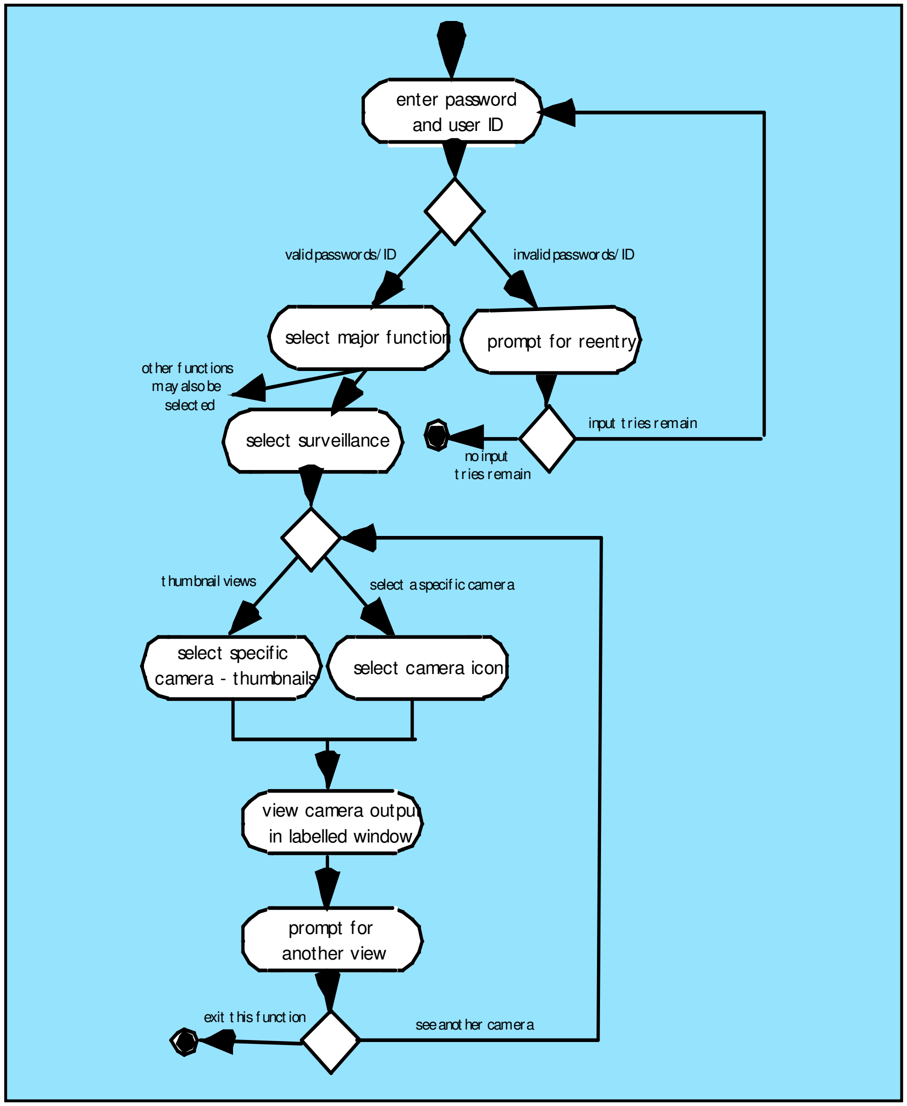
- What are the main tasks or functions that are performed by the actor?
- What system information will the the actor acquire, produce or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Use-Case Diagram


SafeHome
Access camera surveillance via the Internet
cameras
Configure SafeHome system parameters
homeowner
Set alarm

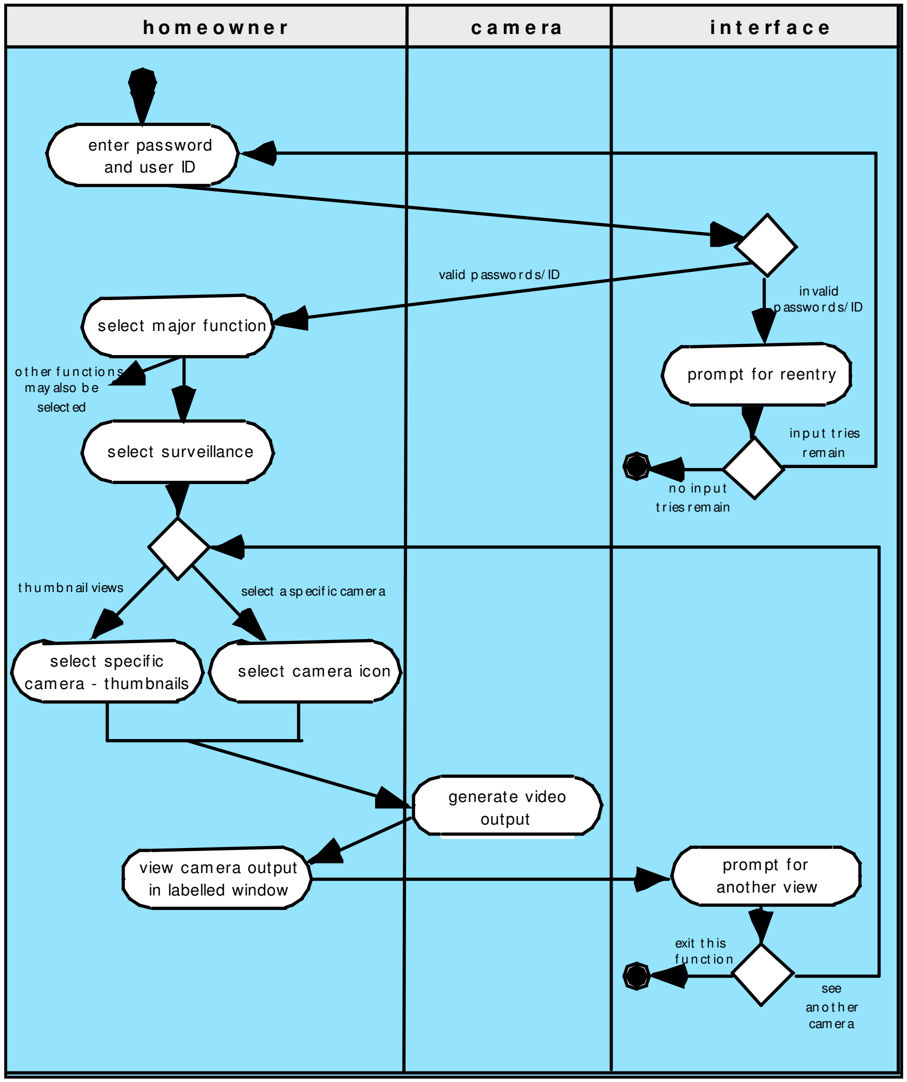
Activity Diagram

Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario



Swimlane Diagrams

Allows the modeler to represent the flow of activities described by the use-case and at the same time indicate which actor (if there are multiple actors involved in a specific use-case) or analysis class has responsibility for the action described by an activity rectangle



Data Modeling

- examines data objects independently of processing
- focuses attention on the data domain
- creates a model at the customer’s level of abstraction
- indicates how data objects relate to one another

What is a Data Object?

- a representation of almost any composite information that must be understood by software.
  - *composite information*—something that has a number of different properties or attributes
- can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only—there is no reference within a data object to operations that act on the data.

Data Objects and Attributes

A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object

object: automobile

attributes:

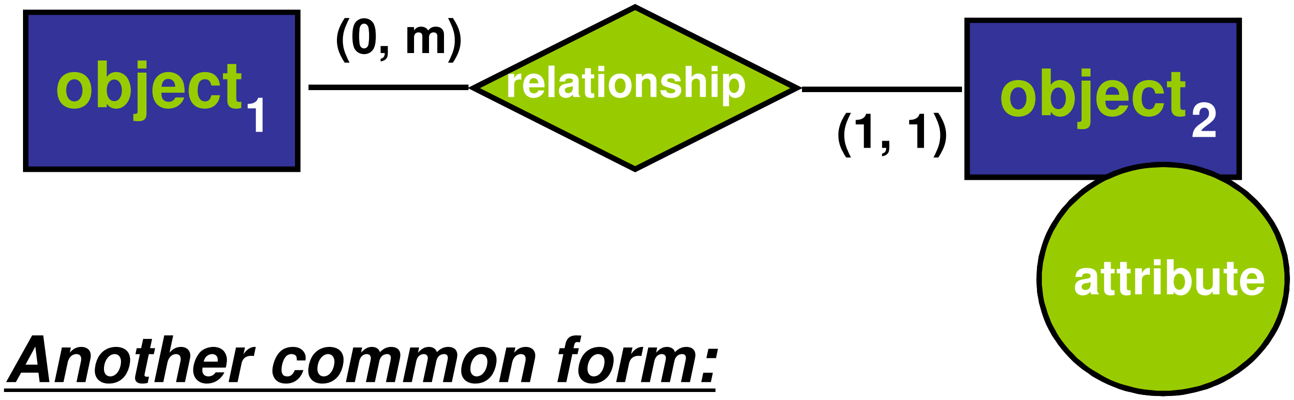
- make
- model
- body type
- price
- options code

What is a Relationship?

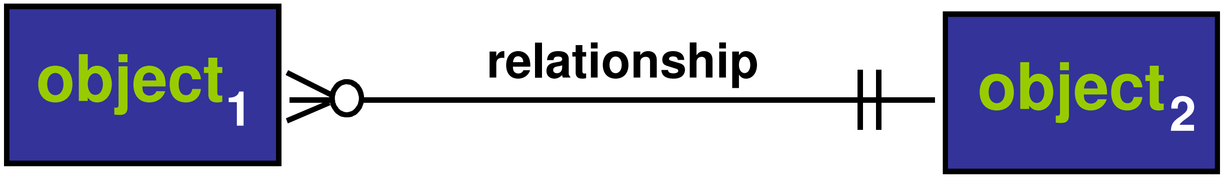
- Data objects are connected to one another in different ways.
  - » A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - A person *is insured to drive* a car
- The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car**.
- Several instances of a relationship can exist
- Objects can be related in many different ways

ERD Notation

One common form:



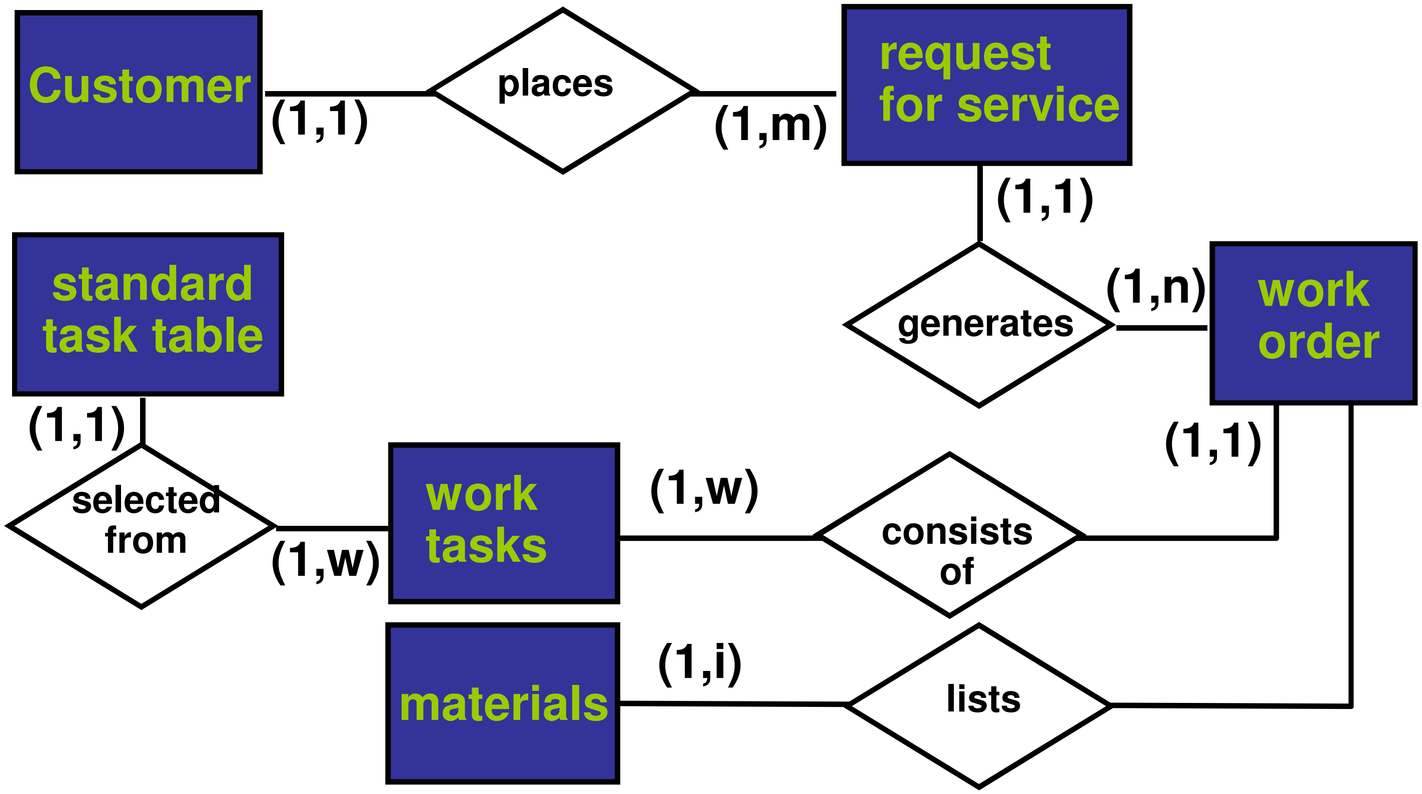
Another common form:



Building an ERD

- *Level 1*—model all data objects (entities) and their “connections” to one another
- *Level 2*—model all entities and relationships
- *Level 3*—model all entities, relationships, and the attributes that provide further depth

The ERD: An Example



Class-Based Modeling

- Class-based modeling represents:
  - » objects that the system will manipulate
  - » operations (also called methods or services) that will be applied to the objects to effect the manipulation
  - » relationships (some hierarchical) between the objects
  - » collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, CRC models, collaboration diagrams and packages.

Identifying Analysis Classes

- Examining the usage scenarios developed as part of the requirements model and perform a "grammatical parse" [Abb83]
  - » Classes are determined by underlining each noun or noun phrase and entering it into a simple table.
  - » Synonyms should be noted.
  - » If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.
- But what should we look for once all of the nouns have been isolated?

Manifestations of Analysis Classes

- *Analysis classes* manifest themselves in one of the following ways:
  - *External entities* (e.g., other systems, devices, people) that produce or consume information
  - *Things* (e.g, reports, displays, letters, signals) that are part of the information domain for the problem
  - *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation
  - *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system
  - *Organizational units* (e.g., division, group, team) that are relevant to an application
  - *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function
  - *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects

Potential Classes

- *Retained information.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
- *Needed services.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
- *Multiple attributes.* During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
- *Common attributes.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
- *Common operations.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
- *Essential requirements.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

Defining Attributes

- *Attributes* describe a class that has been selected for inclusion in the analysis model.
  - » build two different classes for professional baseball players
    - **For Playing Statistics software:** name, position, batting average, fielding percentage, years played, and games played might be relevant
    - **For Pension Fund software:** average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.

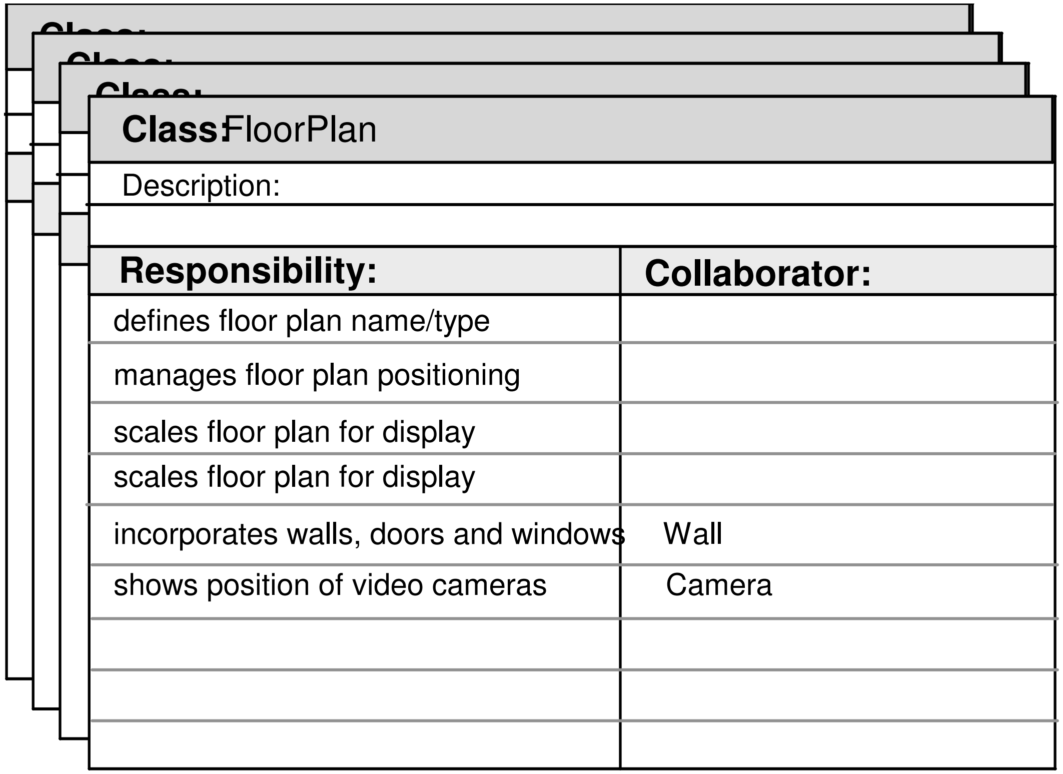
Defining Operations

- Do a grammatical parse of a processing narrative and look at the verbs
- Operations can be divided into four broad categories:
  - » (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting)
  - » (2) operations that perform a computation
  - » (3) operations that inquire about the state of an object, and
  - » (4) operations that monitor an object for the occurrence of a controlling event.

CRC Models

- *Class-responsibility-collaborator (CRC) modeling* [Wir90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [Amb95] describes CRC modeling in the following way:
  - A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

CRC Modeling



Class: FloorPlan

Description:

Responsibility:	**Collaborator:**
defines floor plan name/type	
manages floor plan positioning	
scales floor plan for display	
scales floor plan for display	
incorporates walls, doors and windows	Wall
shows position of video cameras	Camera

Class Types

- *Entity classes*, also called *model* or *business classes*, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- *Controller classes* manage a “unit of work” [UML03] from start to finish. That is, controller classes can be designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

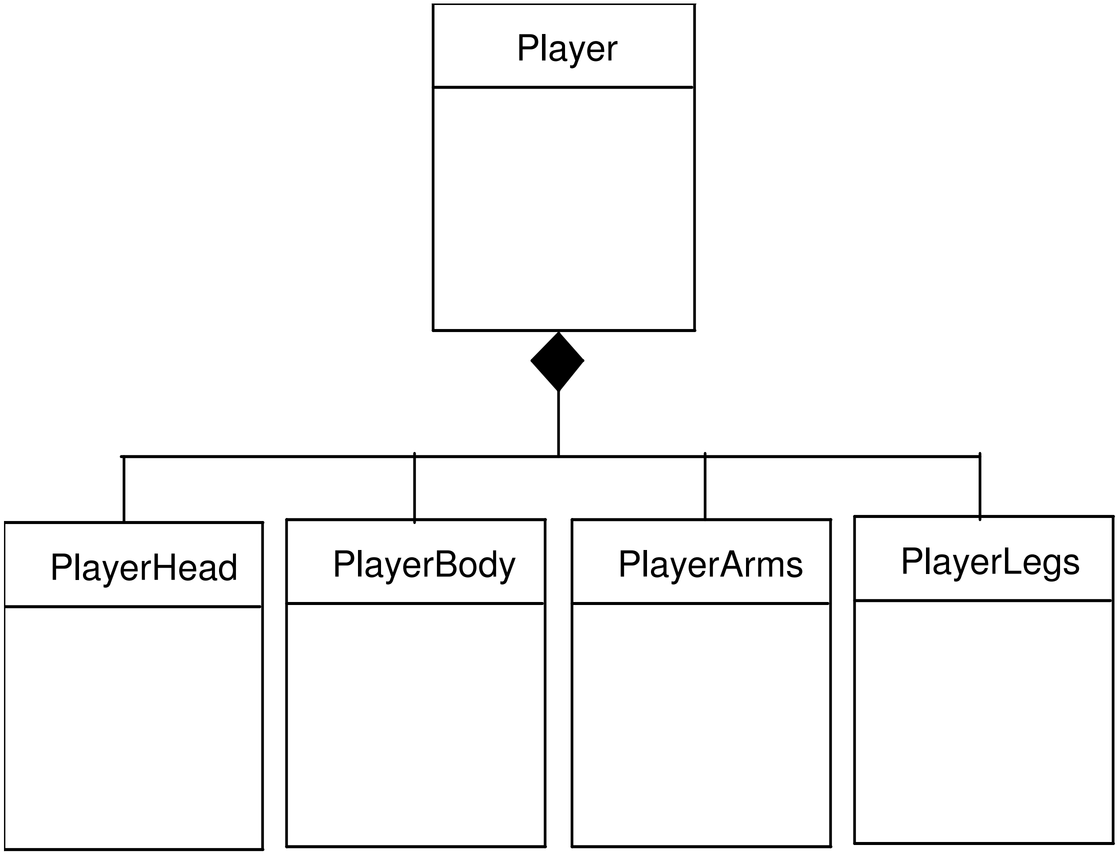
Responsibilities

- System intelligence should be distributed across classes to best address the needs of the problem
- Each responsibility should be stated as generally as possible
- Information and the behavior related to it should reside within the same class
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.

Collaborations

- Classes fulfill their responsibilities in one of two ways:
  - » A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - » a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes [WIR90]:
  - » the *is-part-of* relationship
  - » the *has-knowledge-of* relationship
  - » the *depends-upon* relationship

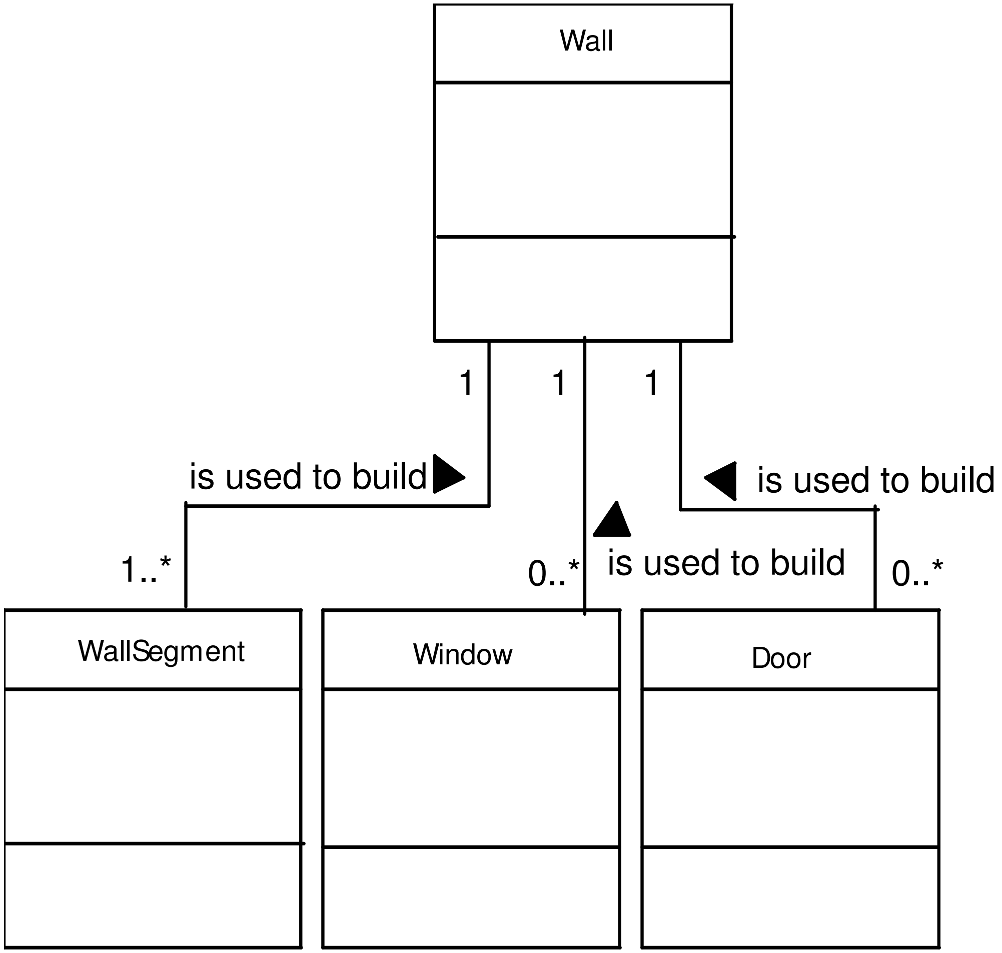
Composite Aggregate Class



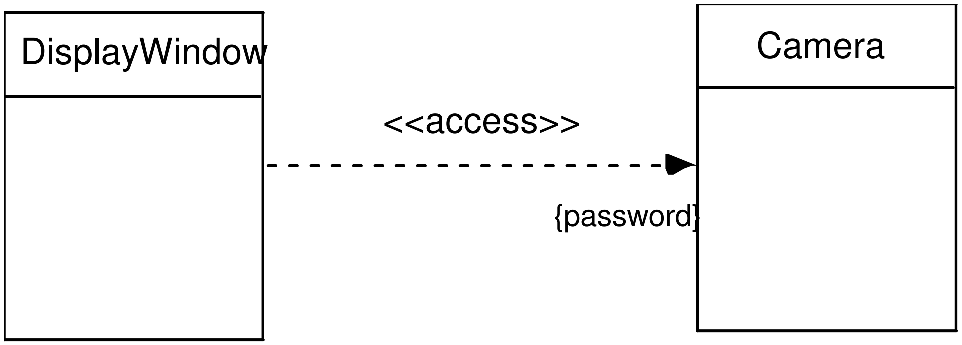
Associations and Dependencies

- Two analysis classes are often related to one another in some fashion
  - In UML these relationships are called *associations*
  - Associations can be refined by indicating *multiplicity* (the term *cardinality* is used in data modeling
- In many instances, a client-server relationship exists between two analysis classes.
  - In such cases, a client-class depends on the server-class in some way and a *dependency relationship* is established

Multiplicity



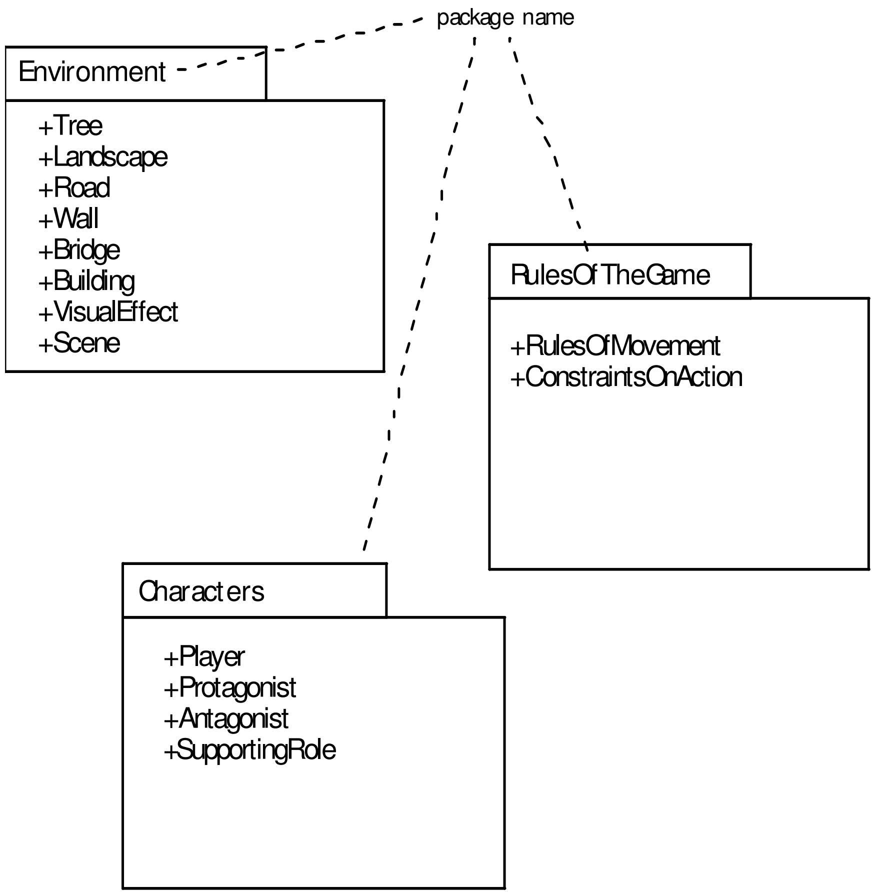
Dependencies



Analysis Packages

- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner that packages them as a grouping
- The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- Other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

Analysis Packages



Reviewing the CRC Model

- All participants in the review (of the CRC model) are given a subset of the CRC model index cards.
  - » Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
- All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
- The review leader reads the use-case deliberately.
  - » As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
- When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card.
  - » The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
- If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards.
  - » This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

Agenda

1 **Introduction**

2 **Requirements Analysis**

3 **Requirements Modeling**

4 **Design Concepts**

5 **Sample Analysis and Design Exercise Using UML**

6 **Summary and Conclusion**

Requirements Modeling Strategies

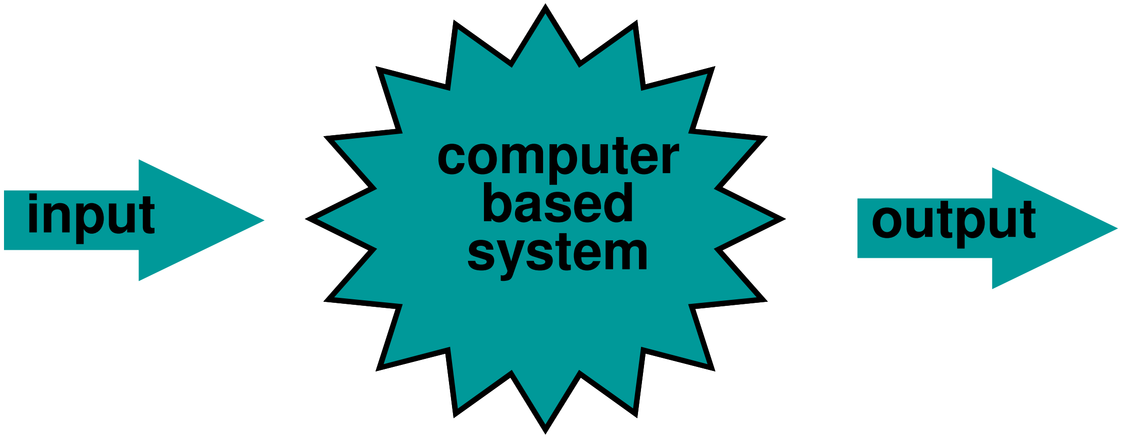
- One view of requirements modeling, called *structured analysis,* considers data and the processes that transform the data as separate entities.
  - » Data objects are modeled in a way that defines their attributes and relationships.
  - » Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
- A second approach to analysis modeled, called *object-oriented analysis,* focuses on
  - » the definition of classes and
  - » the manner in which they collaborate with one another to effect customer requirements.

Flow-Oriented Modeling

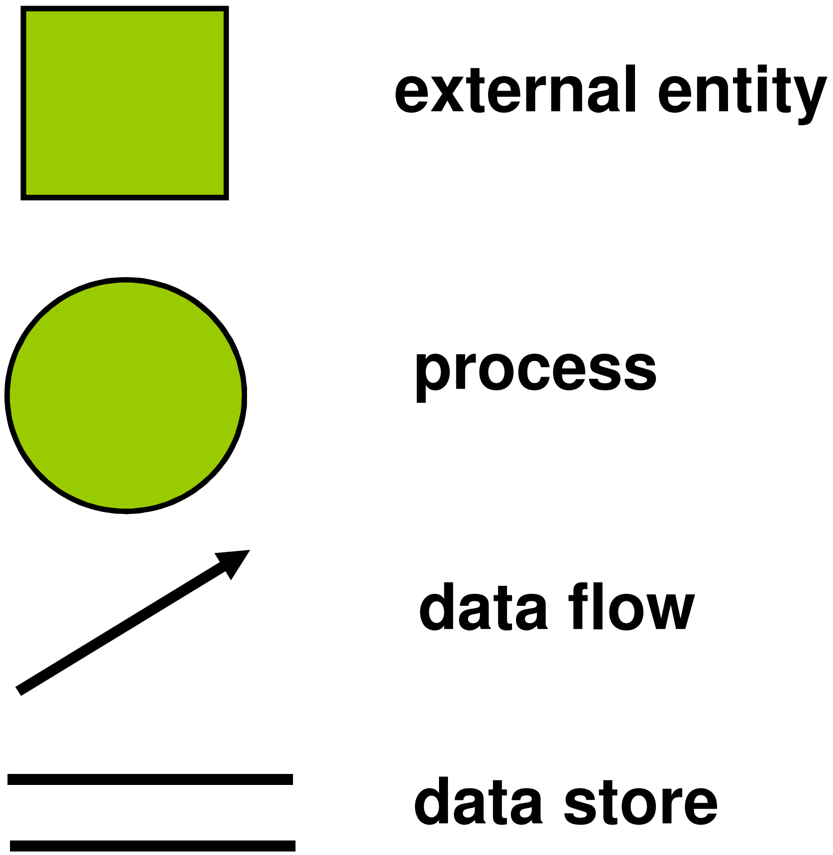
- Represents how data objects are transformed at they move through the system
- **data flow diagram (DFD)** is the diagrammatic form that is used
- Considered by many to be an "old school" approach, but continues to provide a view of the system that is unique—it should be used to supplement other analysis model elements

The Flow Model

Every computer-based system is an information transform ....



Flow Modeling Notation



external entity

process

data flow

data store

External Entity

A producer or consumer of data

Examples: a person, a device, a sensor

Another example: computer-based system

Data must always originate somewhere and must always be sent to something

Process

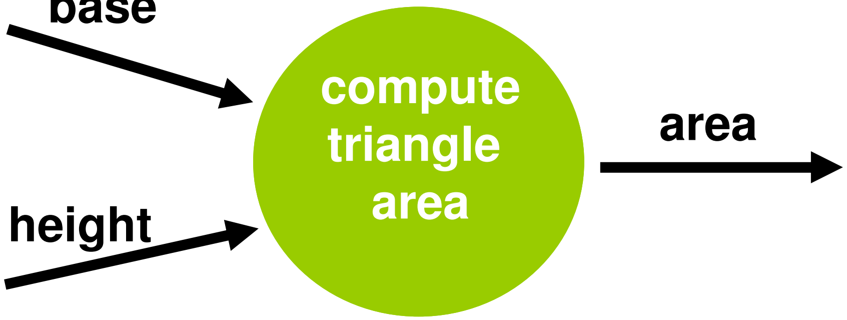
A data transformer (changes input to output)

Examples: *compute taxes, determine area, format report, display graph*

Data must always be processed in some way to achieve system function

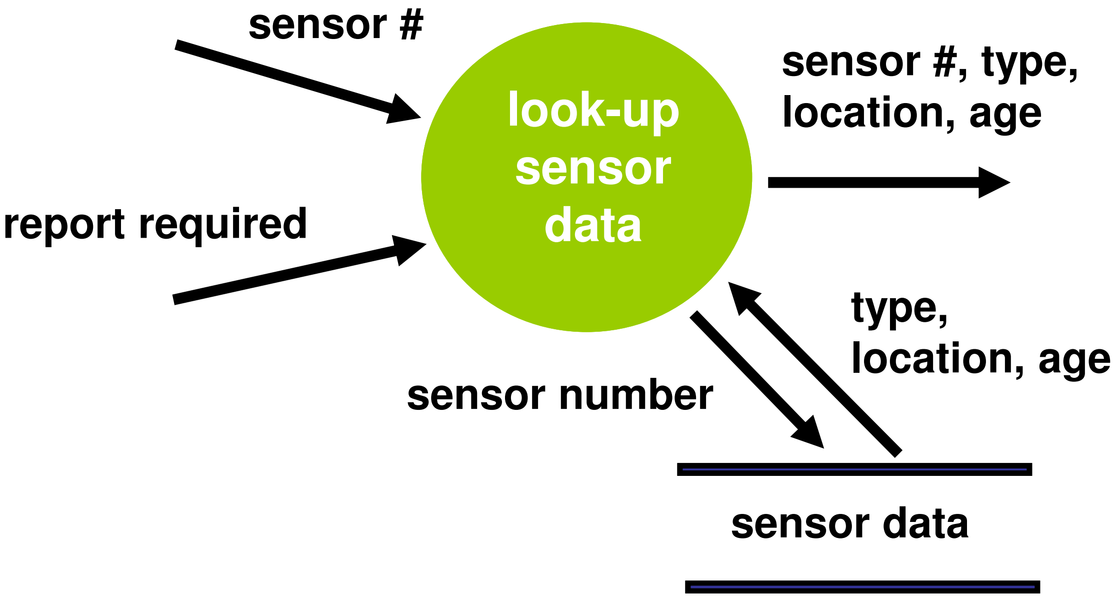
Data Flow

Data flows through a system, beginning as input and transformed into output.



Data Stores

Data is often stored for later use.



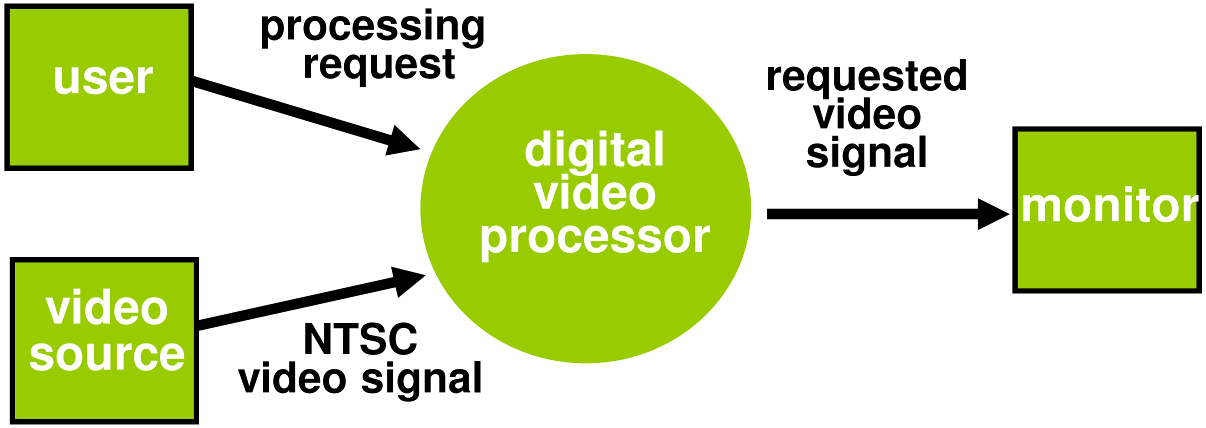
Data Flow Diagramming: Guidelines

- all icons must be labeled with meaningful names
- the DFD evolves through a number of levels of detail
- always begin with a context level diagram (also called level 0)
- always show external entities at level 0
- always label data flow arrows
- do not represent procedural logic

Constructing a DFD—I

- review user scenarios and/or the data model to isolate data objects and use a grammatical parse to determine “operations”
- determine external entities (producers and consumers of data)
- create a level 0 DFD

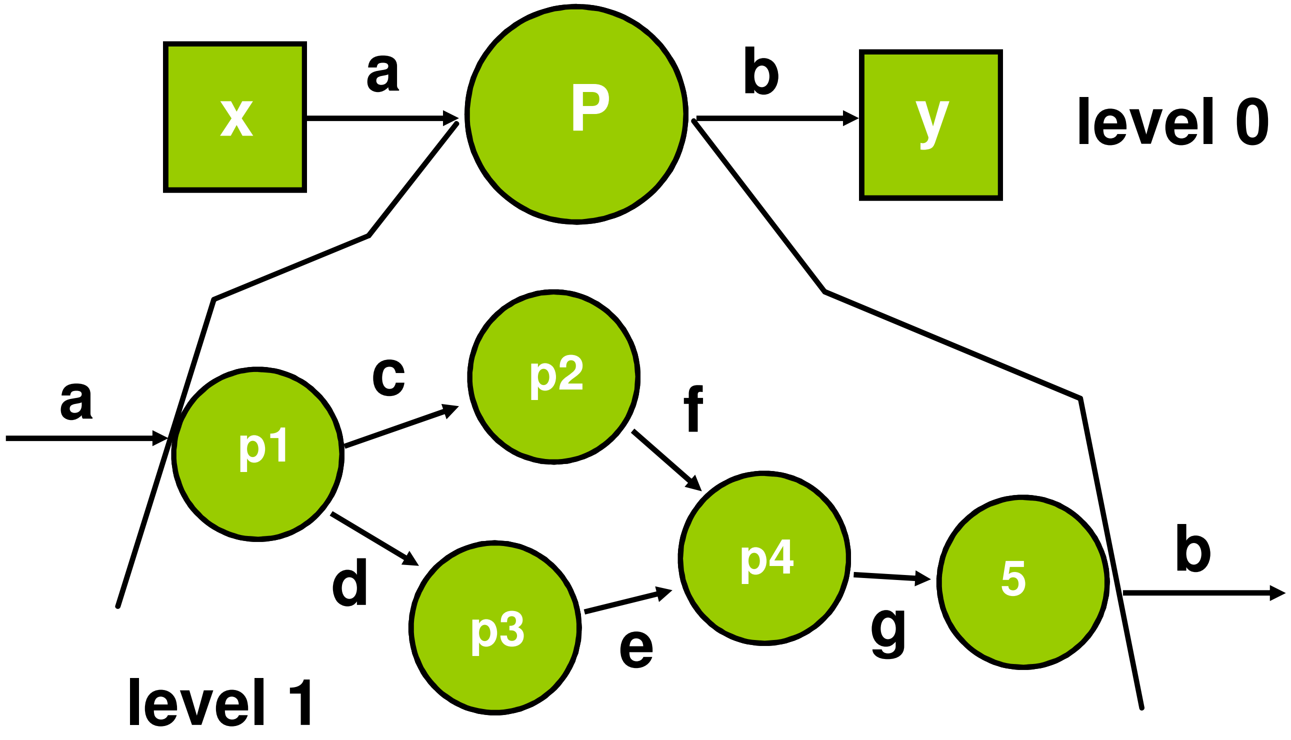
Level 0 DFD Example



Constructing a DFD—II

- write a narrative describing the transform
- parse to determine next level transforms
- “balance” the flow to maintain data flow continuity
- develop a level 1 DFD
- use a 1:5 (approx.) expansion ratio

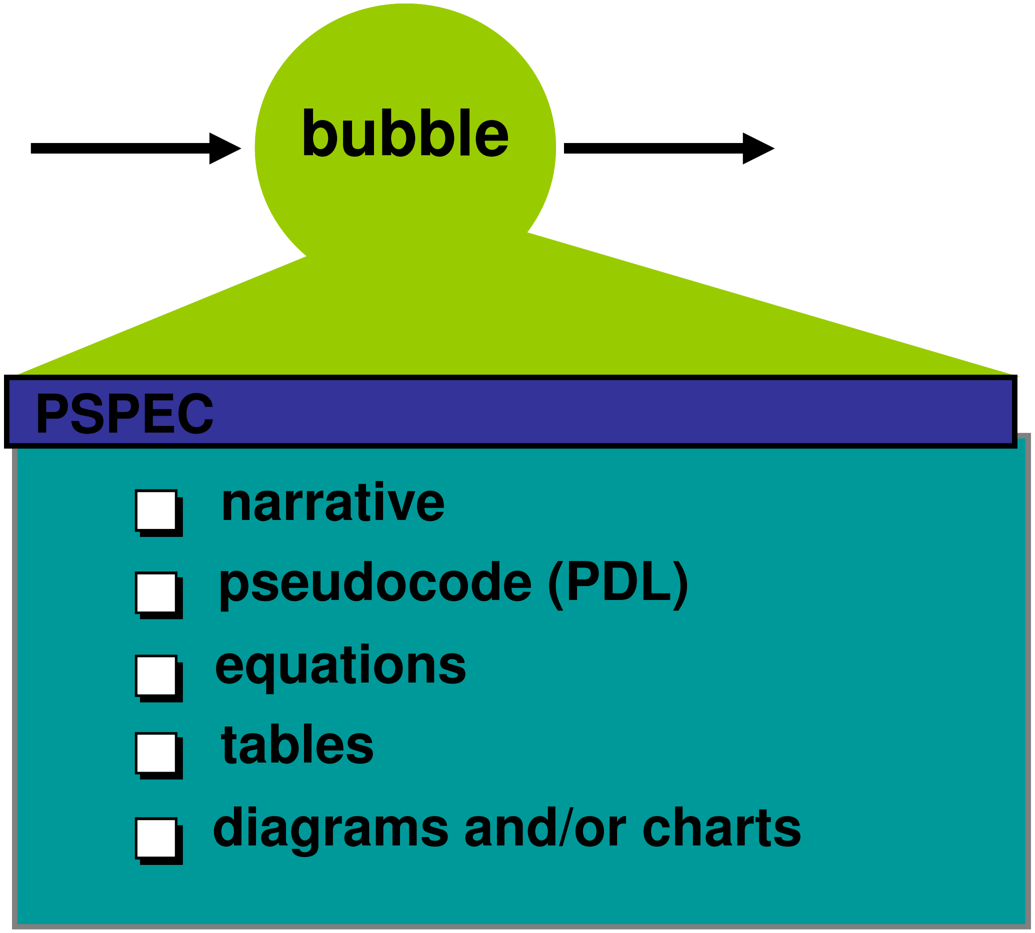
The Data Flow Hierarchy



Flow Modeling Notes

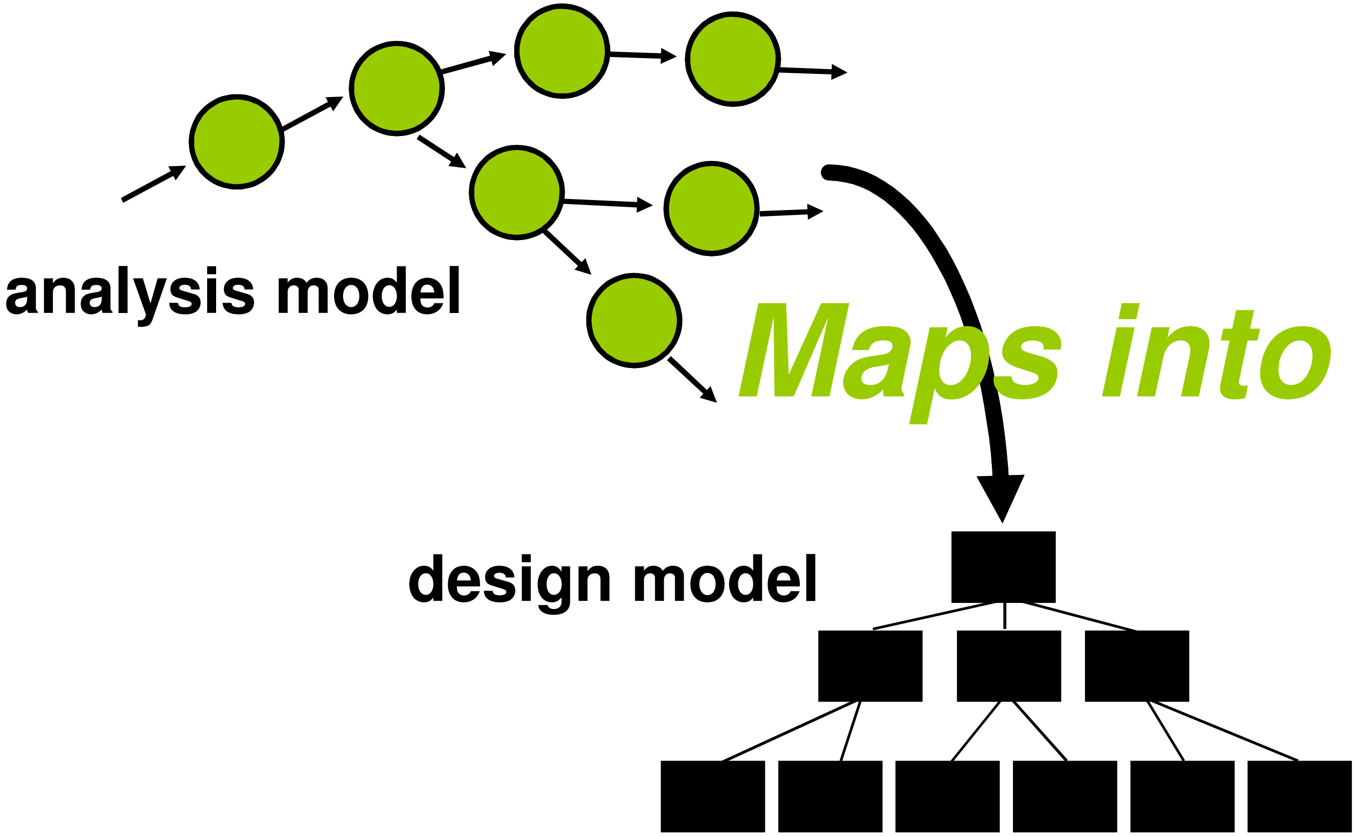
- each bubble is refined until it does just one thing
- the expansion ratio decreases as the number of levels increase
- most systems require between 3 and 7 levels for an adequate flow model
- a single data flow item (arrow) may be expanded as levels increase (data dictionary provides information)

Process Specification (PSPEC)



- □ narrative
- □ pseudocode (PDL)
- □ equations
- □ tables
- □ diagrams and/or charts

DFDs: A Look Ahead



Control Flow Modeling

- Represents “events” and the processes that manage events
- An “event” is a Boolean condition that can be ascertained by:
  - listing all sensors that are "read" by the software.
  - listing all interrupt conditions.
  - listing all "switches" that are actuated by an operator.
  - listing all data conditions.
  - recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.

Control Specification (CSPEC)

The CSPEC can be:

- state diagram (sequential spec)
- state transition table
- decision tables
- activation tables

combinatorial spec

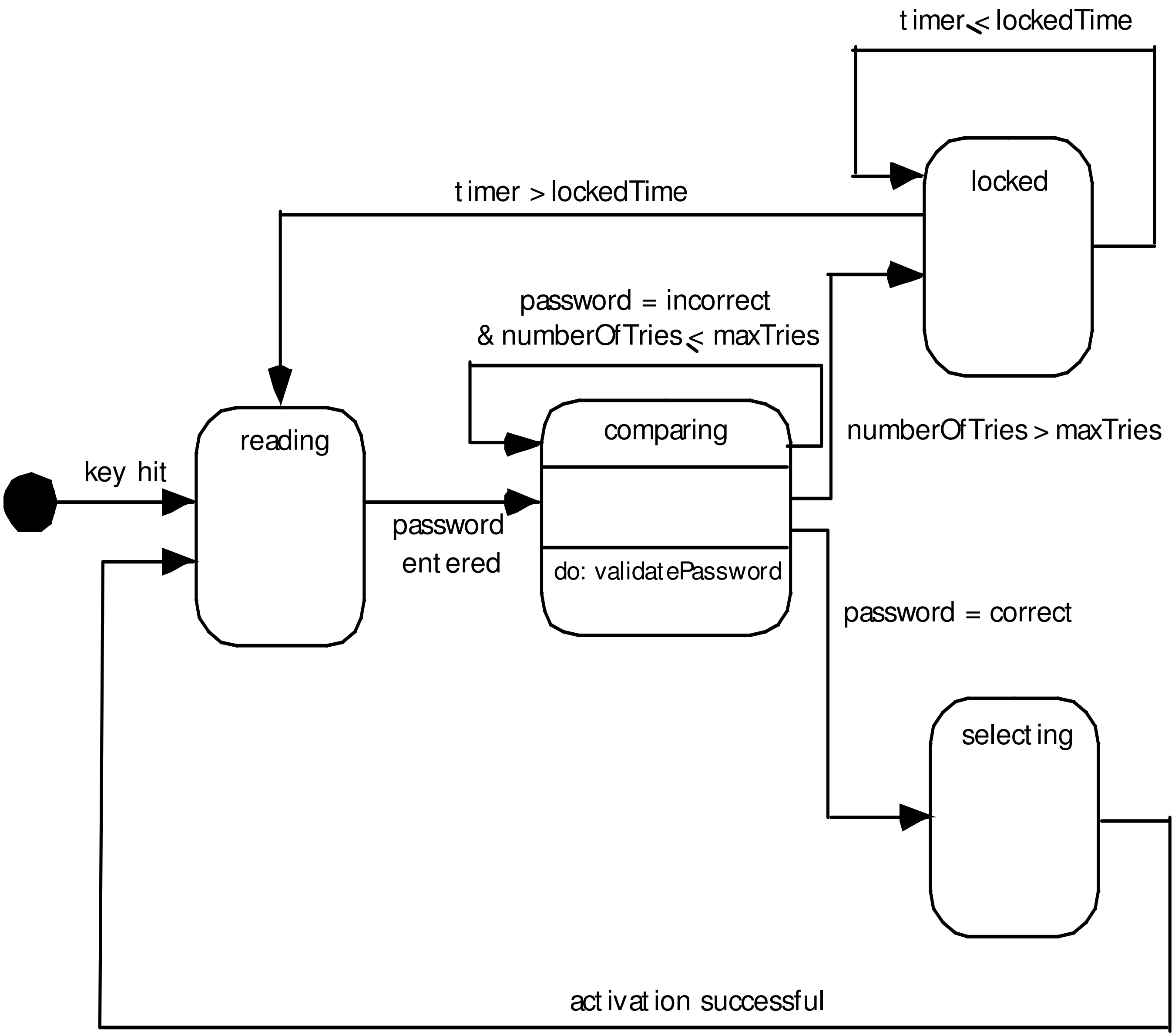
Behavioral Modeling

- The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:
  - Evaluate all use-cases to fully understand the sequence of interaction within the system.
  - Identify events that drive the interaction sequence and understand how these events relate to specific objects.
  - Create a sequence for each use-case.
  - Build a state diagram for the system.
  - Review the behavioral model to verify accuracy and consistency.

State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered:
  - » the state of each class as the system performs its function and
  - » the state of the system as observed from the outside as the system performs its function
- The state of a class takes on both passive and active characteristics [CHA93].
  - » A *passive state* is simply the current status of all of an object's attributes.
  - » The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

State Diagram for the ControlPanel Class



The States of a System

- state—a set of observable circumstances that characterizes the behavior of a system at a given time
- state transition—the movement from one state to another
- event—an occurrence that causes the system to exhibit some predictable form of behavior
- action—process that occurs as a consequence of making a transition

Behavioral Modeling

- make a list of the different states of a system (How does the system behave?)
- indicate how the system makes a transition from one state to another (How does the system change state?)
  - indicate event
  - indicate action
- draw a state diagram or a sequence diagram

Sequence Diagram

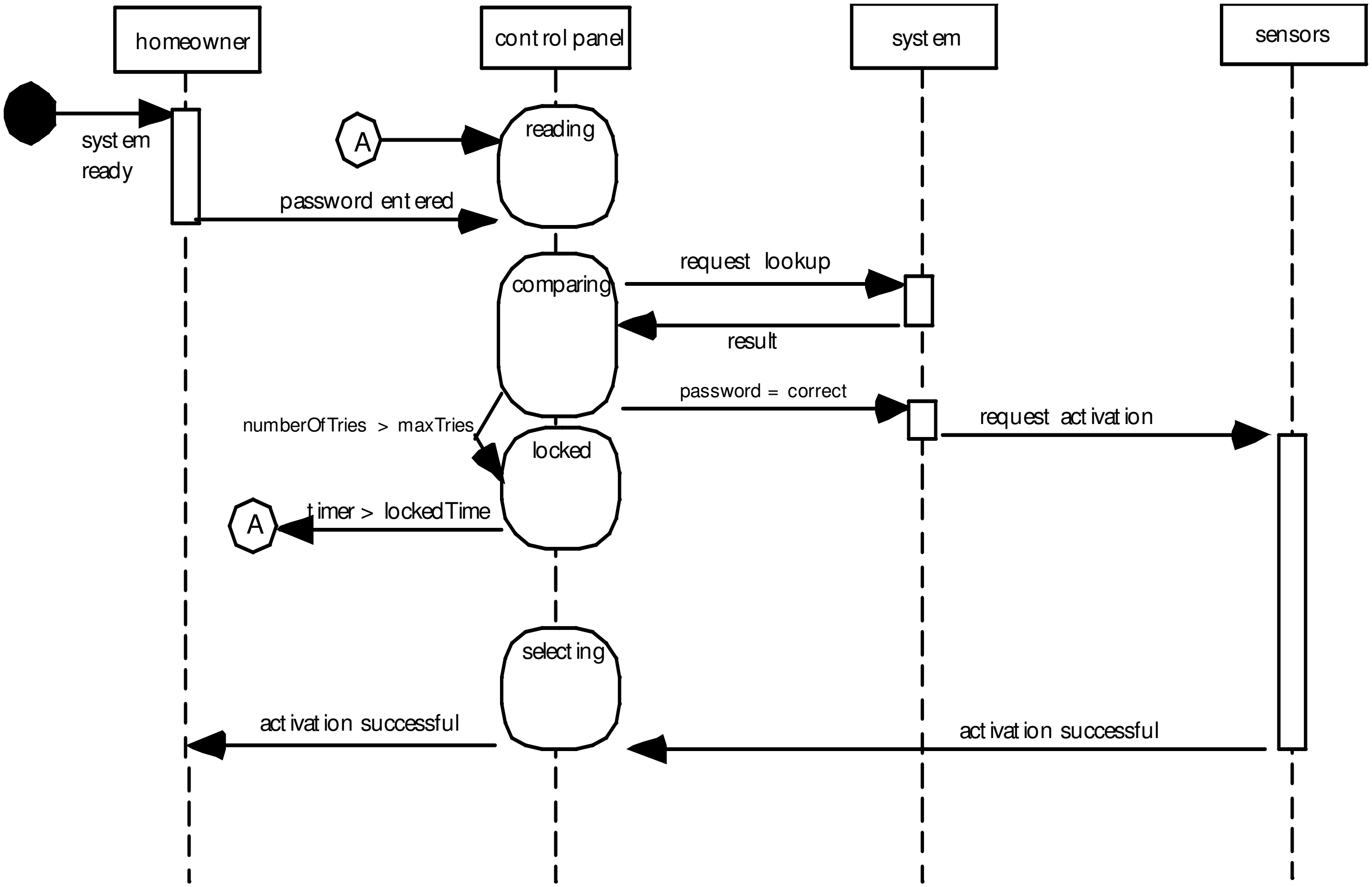
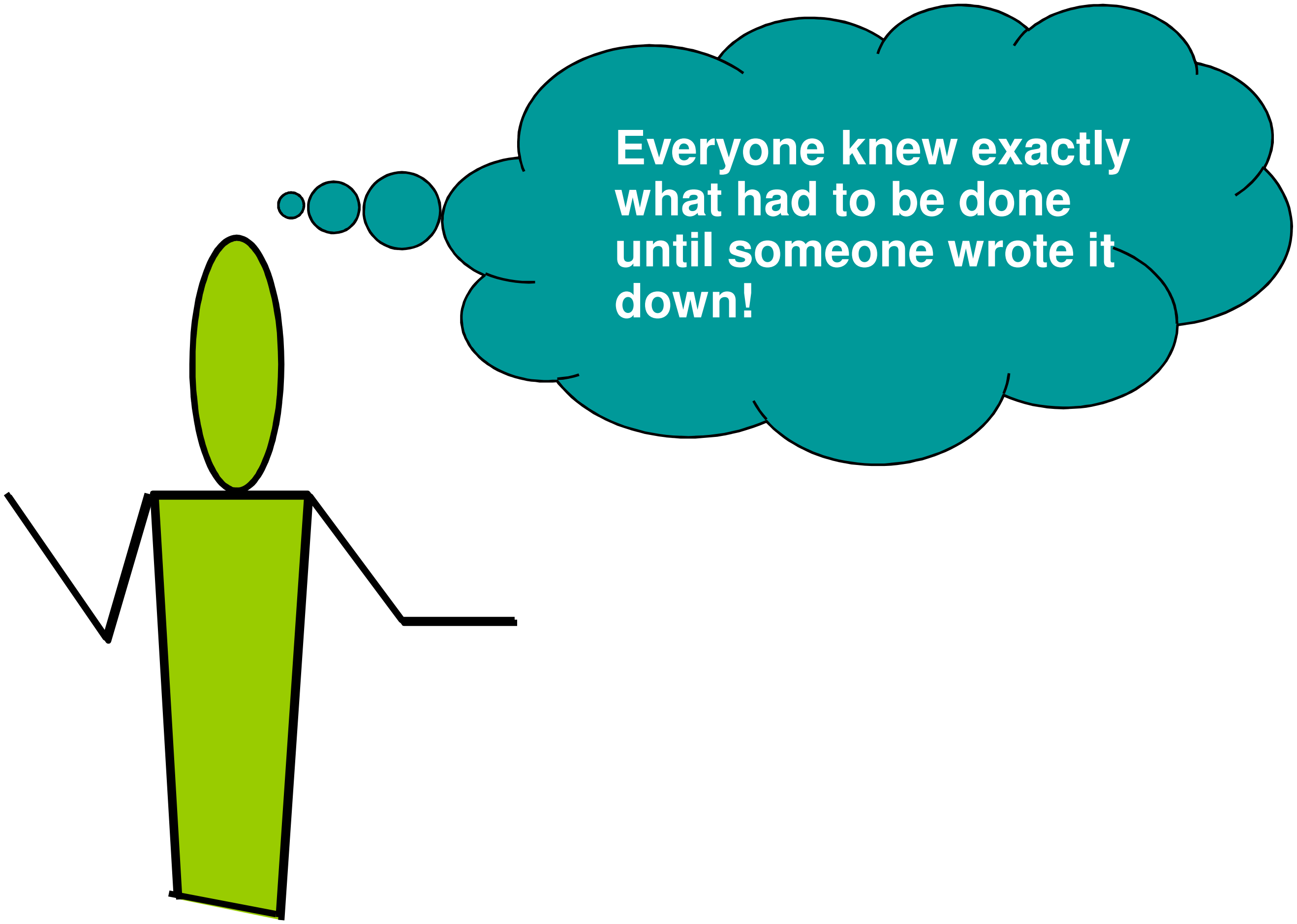


Figure 8.27 Sequence diagram (partial) for *SafeHome* security function

Writing the Software Specification


Everyone knew exactly what had to be done until someone wrote it down!

Patterns for Requirements Modeling

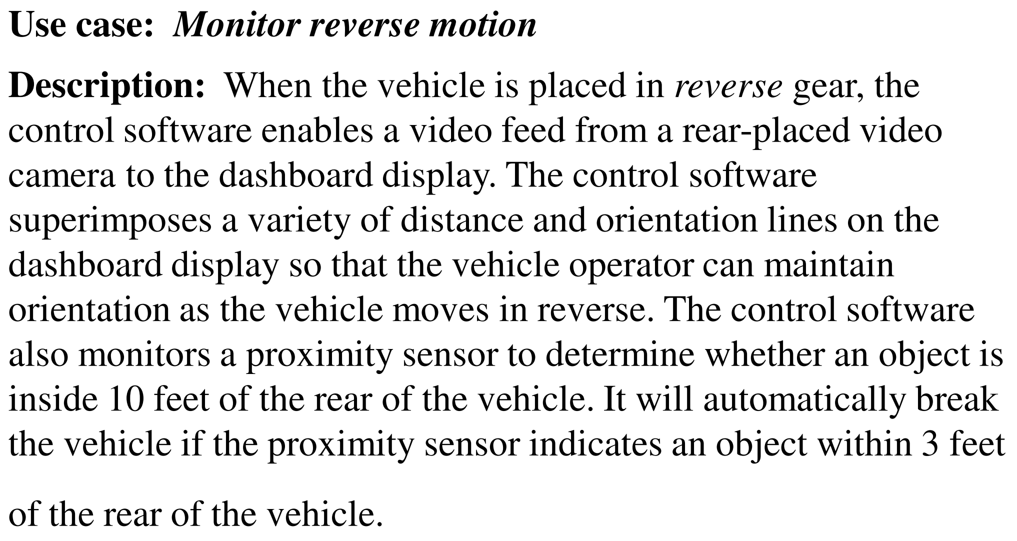
- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered
  - » domain knowledge can be applied to a new problem within the same application domain
  - » the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.
- The original author of an analysis pattern does not "create" the pattern, but rather, *discovers* it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented

Discovering Analysis Patterns

- The most basic element in the description of a requirements model is the use case.
- A coherent set of use cases may serve as the basis for discovering one or more analysis patterns.
- A *semantic analysis pattern* (SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application.” [Fer00]

An Example

- Consider the following preliminary use case for software required to control and monitor a real-view camera and proximity sensor for an automobile:



Use case: ***Monitor reverse motion***

Description: When the vehicle is placed in *reverse* gear, the control software enables a video feed from a rear-placed video camera to the dashboard display. The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse. The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle. It will automatically break the vehicle if the proximity sensor indicates an object within 3 feet of the rear of the vehicle.

An Example

- This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling.
- Regardless of how much elaboration is accomplished, the use case(s) suggest(s) a simple, yet widely applicable SAP—the software-based monitoring and control of sensors and actuators in a physical system.
- In this case, the “sensors” provide information about proximity and video information. The “actuator” is the breaking system of the vehicle (invoked if an object is very close to the vehicle.
- But in a more general case, a widely applicable pattern is discovered --> **Actuator-Sensor**

Actuator-Sensor Pattern—I

Pattern Name: ***Actuator-Sensor***

Intent: Specify various kinds of sensors and actuators in an embedded system.

Motivation: Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The *Actuator-Sensor* pattern uses a *pull* mechanism (explicit request for information) for **PassiveSensors** and a *push* mechanism (broadcast of information) for the **ActiveSensors**.

Constraints:

Each passive sensor must have some method to read sensor input and attributes that represent the sensor value.

Each active sensor must have capabilities to broadcast update messages when its value changes.

Each active sensor should send a *life tick*, a status message issued within a specified time frame, to detect malfunctions.

Each actuator must have some method to invoke the appropriate response determined by the **ComputingComponent**.

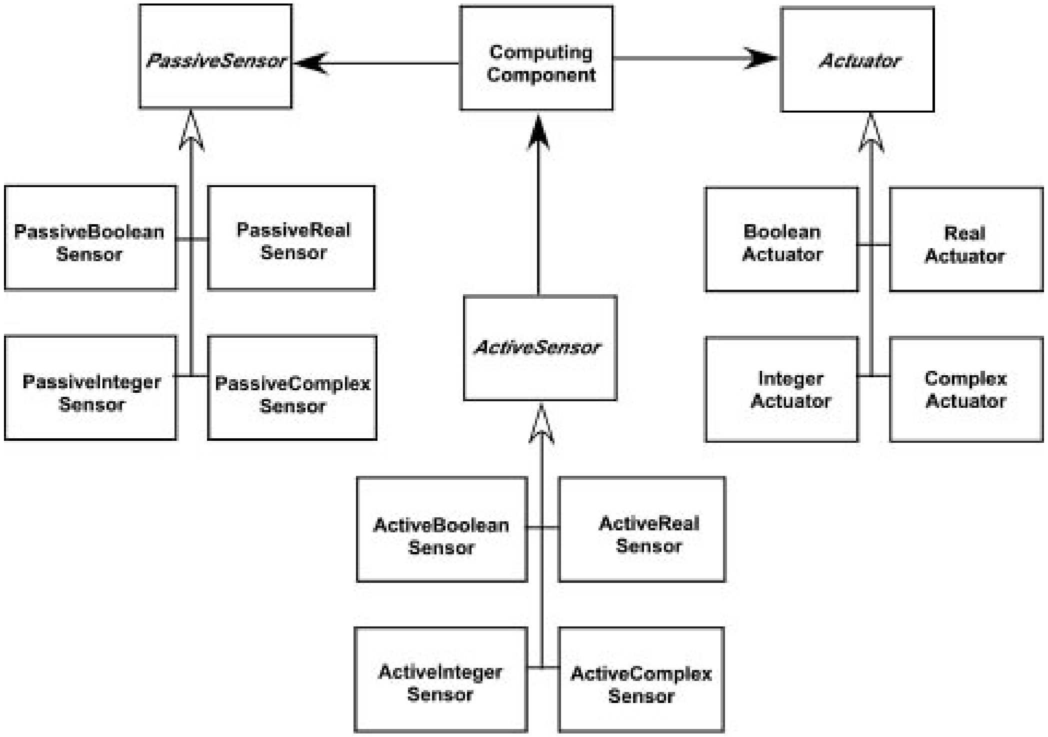
Each sensor and actuator should have a function implemented to check its own operation state.

Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications.

Actuator-Sensor Pattern—II

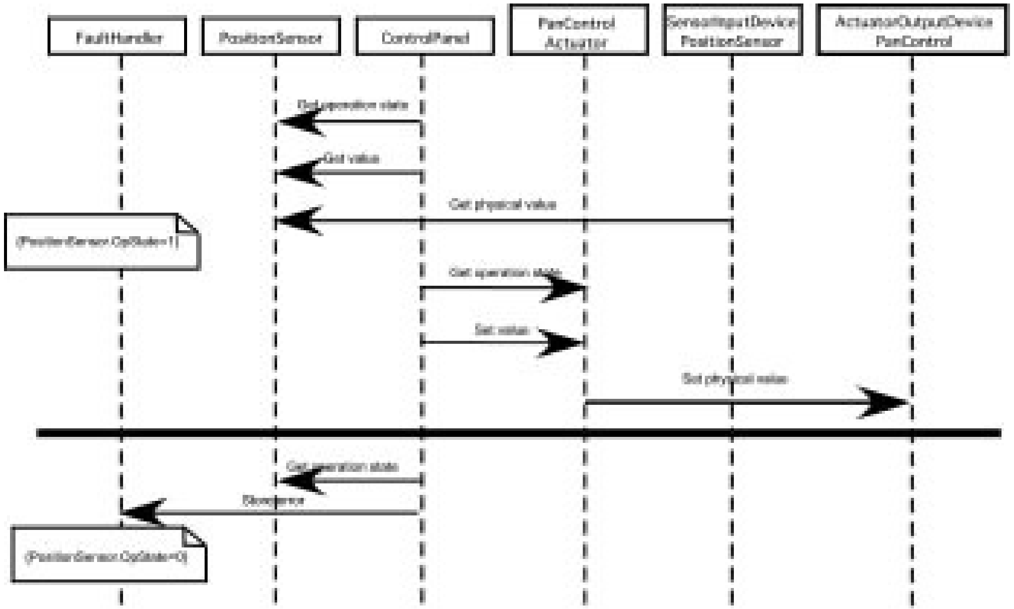
Applicability: Useful in any system in which multiple sensors and actuators are present.

Structure: A UML class diagram for the *Actuator-Sensor* Pattern is shown in Figure 7.8. **Actuator, PassiveSensor** and **ActiveSensor** are abstract classes and denoted in italics. There are four different types of sensors and actuators in this pattern. The Boolean, integer, and real classes represent the most common types of sensors and actuators. The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device. Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.



Actuator-Sensor Pattern—III

Behavior: Figure 7.9 presents a UML sequence diagram for an example of the *Actuator-Sensor* Pattern as it might be applied for the *SafeHome* function that controls the positioning (e.g., pan, zoom) of a security camera. Here, the **ControlPanel** queries a sensor (a passive position sensor) and an actuator (pan control) to check the operation state for diagnostic purposes before reading or setting a value. The messages *Set Physical Value* and *Get Physical Value* are not messages between objects. Instead, they describe the interaction between the physical devices of the system and their software counterparts. In the lower part of the diagram, below the horizontal line, the **PositionSensor** reports that the operation state is zero. The **ComputingComponent** then sends the error code for a position sensor failure to the **FaultHandler** that will decide how this error affects the system and what actions are required. it gets the data from the sensors and computes the required response for the actuators.



Actuator-Sensor Pattern—III

- See textbook for additional information on:
  - » Participants
  - » Collaborations
  - » Consequences

Requirements Modeling for WebApps

Content Analysis. The full spectrum of content to be provided by the WebApp is identified, including text, graphics and images, video, and audio data. Data modeling can be used to identify and describe each of the data objects.

Interaction Analysis. The manner in which the user interacts with the WebApp is described in detail. Use-cases can be developed to provide detailed descriptions of this interaction.

Functional Analysis. The usage scenarios (use-cases) created as part of interaction analysis define the operations that will be applied to WebApp content and imply other processing functions. All operations and functions are described in detail.

Configuration Analysis. The environment and infrastructure in which the WebApp resides are described in detail.

When Do We Perform Analysis?

- In some WebE situations, analysis and design merge. However, an explicit analysis activity occurs when …
  - » the WebApp to be built is large and/or complex
  - » the number of stakeholders is large
  - » the number of Web engineers and other contributors is large
  - » the goals and objectives (determined during formulation) for the WebApp will effect the business' bottom line
  - » the success of the WebApp will have a strong bearing on the success of the business

The Content Model

- Content objects are extracted from use-cases
  - » examine the scenario description for direct and indirect references to content
- Attributes of each content object are identified
- The relationships among content objects and/or the hierarchy of content maintained by a WebApp
  - » Relationships—entity-relationship diagram or UML
  - » Hierarchy—data tree or UML

Data Tree

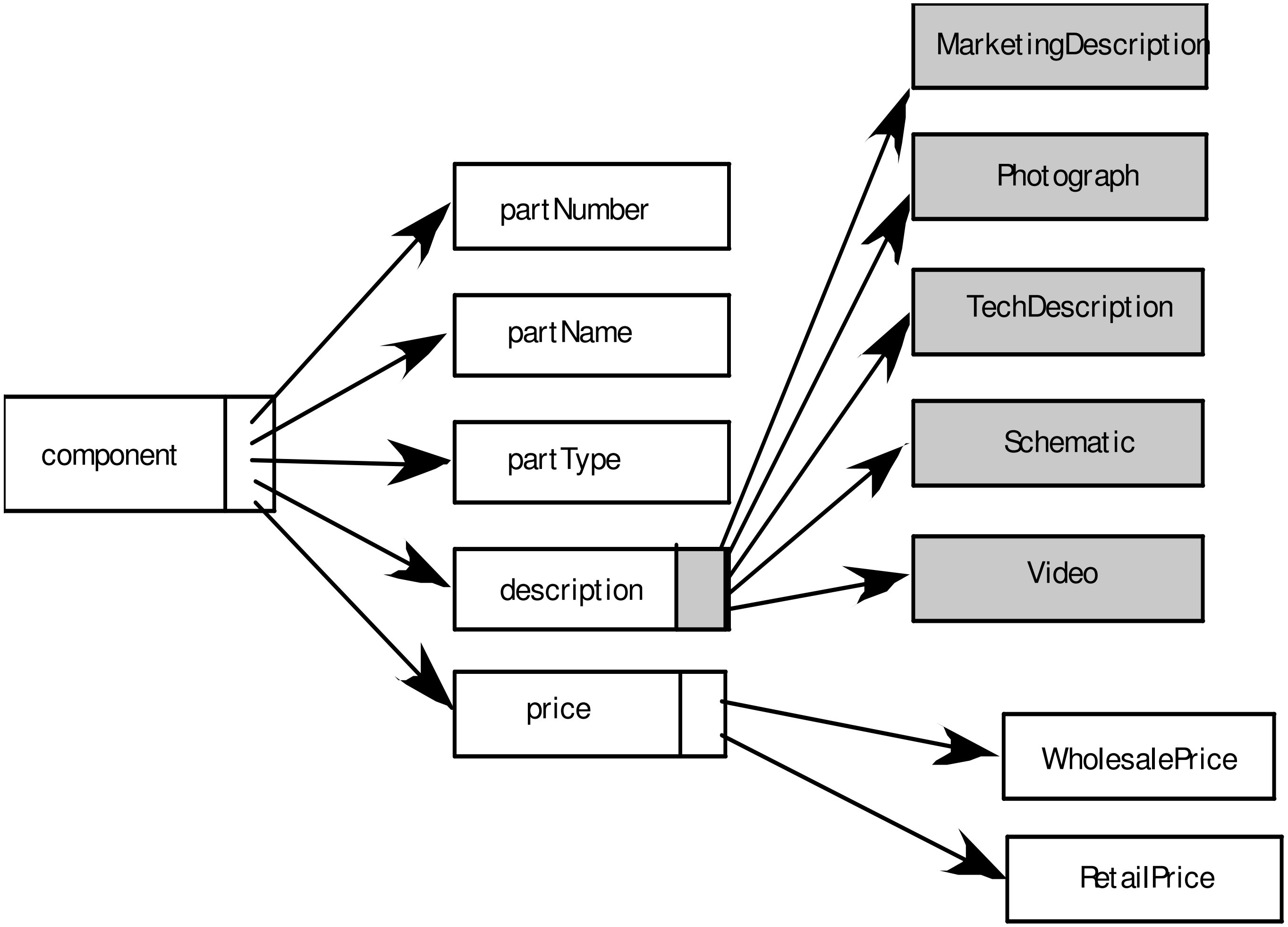


Figure 18.3 Data tree for a *SafeHome* component

The Interaction Model

- Composed of four elements:
  - use-cases
  - sequence diagrams
  - state diagrams
  - a user interface prototype
- Each of these is an important UML notation and is described in Appendix I of the textbook

Sequence Diagram

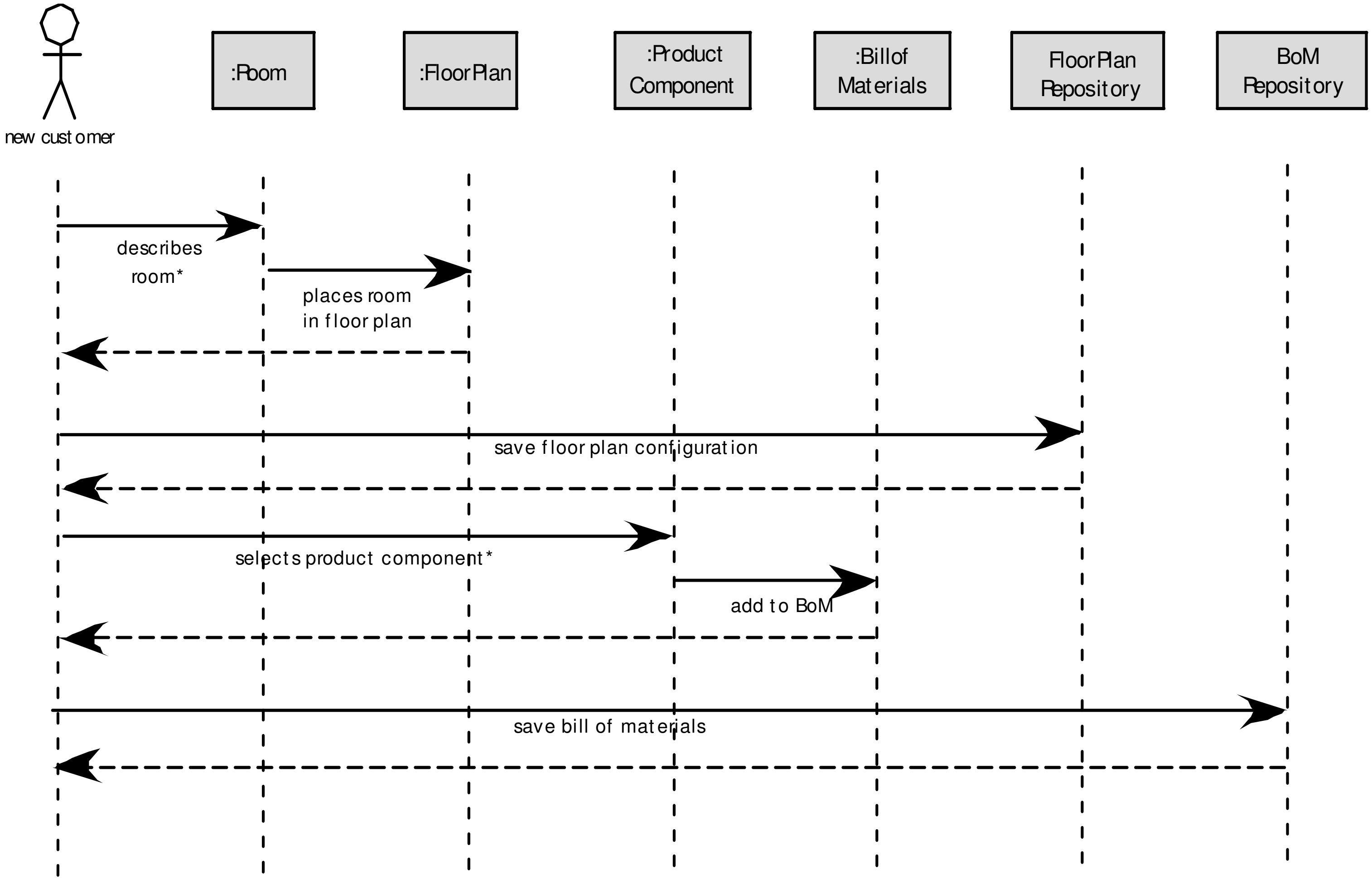
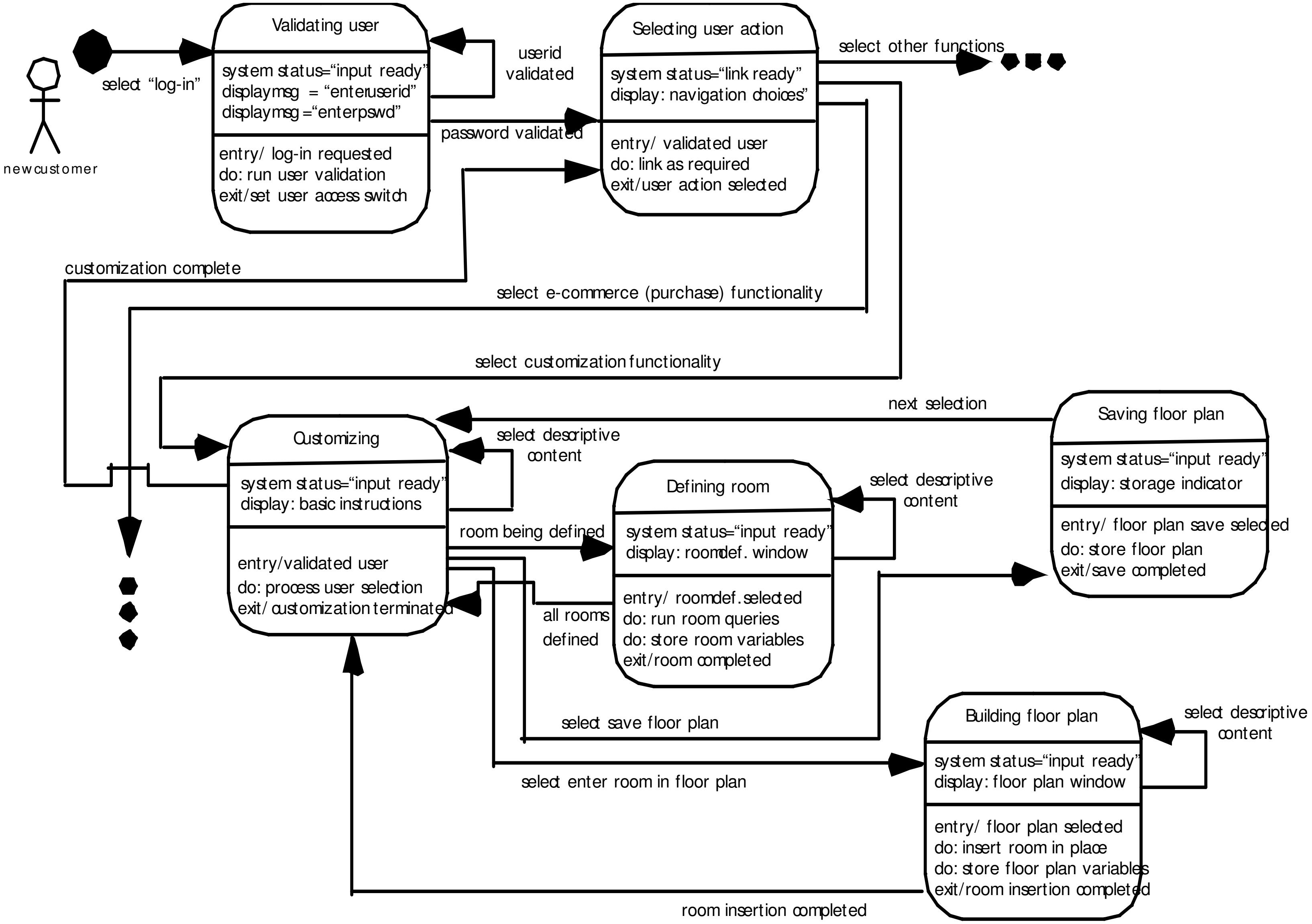


Figure 18.5 Sequence diagram for use-case: *select SafeHome components*

State Diagram



new customer

select "log-in"

Validating user

system status="input ready"
displaymsg = "enteruserid"
displaymsg ="enterpswd"

entry/ log-in requested
do: run user validation
exit/set user access switch

userid validated

password validated

Selecting user action

system status="link ready"
display: navigation choices"

entry/ validated user
do: link as required
exit/user action selected

select other functions

customization complete

select e-commerce (purchase) functionality

select customization functionality

next selection

Customizing

system status="input ready"
display: basic instructions

entry/validated user
do: process user selection
exit/ customization terminated

select descriptive content

room being defined

all rooms defined

Defining room

system status="input ready"
display: roomdef. window

entry/ roomdef. selected
do: run room queries
do: store room variables
exit/room completed

select descriptive content

Saving floor plan

system status="input ready"
display: storage indicator

entry/ floor plan save selected
do: store floor plan
exit/save completed

select save floor plan

select enter room in floor plan

Building floor plan

system status="input ready"
display: floor plan window

entry/ floor plan selected
do: insert room in place
do: store floor plan variables
exit/room insertion completed

select descriptive content

room insertion completed

Figure 18.6 Partial state diagram for **new customer** interaction

The Functional Model

- The functional model addresses two processing elements of the WebApp
  - » user observable functionality that is delivered by the WebApp to end-users
  - » the operations contained within analysis classes that implement behaviors associated with the class.
- An activity diagram can be used to represent processing flow

Activity Diagram

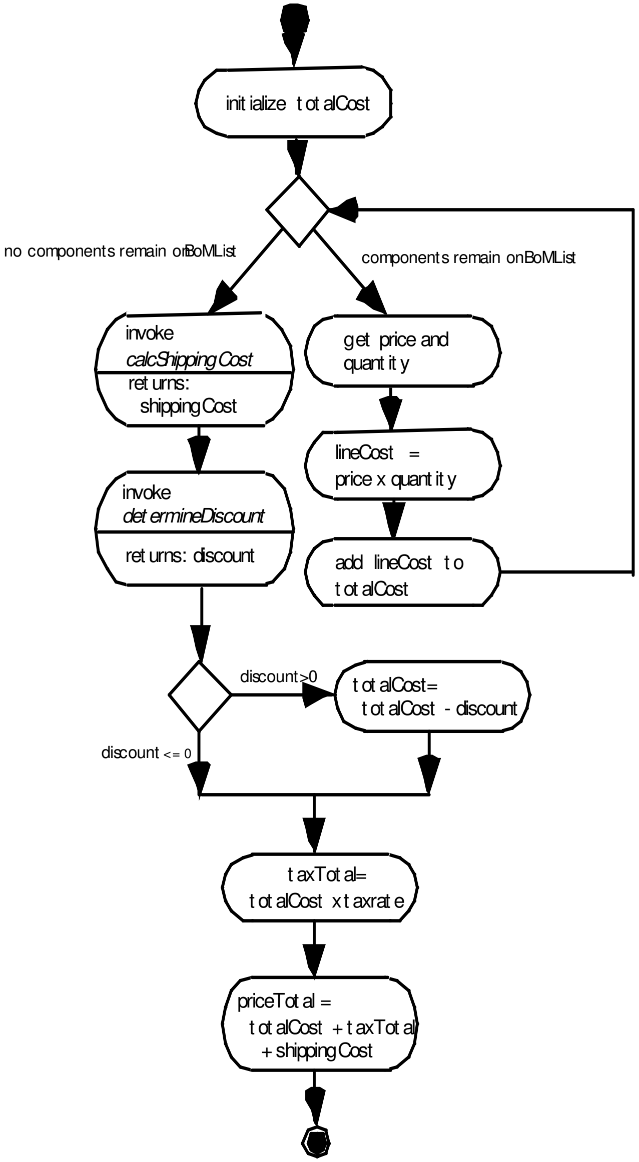


Figure 18.7 Activity diagram for ***computePrice()operatic***

The Configuration Model

- Server-side
  - » Server hardware and operating system environment must be specified
  - » Interoperability considerations on the server-side must be considered
  - » Appropriate interfaces, communication protocols and related collaborative information must be specified
- Client-side
  - » Browser configuration issues must be identified
  - » Testing requirements should be defined

Navigation Modeling-I

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element.
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?

Navigation Modeling-II

- Should a full navigation map or menu (as opposed to a single “back” link or directed pointer) be available at every point in a user’s interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user “store” his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? overlaying the existing browser window? as a new browser window? as a separate frame?

Agenda

1 **Introduction**

2 **Requirements Analysis**

3 **Requirements Modeling**

➡ 4 **Design Concepts**

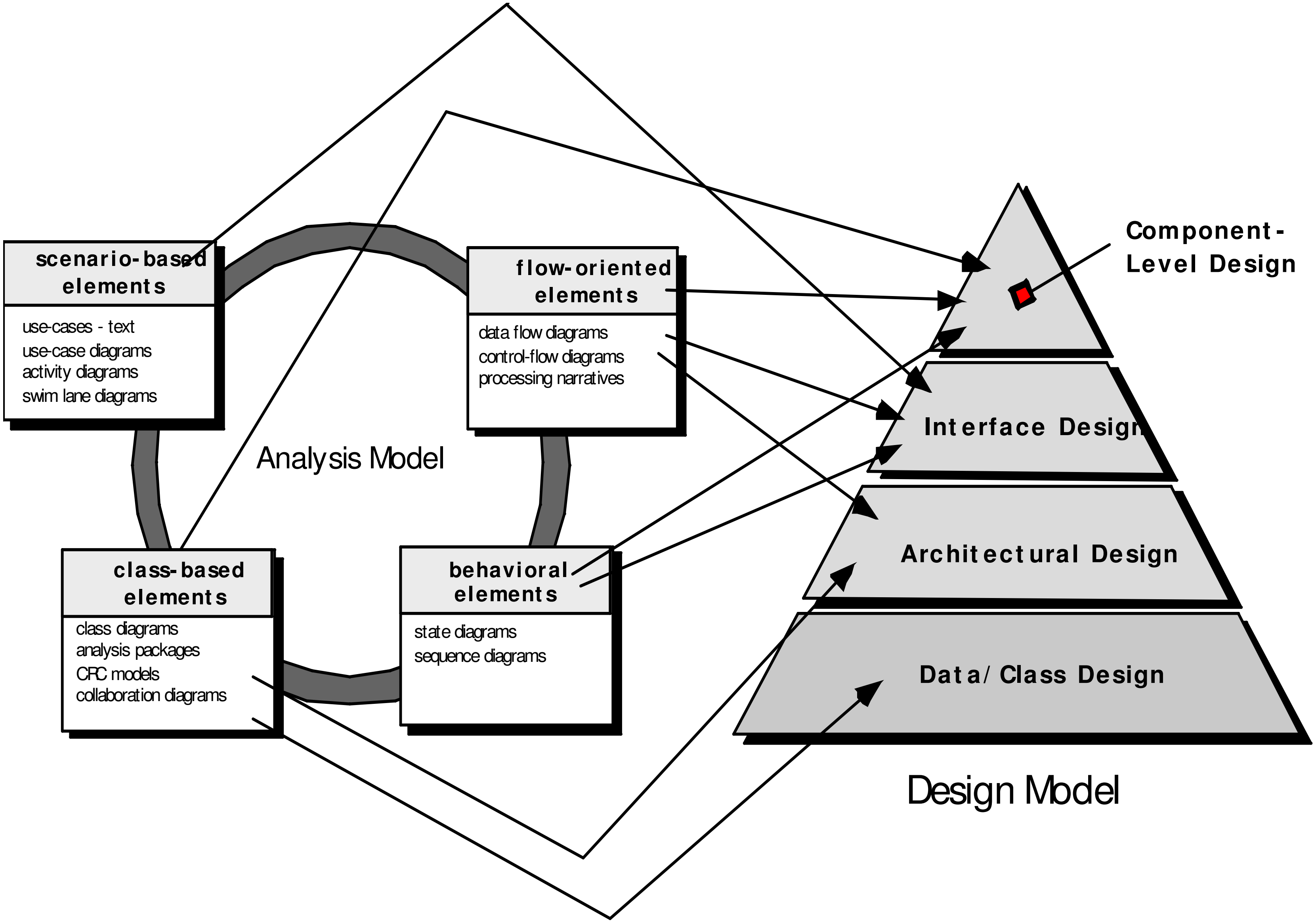
5 **Sample Analysis and Design Exercise Using UML**

6 **Summary and Conclusion**

Design

- Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal*. He said:
  - Good software design should exhibit:
  - *Firmness:* A program should not have any bugs that inhibit its function.
  - *Commodity:* A program should be suitable for the purposes for which it was intended.
  - *Delight:* The experience of using the program should be pleasurable one.

Analysis Model -> Design Model



Design and Quality

- the design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- the design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- the design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  - For smaller systems, design can sometimes be developed linearly.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

Design Principles

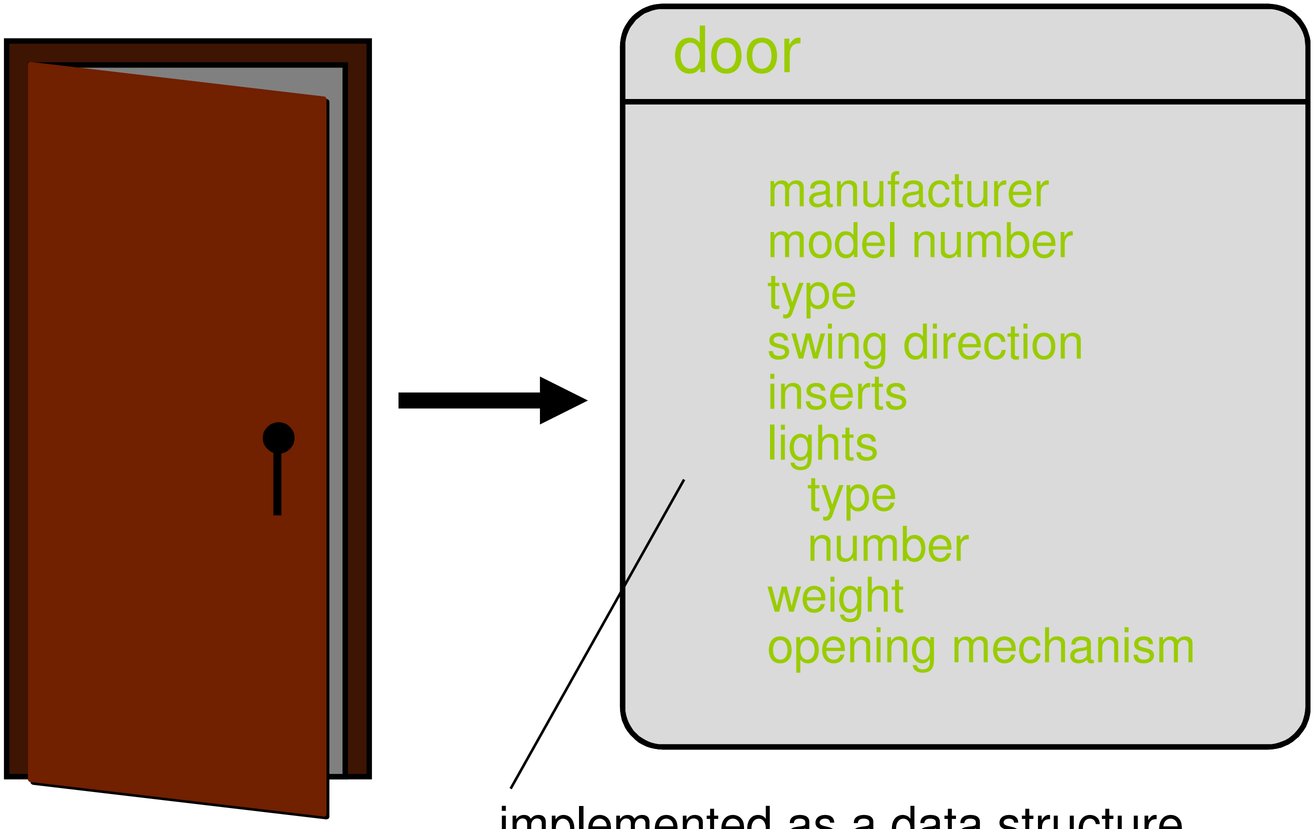
- The design process should not suffer from 'tunnel vision.'
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" [DAV95] between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

From Davis [DAV95]

Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—Appendix II
- Design Classes—provide design detail that will enable analysis classes to be implemented

Data Abstraction

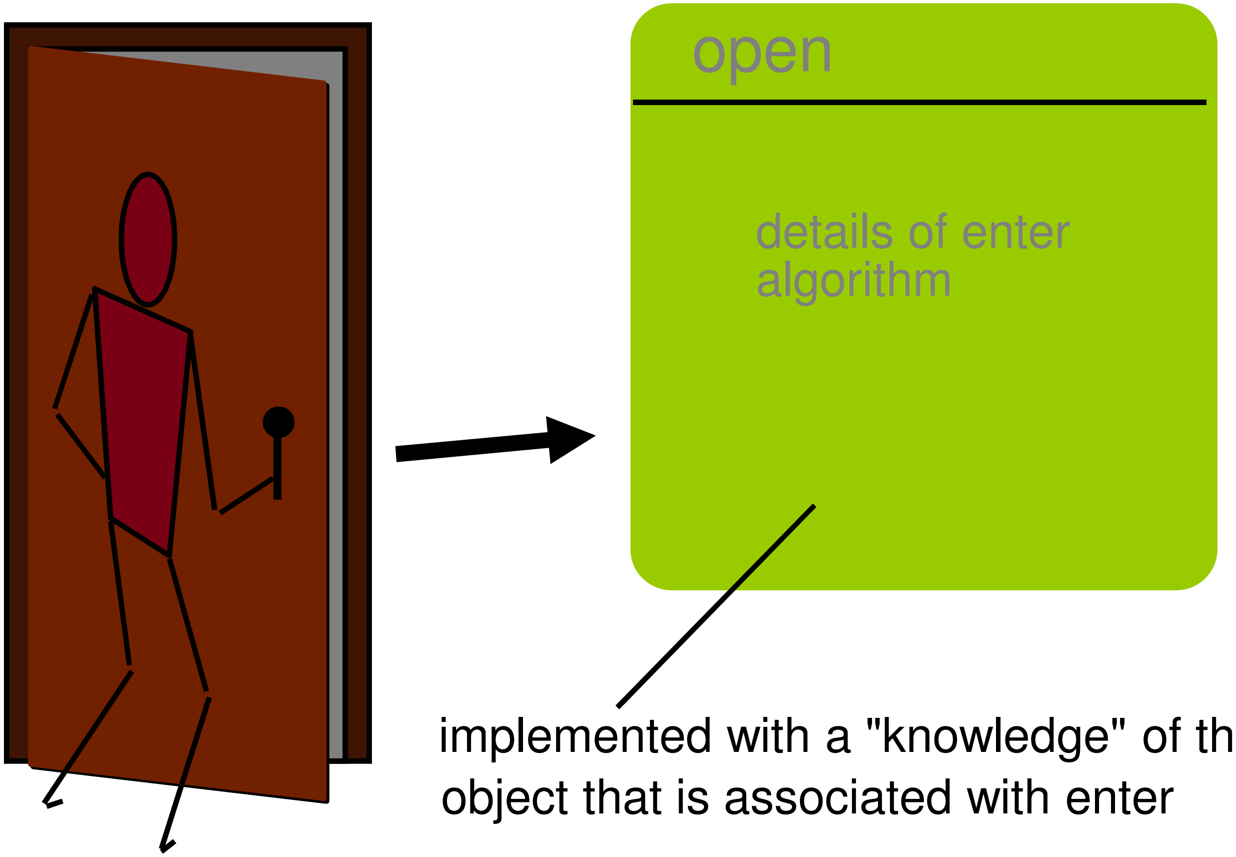


door

manufacturer
model number
type
swing direction
inserts
lights
  type
  number
weight
opening mechanism

implemented as a data structure

Procedural Abstraction



open

details of enter algorithm

implemented with a "knowledge" of the object that is associated with enter

Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.” [SHA95a]

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name

Intent—describes the pattern and what it does

Also-known-as—lists any synonyms for the pattern

Motivation—provides an example of the problem

Applicability—notes specific design situations in which the pattern is applicable

Structure—describes the classes that are required to implement the pattern

Participants—describes the responsibilities of the classes that are required to implement the pattern

Collaborations—describes how the participants collaborate to carry out their responsibilities

Consequences—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

Related patterns—cross-references related design patterns

Separation of Concerns

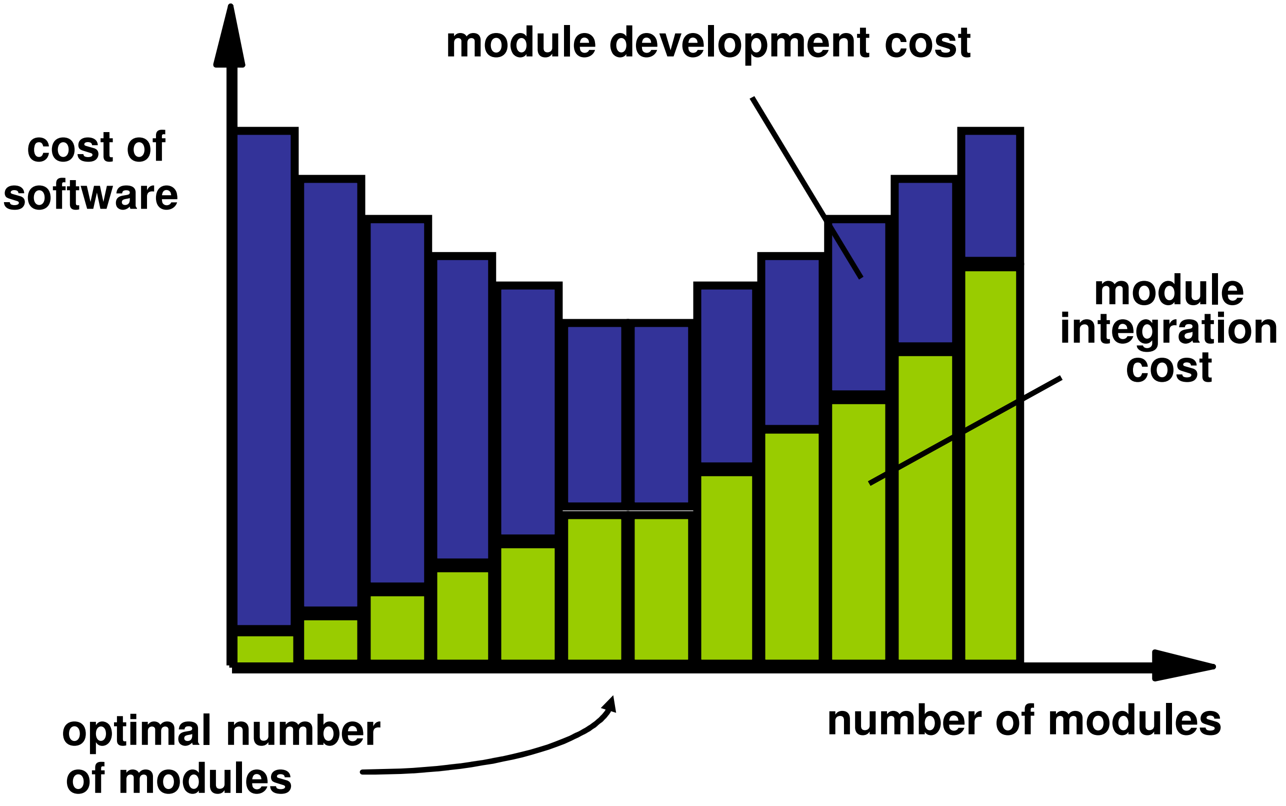
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

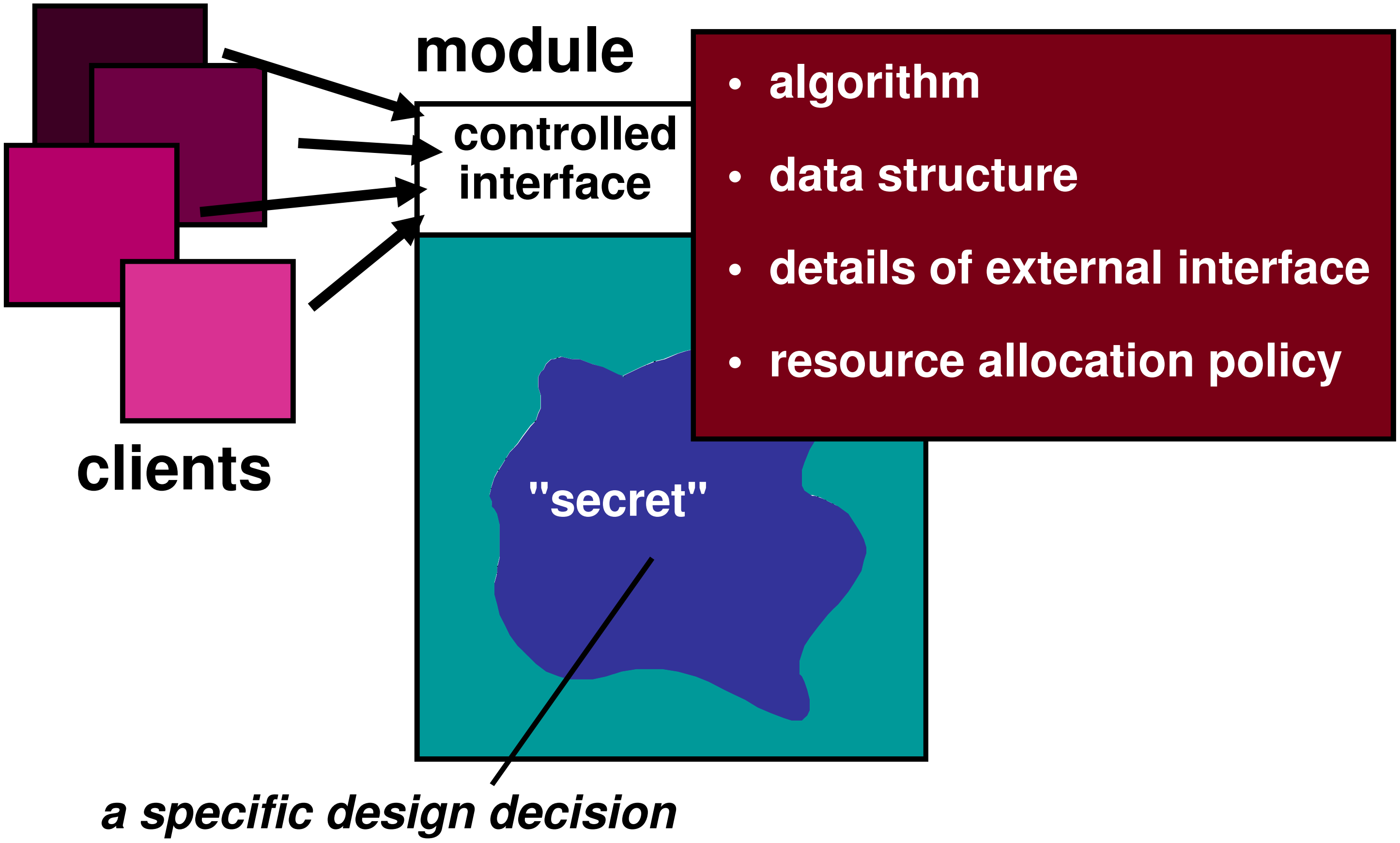
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - » The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



Information Hiding

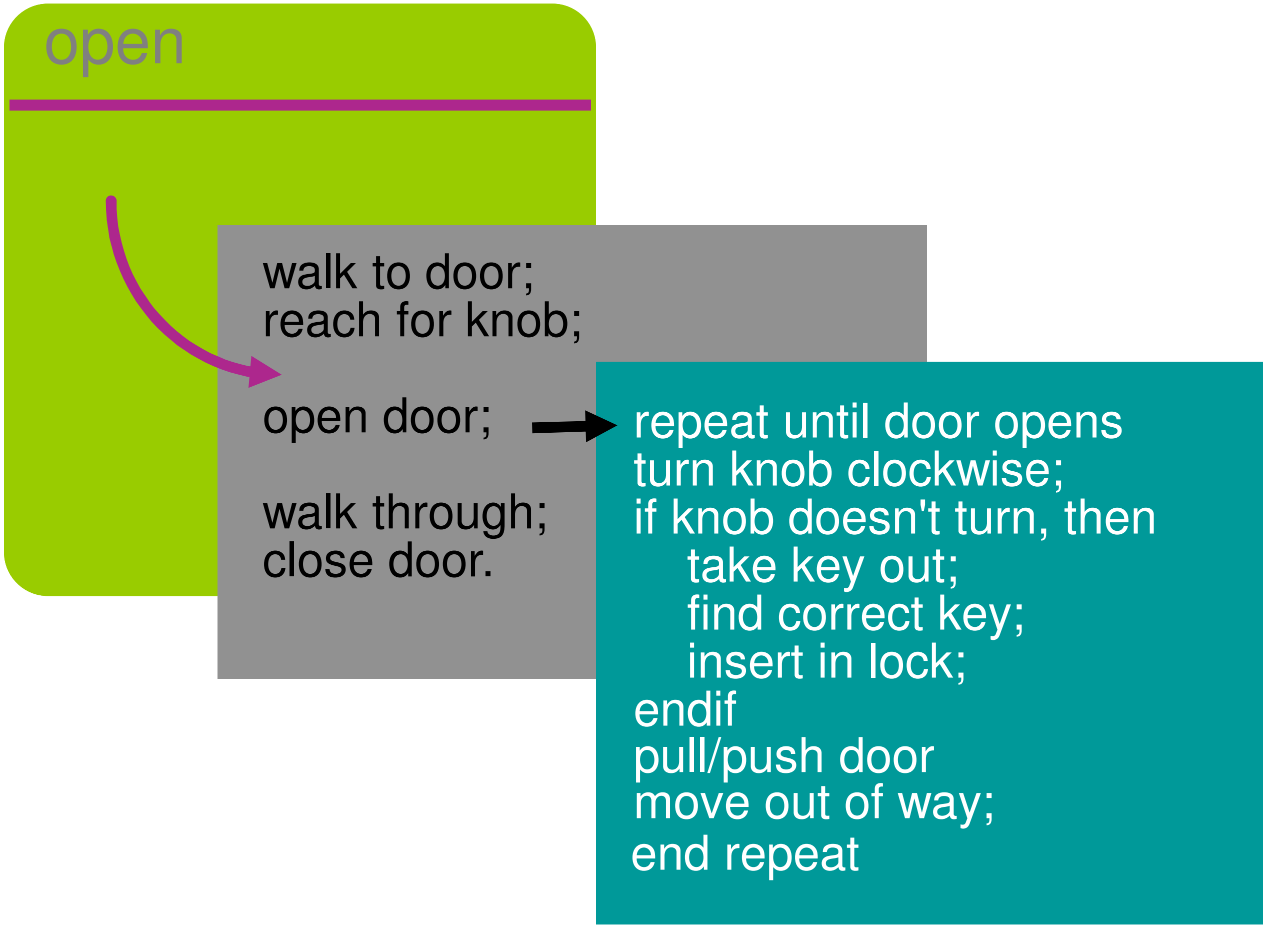


- algorithm
- data structure
- details of external interface
- resource allocation policy

Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Stepwise Refinement

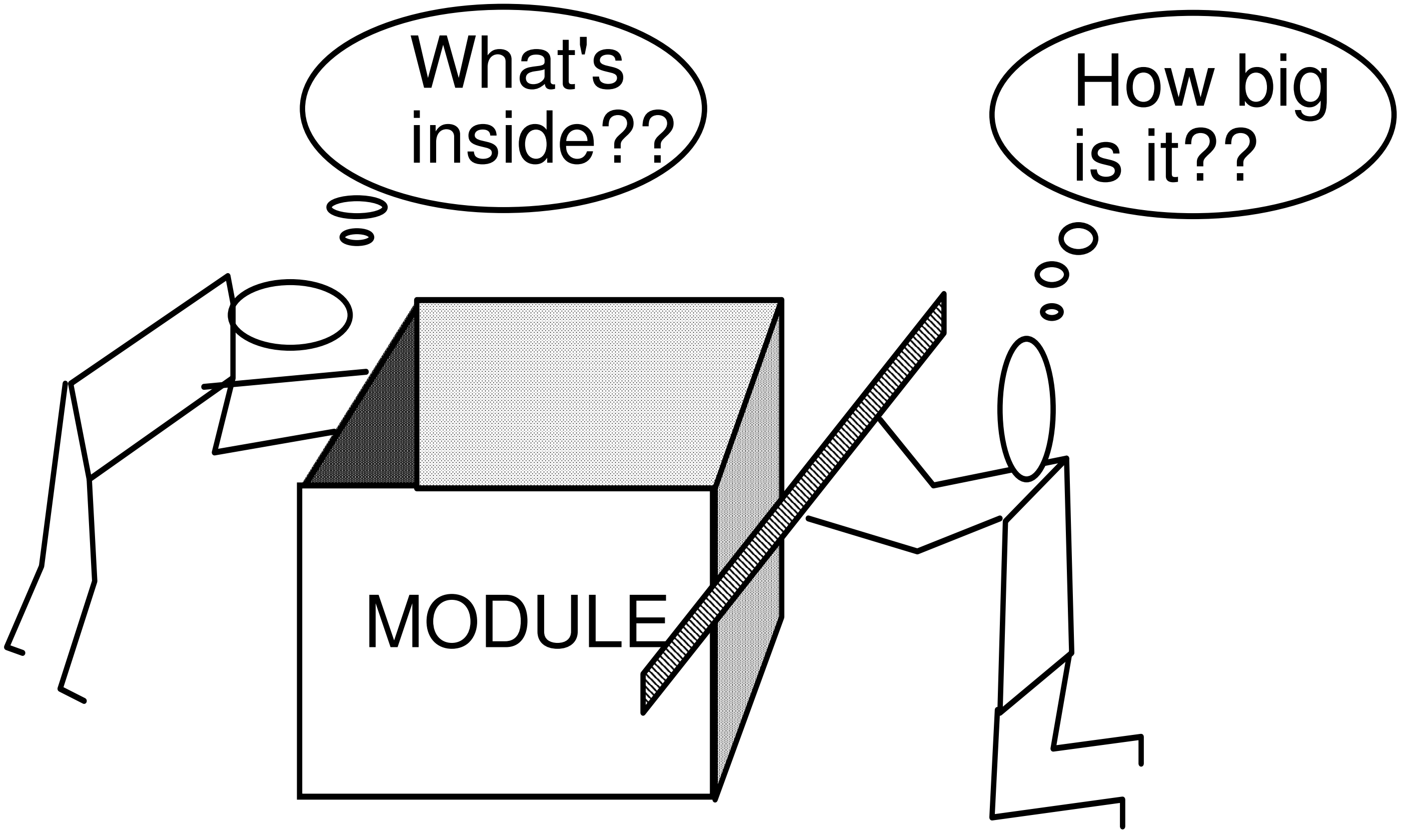


open

```
walk to door;
reach for knob;

open door;

walk through;
close door.
```

```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

Sizing Modules: Two Views



Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - » A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - » Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An *aspect* is a representation of a cross-cutting concern.

Aspects—An Example

- Consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the use-case **Access camera surveillance via the Internet.** A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using* **SafeHomeAssured.com.** This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and B* *cross-cuts* A*.
- An *aspect* is a representation of a cross-cutting concern. Therefore, the design representation, *B**, of the requirement, *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp.

Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design.

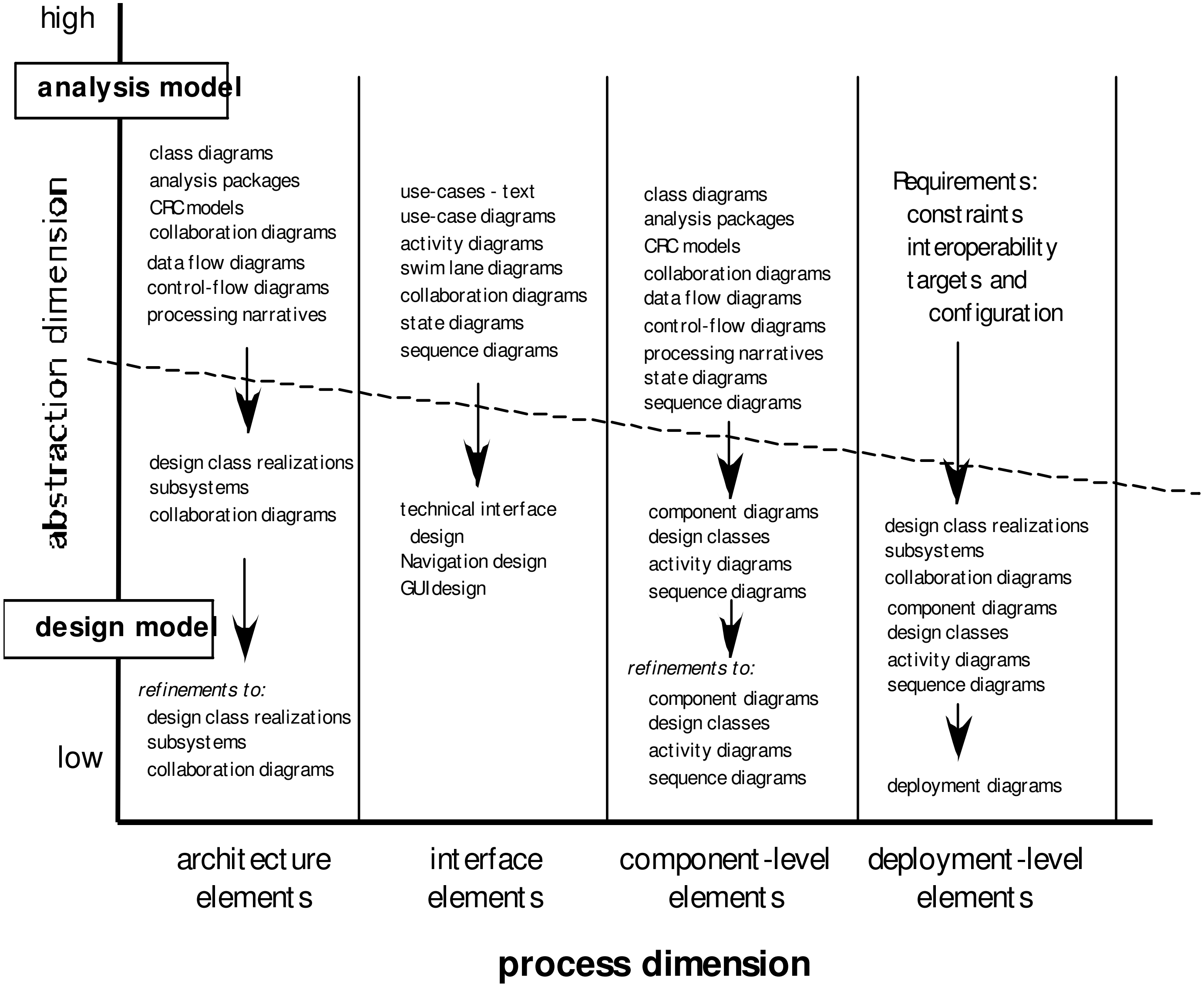
OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design Classes

- Analysis classes are refined during design to become entity classes
- Boundary classes are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- Controller classes are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

The Design Model



high

analysis model

abstraction dimension

design model

low

class diagrams
analysis packages
CRC models
collaboration diagrams
data flow diagrams
control-flow diagrams
processing narratives

design class realizations
subsystems
collaboration diagrams

refinements to:
design class realizations
subsystems
collaboration diagrams

use-cases - text
use-case diagrams
activity diagrams
swim lane diagrams
collaboration diagrams
state diagrams
sequence diagrams

technical interface
design
Navigation design
GUI design

class diagrams
analysis packages
CRC models
collaboration diagrams
data flow diagrams
control-flow diagrams
processing narratives
state diagrams
sequence diagrams

component diagrams
design classes
activity diagrams
sequence diagrams

refinements to:
component diagrams
design classes
activity diagrams
sequence diagrams

Requirements:
constraints
interoperability
targets and
configuration

design class realizations
subsystems
collaboration diagrams
component diagrams
design classes
activity diagrams
sequence diagrams

deployment diagrams

architecture elements

interface elements

component-level elements

deployment-level elements

process dimension

Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles” (see textbook chapters 9 and 12)
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

Architectural Elements

- The architectural model [Sha96] is derived from three sources:
  - information about the application domain for the software to be built;
  - specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
  - the availability of architectural patterns (see textbook chapter 12) and styles (see textbook chapter 9).

Interface Elements

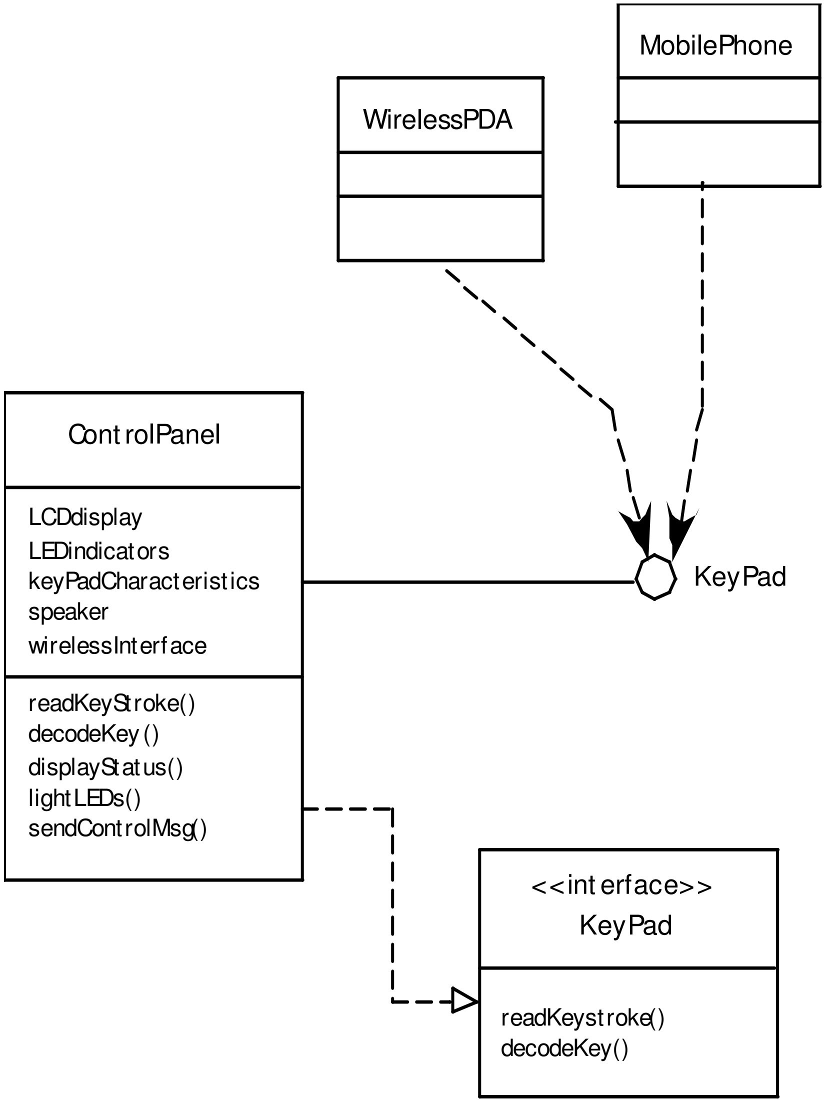
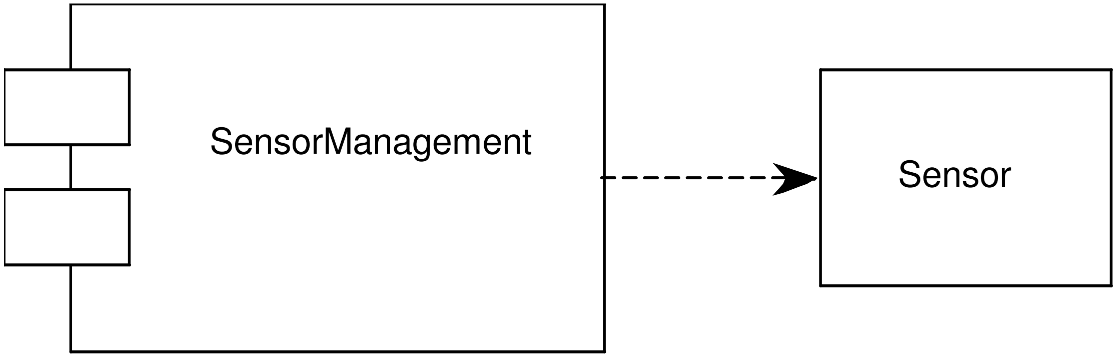


Figure 9.6 UML interface representation for **ControlPanel**

Component Elements



Deployment Elements

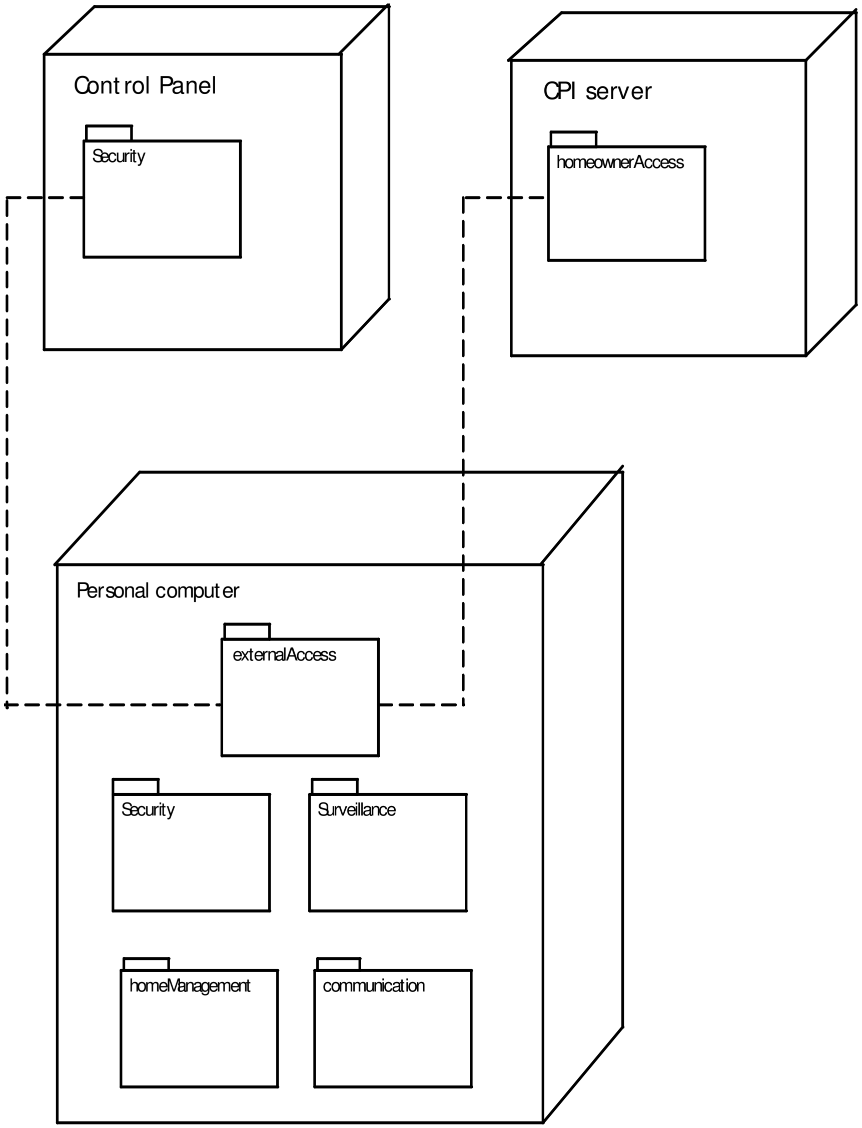
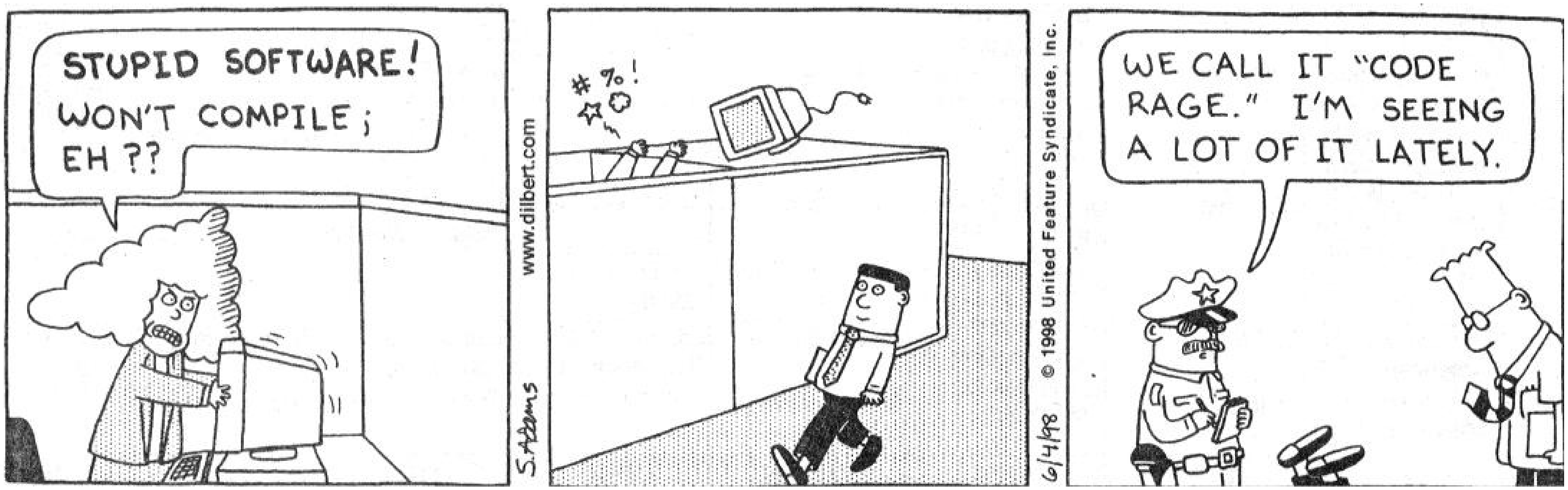


Figure 9.8 UML deployment diagram for *SafeHome*

Agenda

1. Introduction
2. Requirements Analysis
3. Requirements Modeling
4. Design Concepts
5. Sample Analysis and Design Exercise Using UML
6. Summary and Conclusion


STUPID SOFTWARE! WON'T COMPILE; EH??
www.dilbert.com
S.Adams
# % !
© 1998 United Feature Syndicate, Inc.
6/4/98
WE CALL IT "CODE RAGE." I'M SEEING A LOT OF IT LATELY.

UML …

- … is a modeling language, a notation used to express and document designs
- … unifies the notation of Booch, Rumbaugh (OMT) and Jacobson, and augmented with other contributors once submitted to OMG
- … proposes a standard for technical exchange of models and designs
- … defines a “meta-model”, a diagram that defines the syntax of the UML notation

UML is not …

- … a method or methodology (Method = Notation (e.g.,UML) + Process)
- … a proponent of a particular process (although the “Rational Objectory Process” is being proposed by Booch, Rumbaugh and Jacobson)

Starting Point

- Identify key domain abstractions … classes integrating:
  - » Attributes
  - » Behavior (responsibilities, methods)
  - » Messaging
    - providing logical independence between client and object
  - » Polymorphism
    - providing physical independence between client and implementation
- Consider relationships … integrating classes and objects to form higher levels of abstraction
  - » Association (“Uses, Needs”)
  - » Aggregation (“Has-A”)
  - » Inheritance (“Is-A”)

Model Perspectives

- Conceptual
  - » Book [Title]
  - » objects, “things” from the domain
  - » conceptual map to implementation
- Specification
  - » BookIface { void setTitle(String value); }
  - » identifies how to obtain properties
- Implementation
  - » PersistentBook : BookIface { -> DB }
  - » identifies how interface will be implemented

Model Perspective Hints

- Works as a map of the system
- Different subsystems become UML packages
- Keep dependencies simple and domain-related
- Define relationships and interactions between packages
- Address both functional and non-functional requirements
- Take time to factor in reuse

Initial Modeling Results

- List of use cases, describing system requirements
- Domain model, capturing your understanding of the business process and key domain classes
- Design model, realizing both the information in the domain objects and the behavior described in the use cases
- Add classes in the design model that actually do the work and also provide a reusable architecture for future extensions

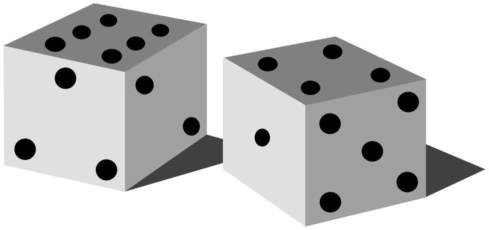
UML 1.x Notation Baseline

Diagram Name	Type	Phase
Use Case	Static*	Analysis
Class	Static	Analysis
Activity	Dynamic**	Analysis
State-Transition	Dynamic	Analysis
Event Trace (Interaction)	Dynamic	Design
Sequence	Static	Design
Collaboration	Dynamic	Design
Package	Static	Delivery
Deployment	Dynamic	Delivery

*Static describes structural system properties

**Dynamic describes behavioral system properties.

Example (Using UML 1.x)

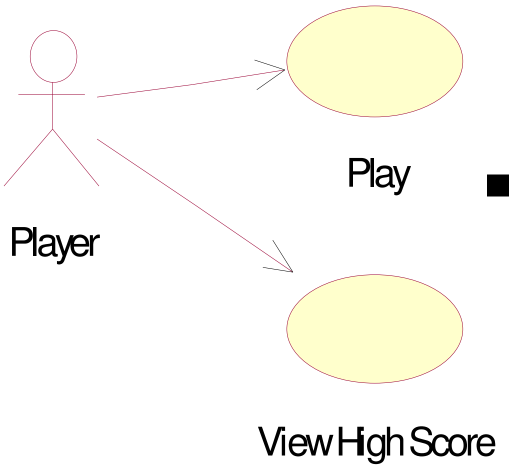


- A die
- The player throws die 10 x 2
- When total is 7, player scores 10 points
- At the end of the game, the score is collected in a score map

Requirements Analysis

- First Use Case
- Identify Actors?
- Identify possible **System** use cases
- External functionality !

First Use Case



- Play:
  - Actor: Player
  - Descr: Player rolls the dices 10 times, whenever total is 7, +10pts
- View High Score
  - Actor: Player
  - Descr: Player looks up highest score in read-only mode

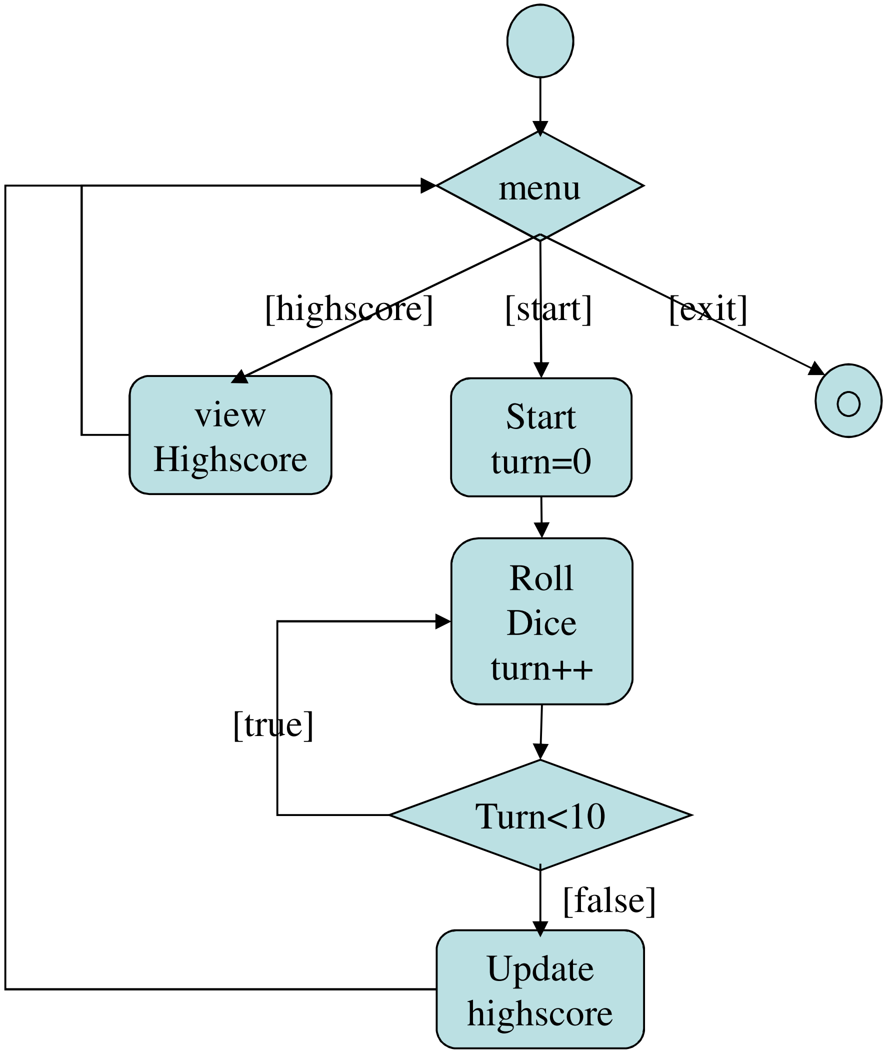
Use Case

- Very important diagram !
- A must for requirements analysis
- A must to present an application !
- MUST BE formally commented
- Used as a reference for all remaining modeling stages

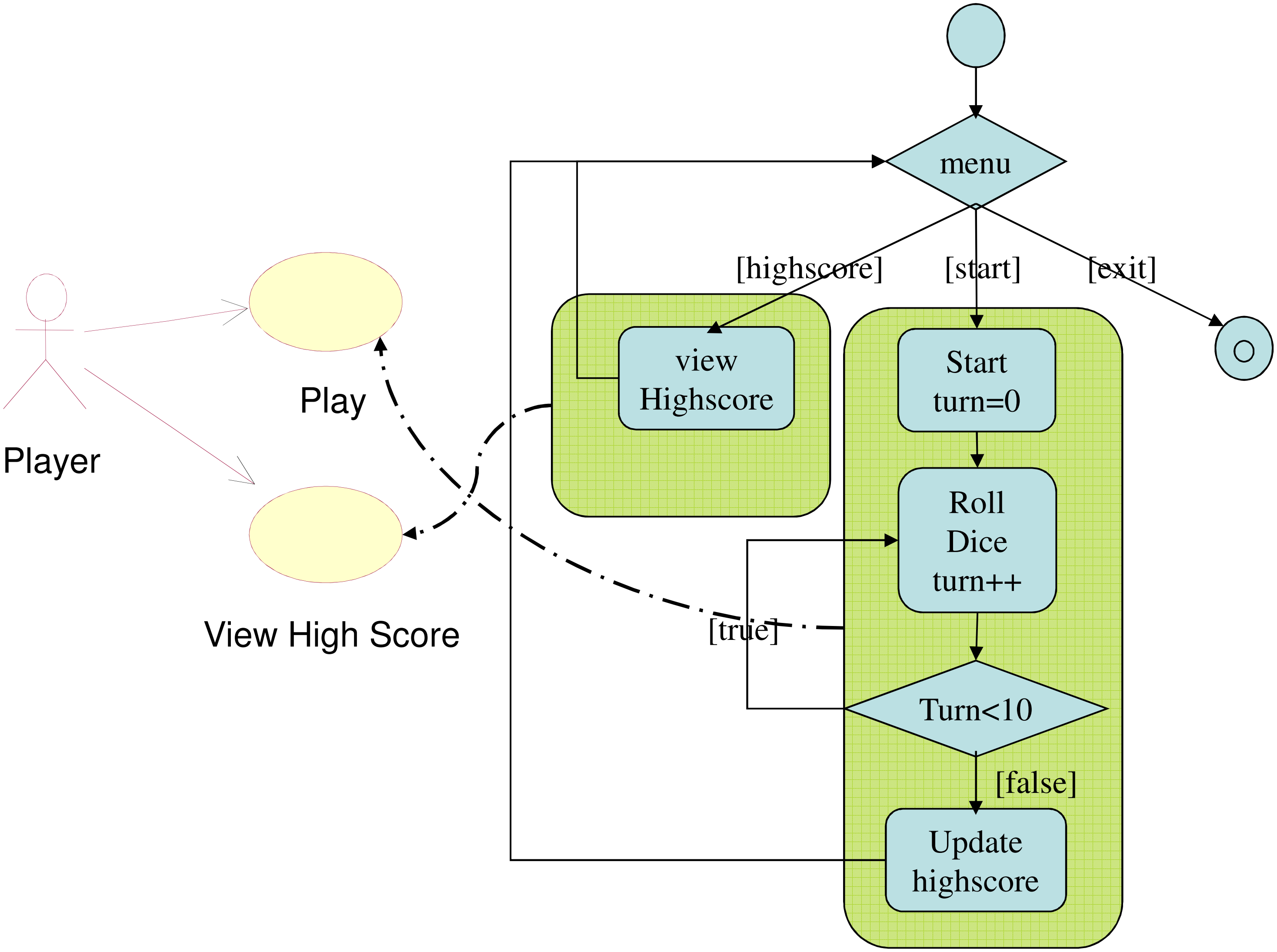
Activity Diagram

- Looks awfully close to a flow diagram
- Identify activities based on a use case
- Identify transitions between activities

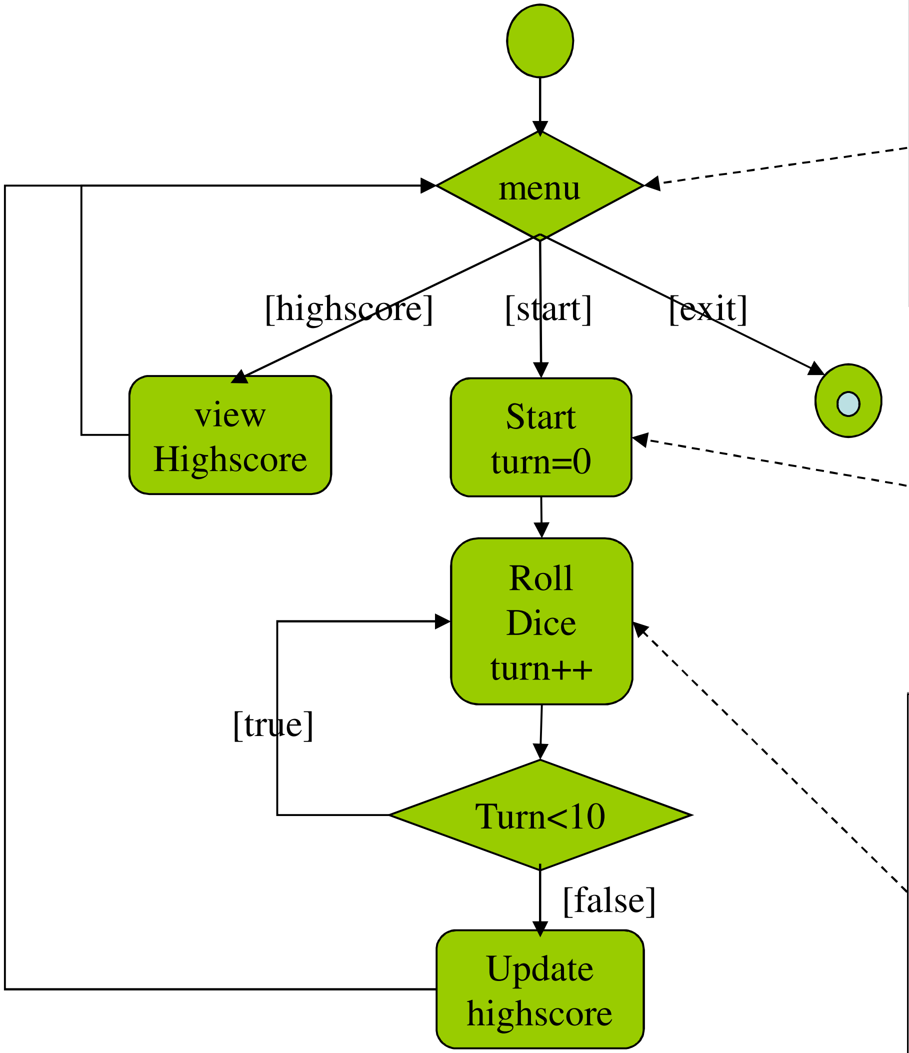
Activity Diagram

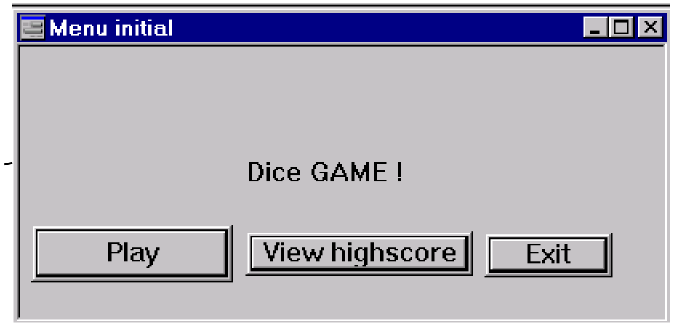


Activity Diagram

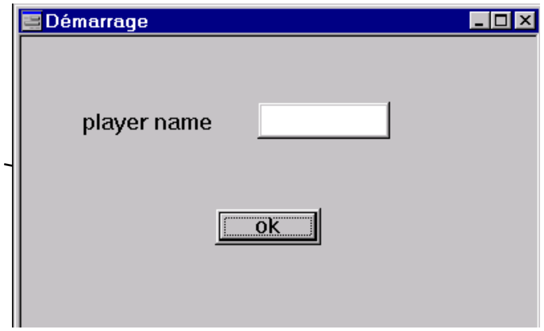


Activity Diagram

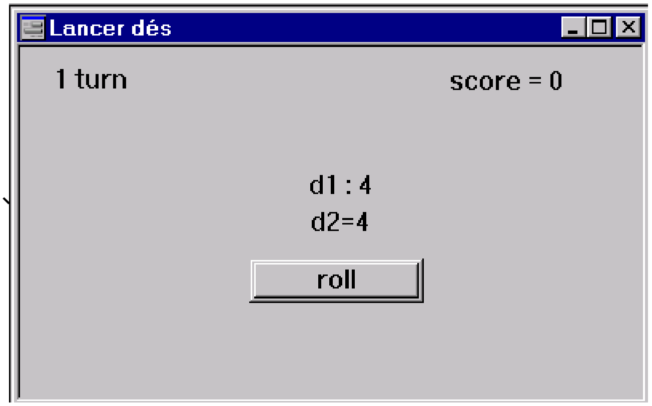




Dice GAME !



player name



1 turn

score = 0

d1 : 4

d2=4

Activity Diagram

- Requirements analysis or analysis phase?
- More business process than object
- Provide message calling sequence and detail Use-cases
- Very useful for tests...

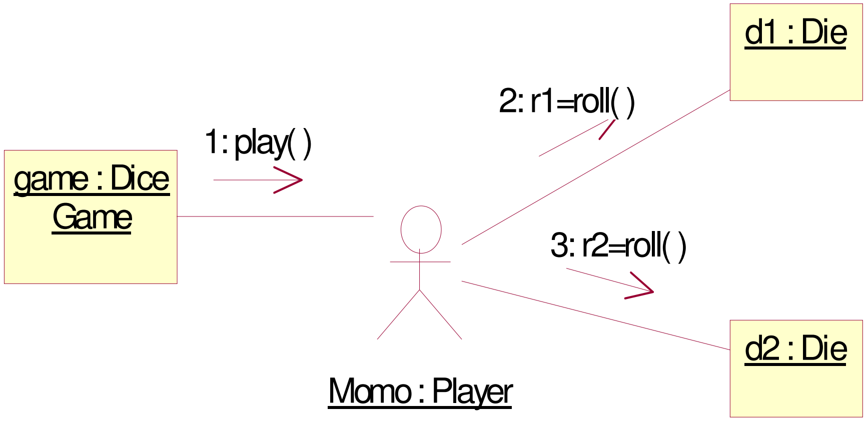
Analysis

- Model the real world
- Implementation independent
- Determine real world classes: first class diagram
- Model system's dynamics: collaboration diagram

Collaboration Diagram

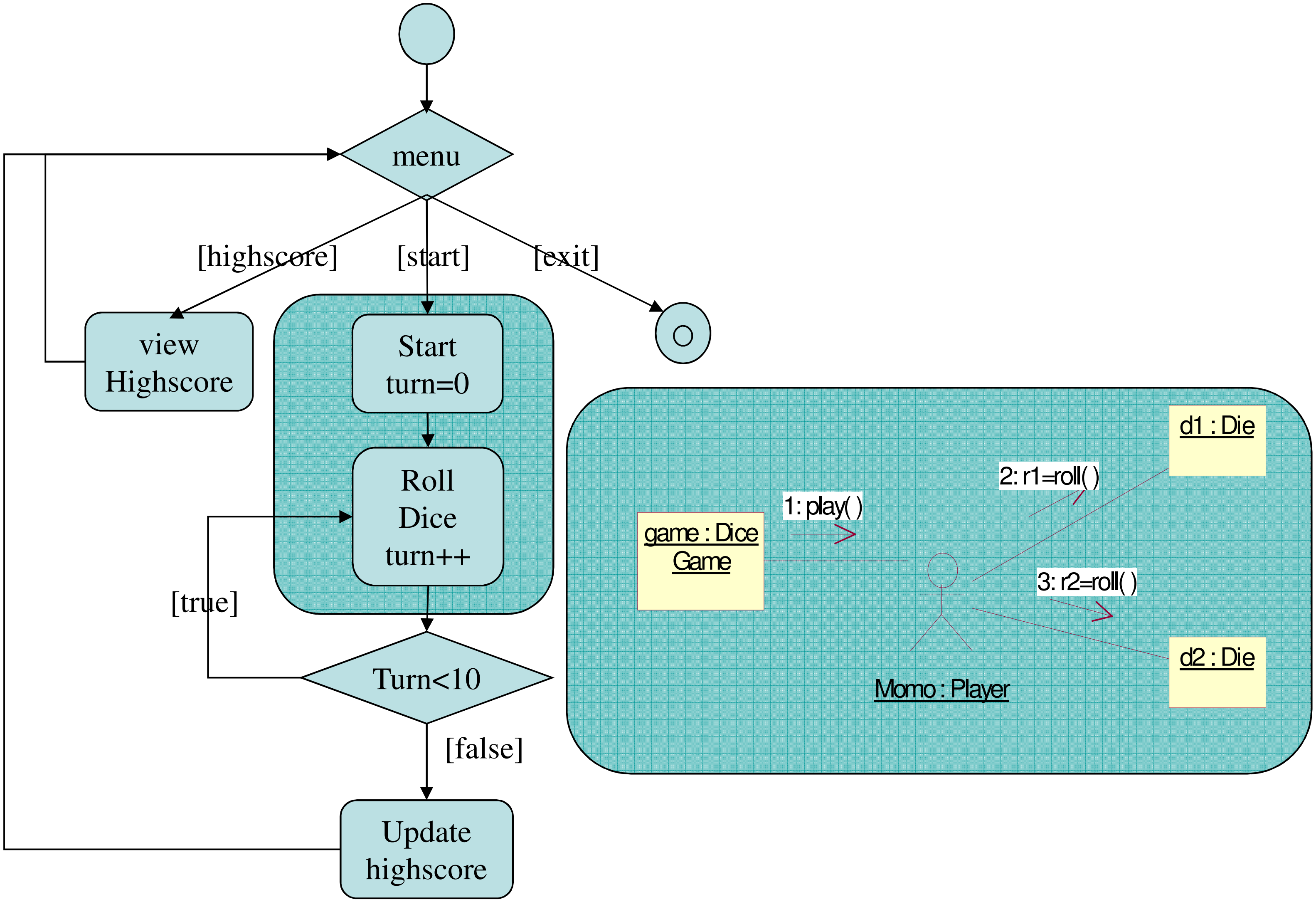
- Identify the Objects
- Identify relationships between Objects (objects graph)
- Identify messages and message calling sequence between objects

Collaboration Diagram



- Display the **objects**
- Display **relationships between objects**
- Display message **calling sequence** on individual objects

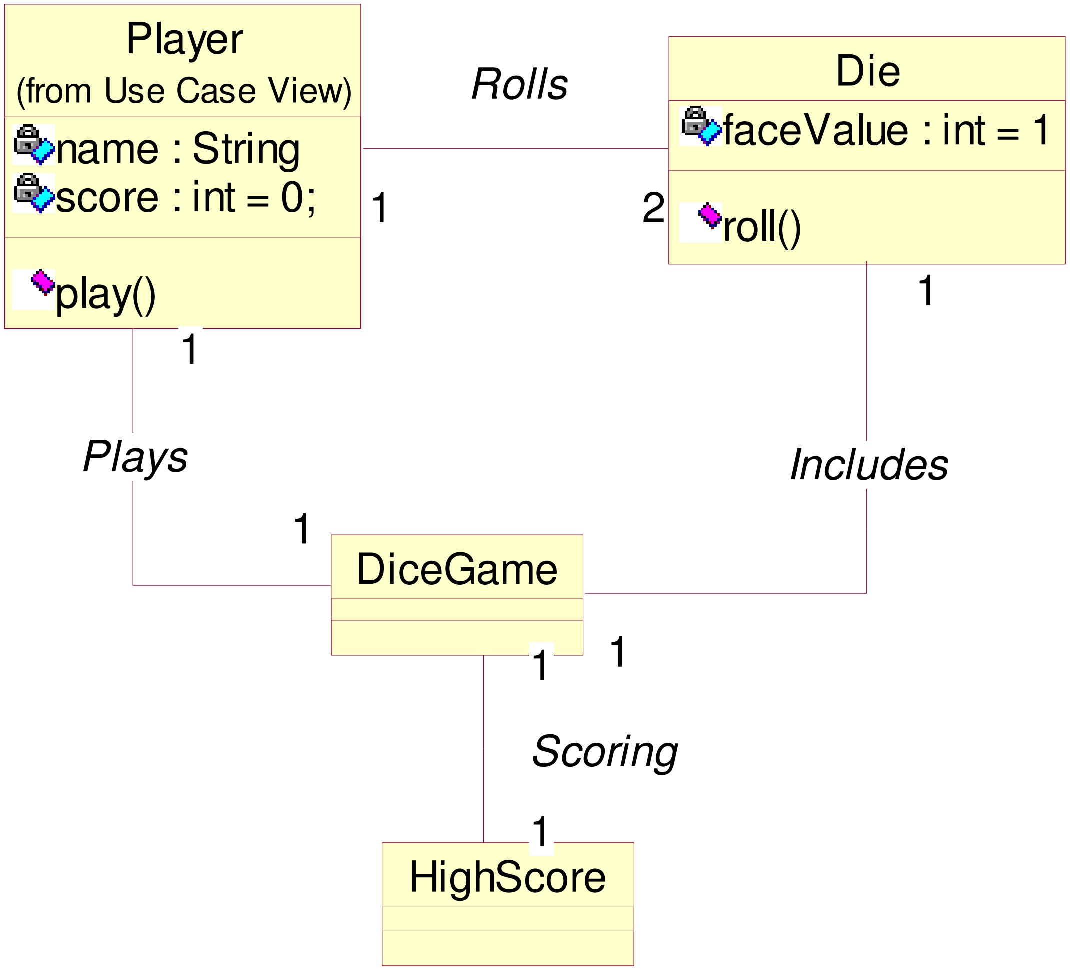
Collaboration Diagram



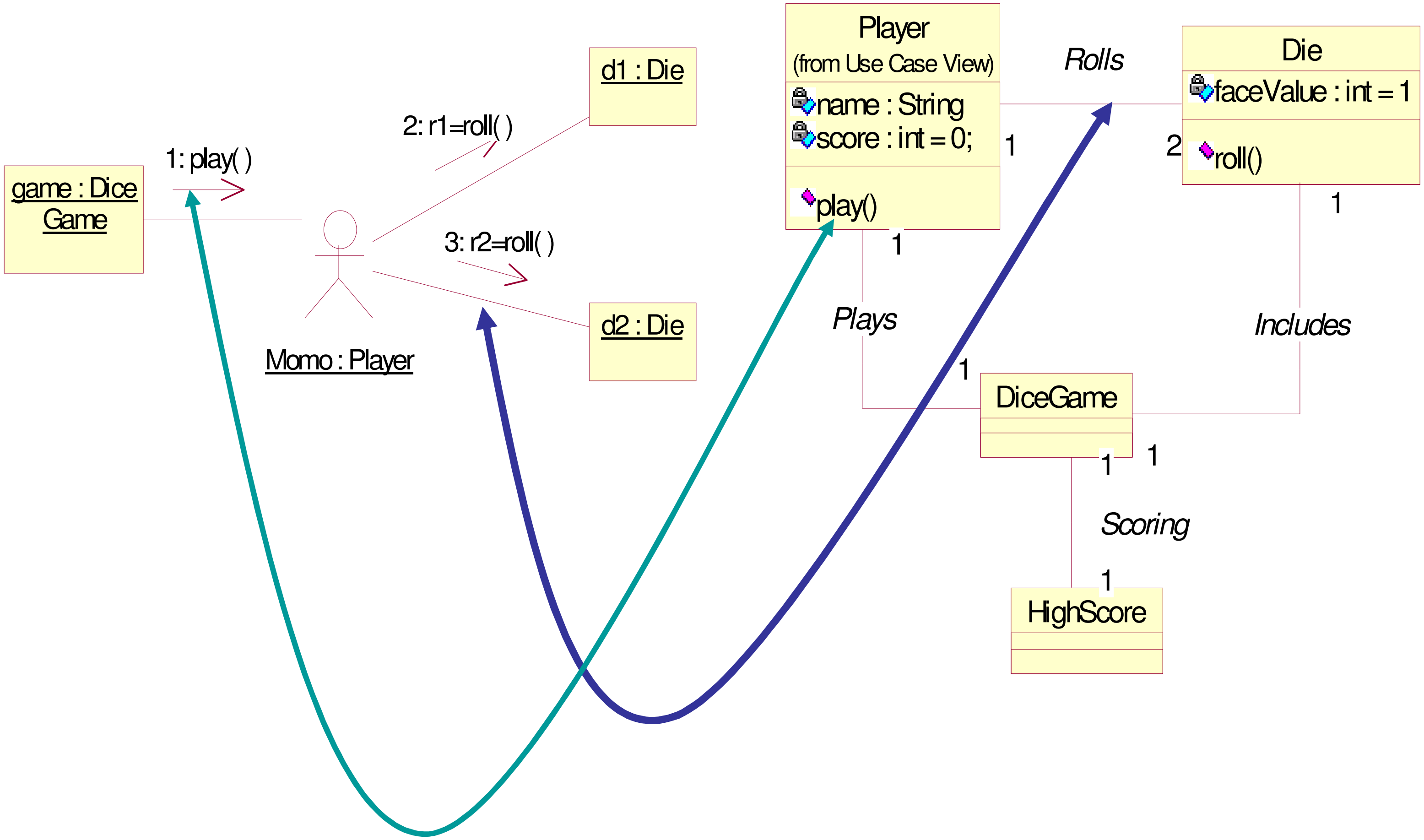
Class Diagram

- Identify classes
- Identify **static** and **dynamic** relationships between classes
- Identify relationships’ cardinality
- Identify class attributes
- Identify methods and their **parameters**

Class Diagram



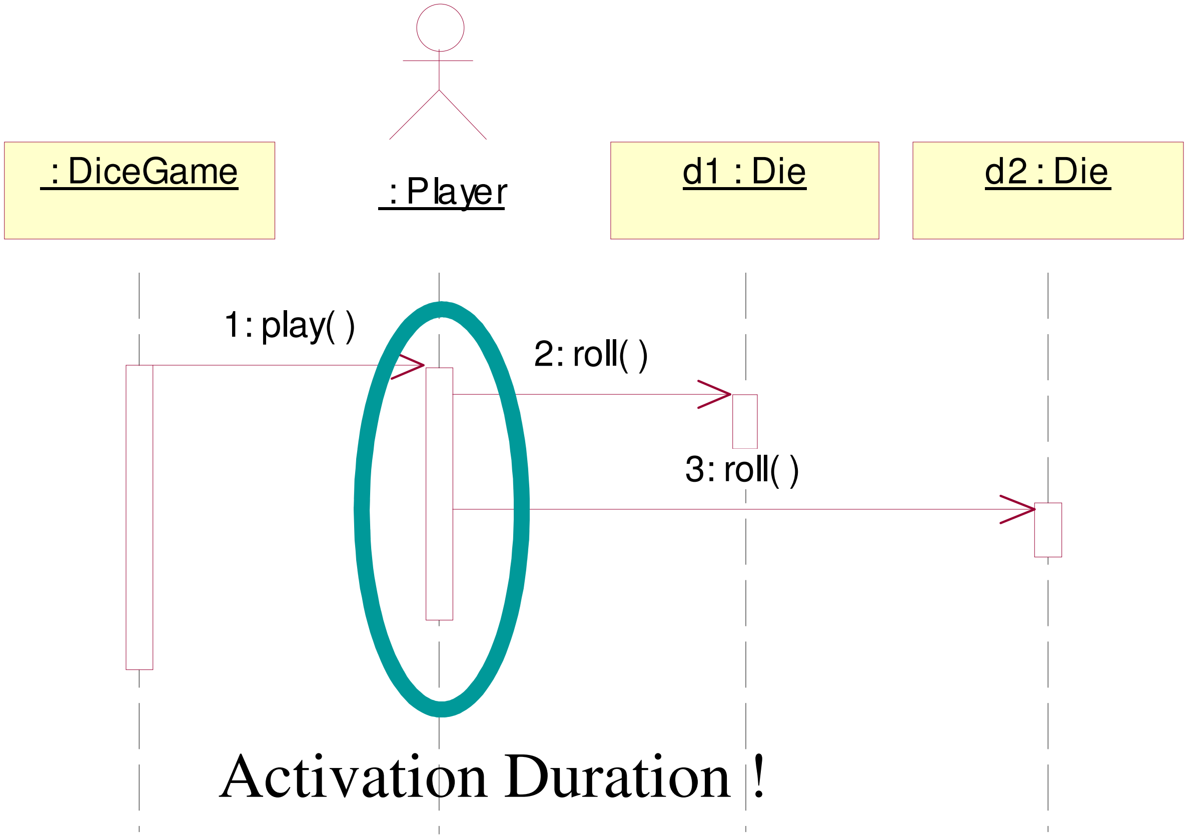
Class Diagram



Sequence Diagram

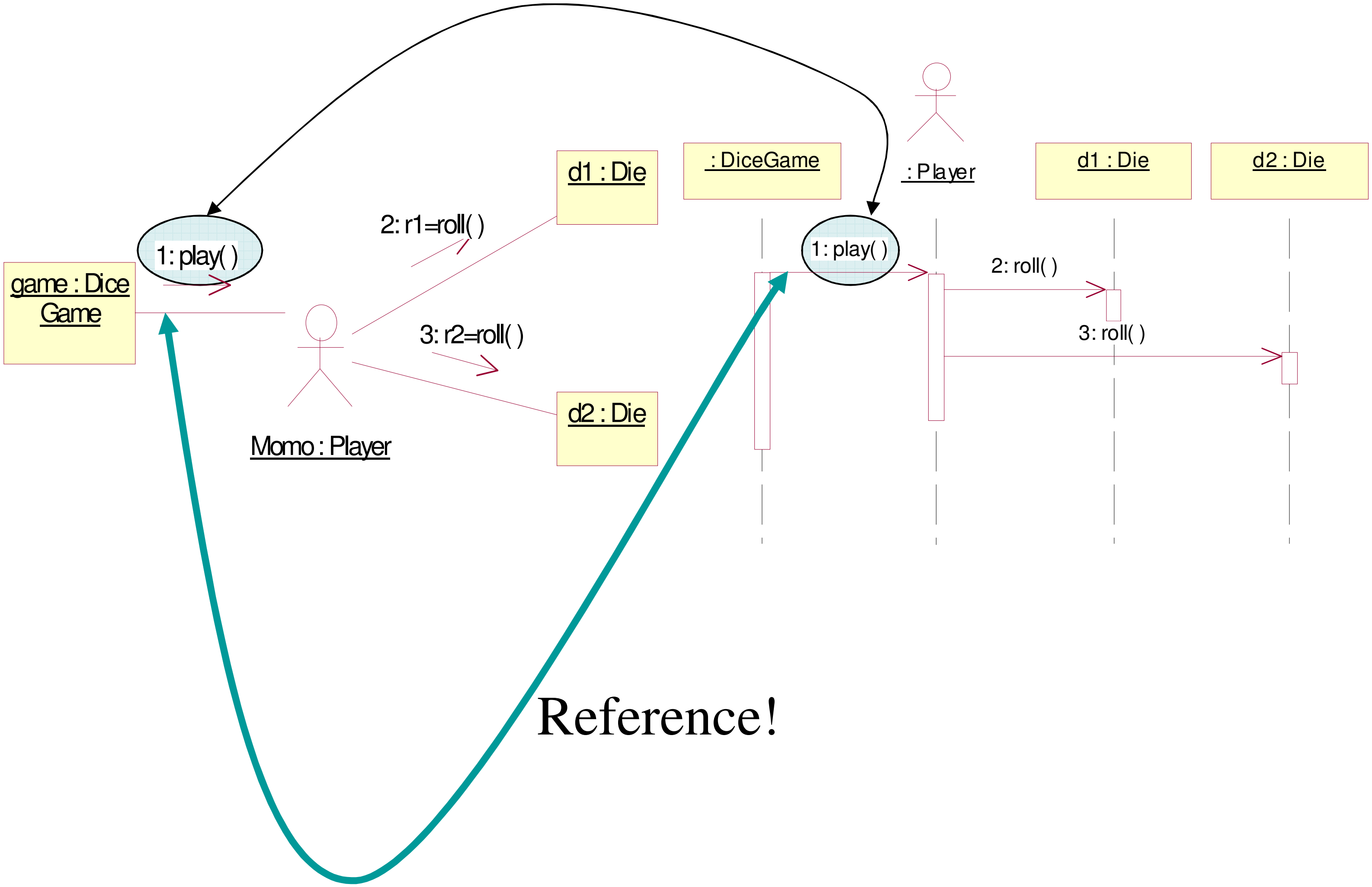
- Dynamic modeling (~same as collaboration)
- Focuses on message sequencing
- Identify objects, their messages, and the message calling sequence

Sequence Diagram



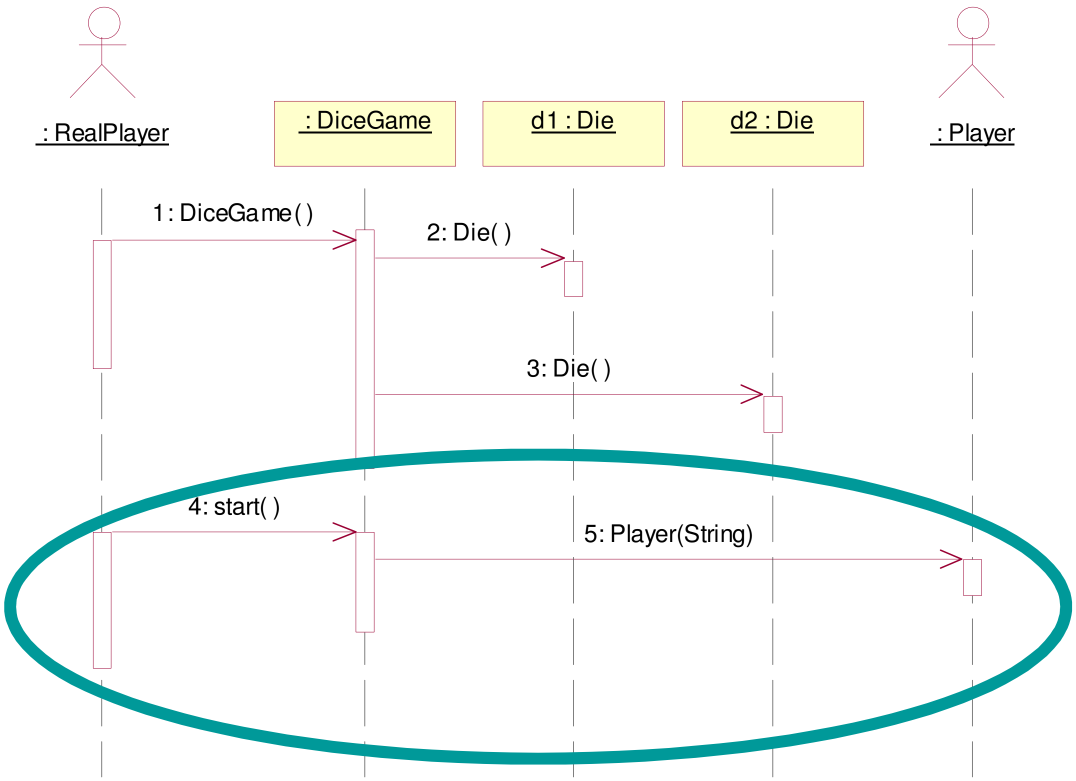
Activation Duration !

Sequence Diagram



Reference!

Sequence Diagram

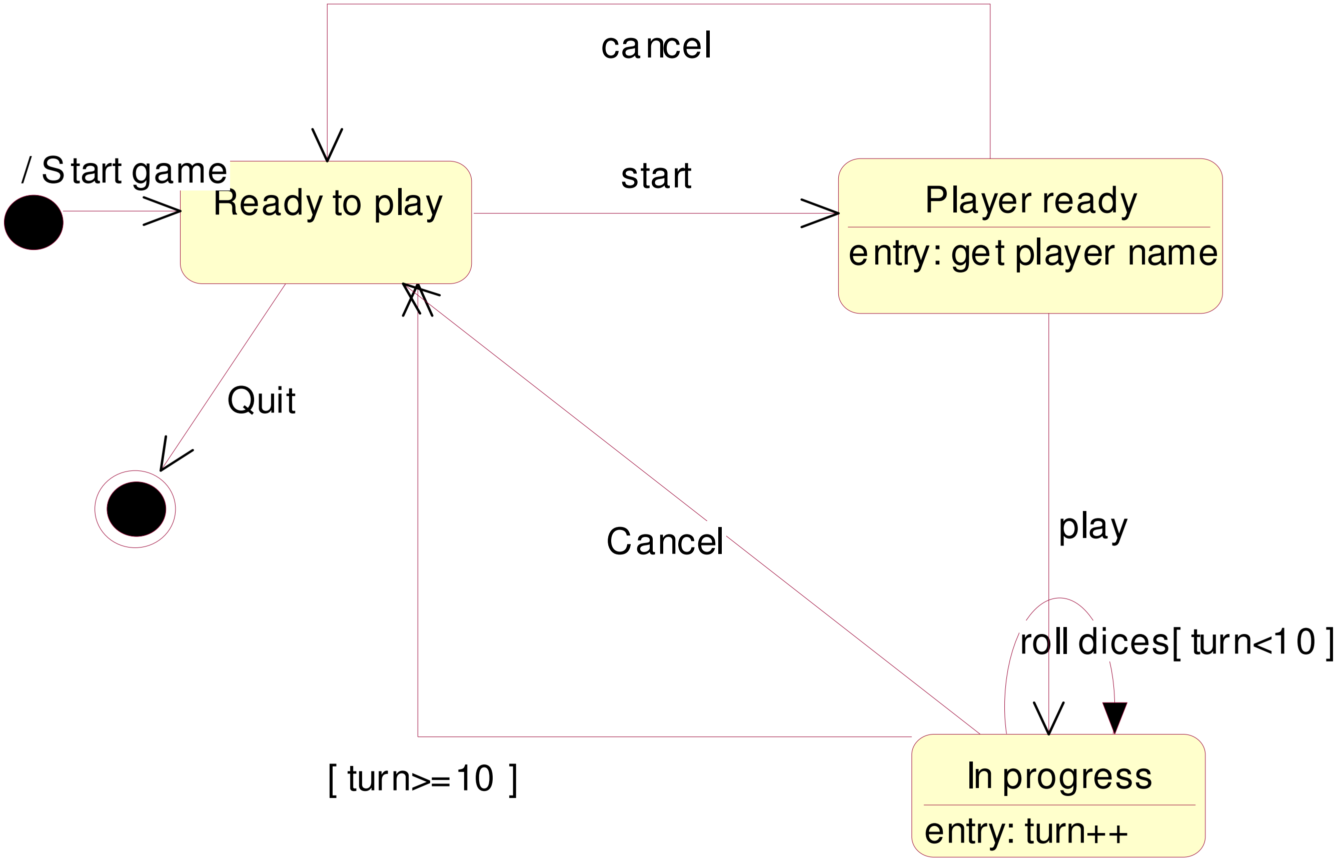


The player is only created at the beginning of the game !

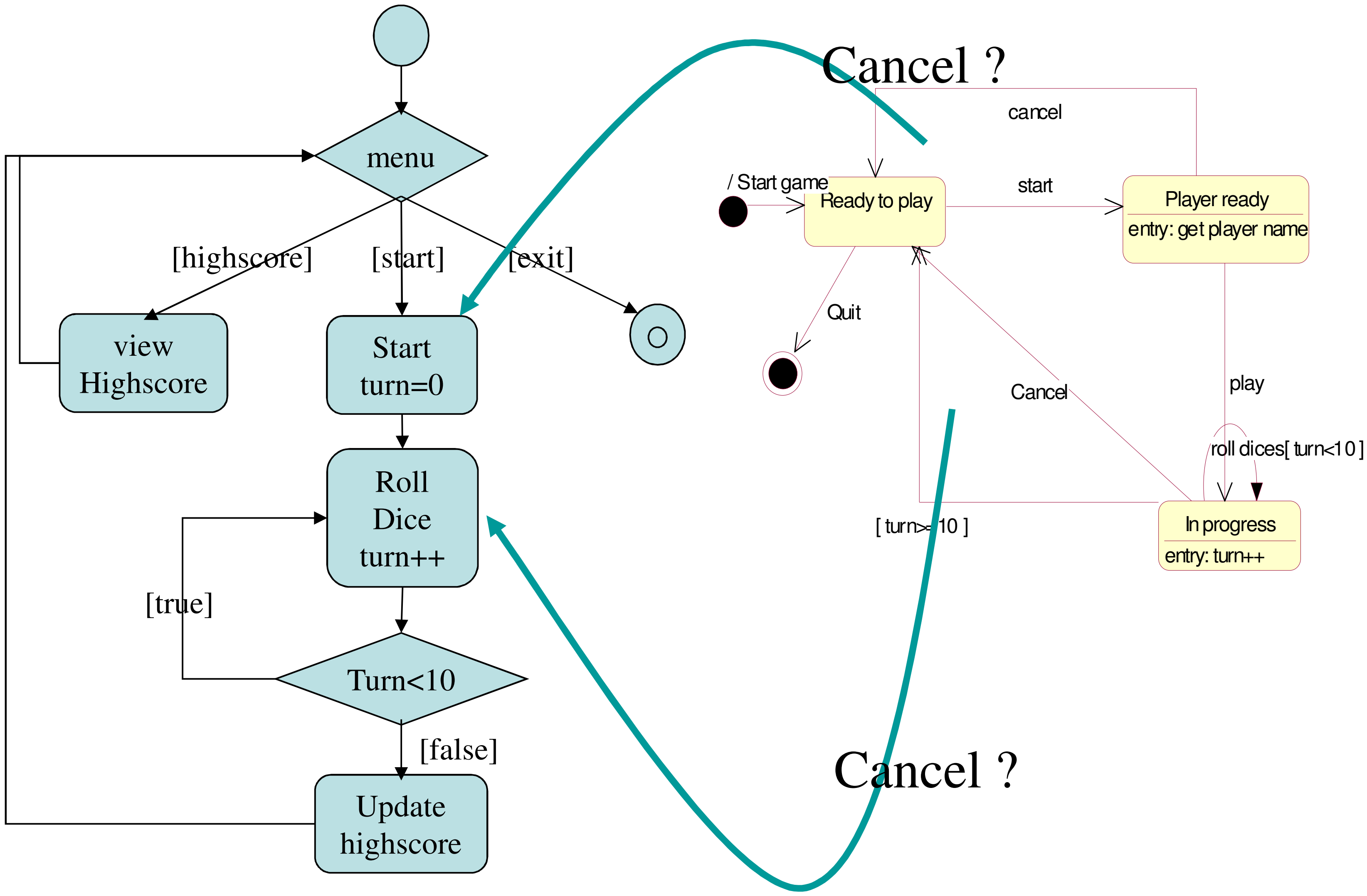
State Diagram

- Identify the states of **an object**
- Identify state transitions

State Diagram for a « game » object



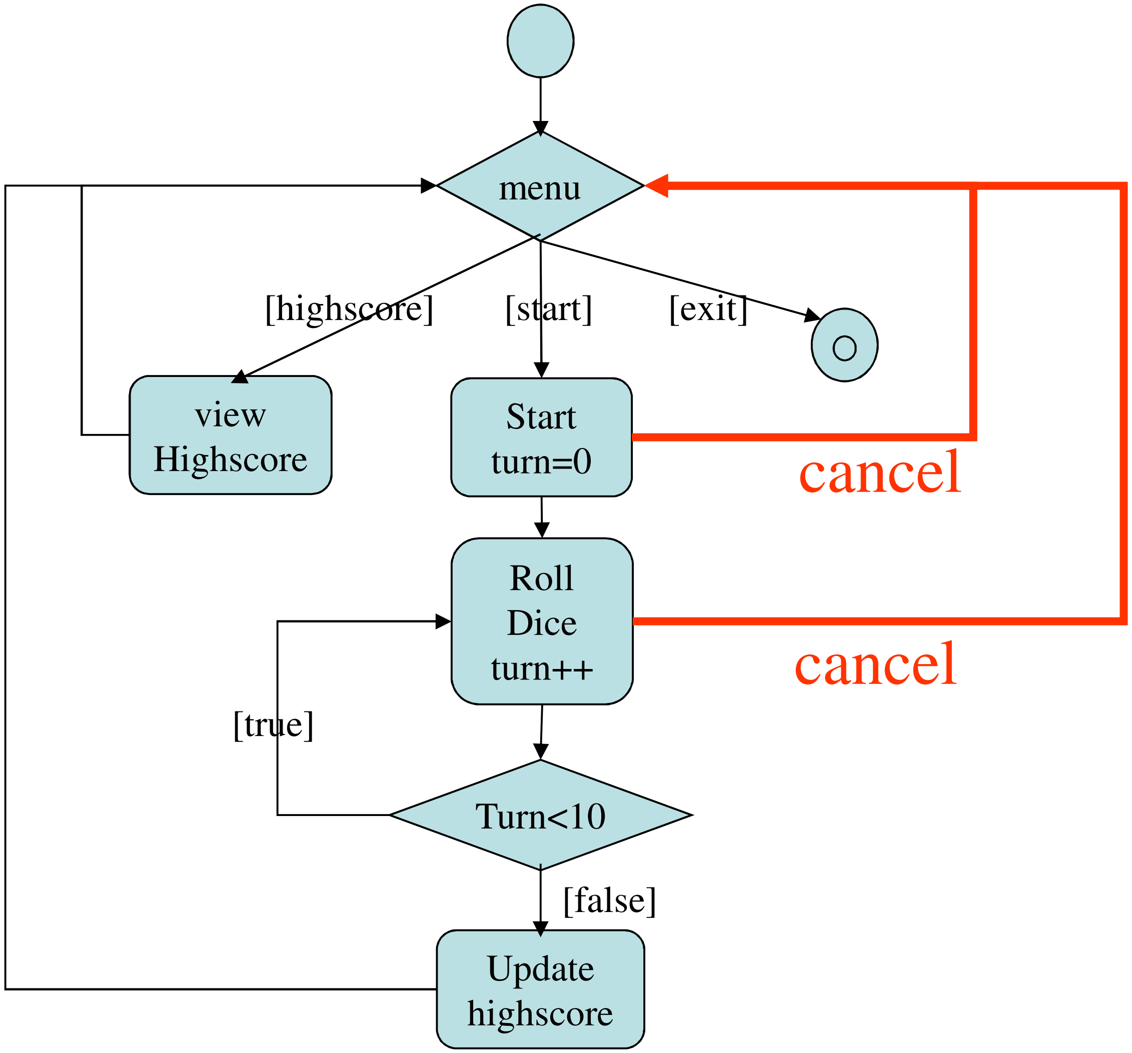
State Diagram



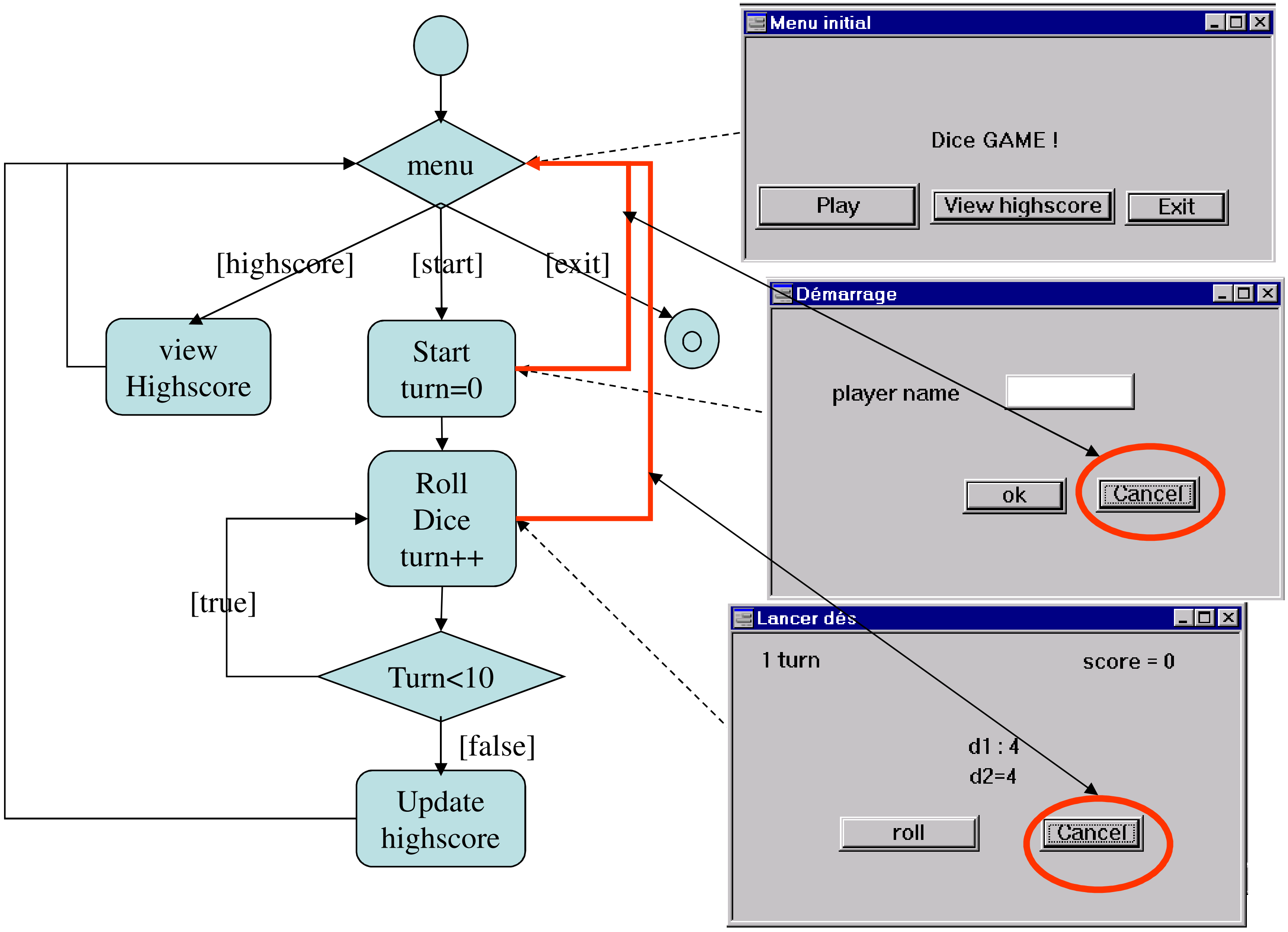
Cancel ?

Cancel ?

Modify Schema...



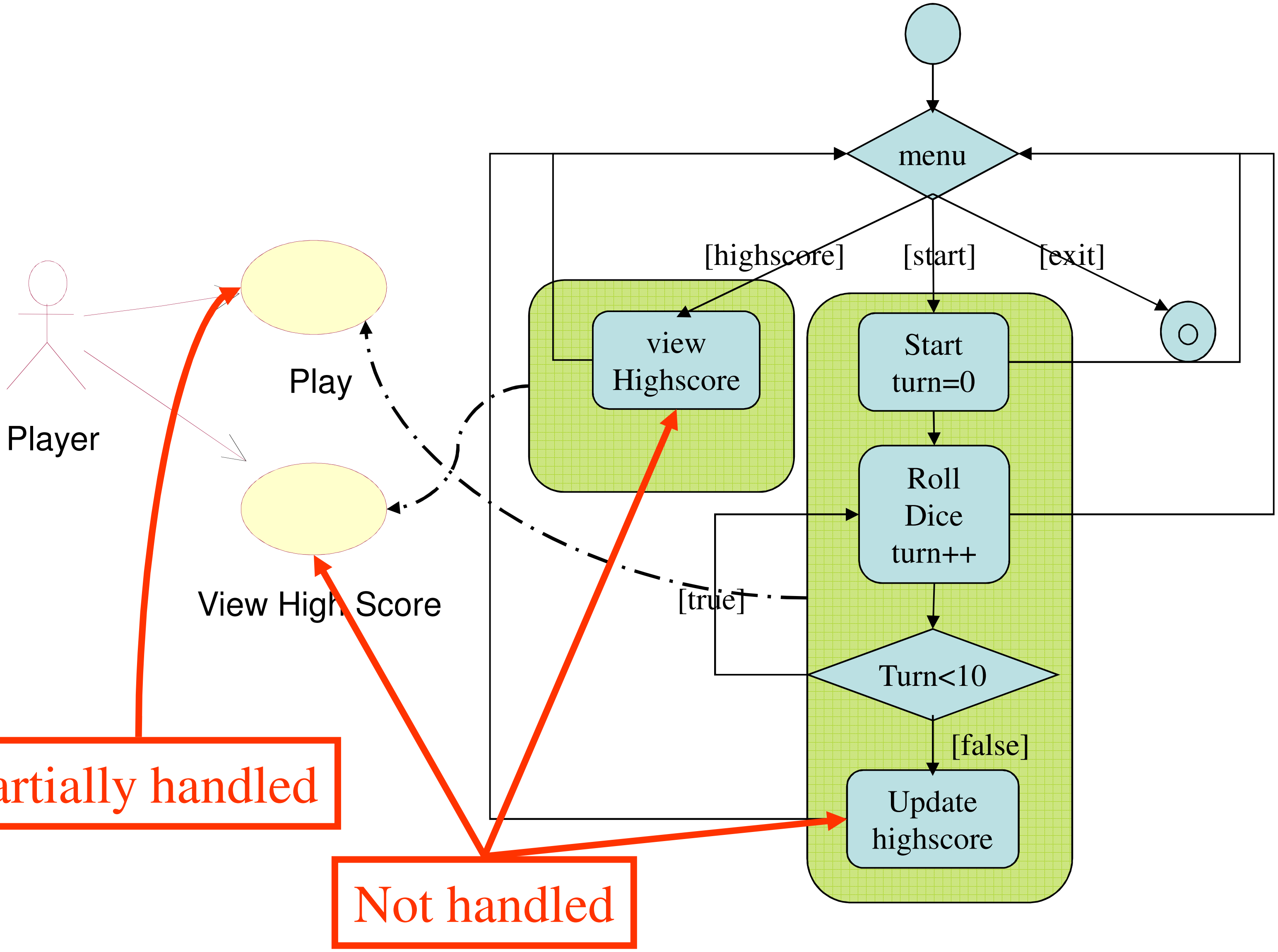
Modify Schemas ?



End of Analysis ?

- Verify coverage for use-case and activity diagrams…
- Use case « view highscores » ?
- Use case « play » partially handled

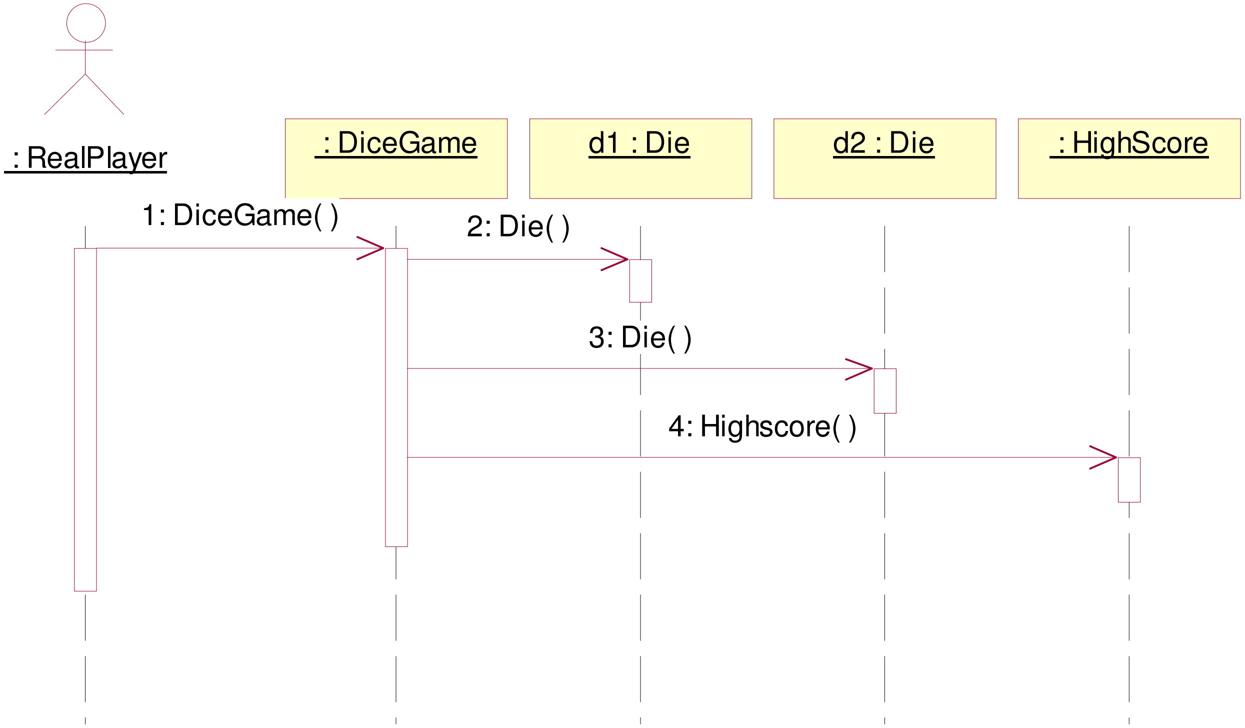
Diagram Coverage



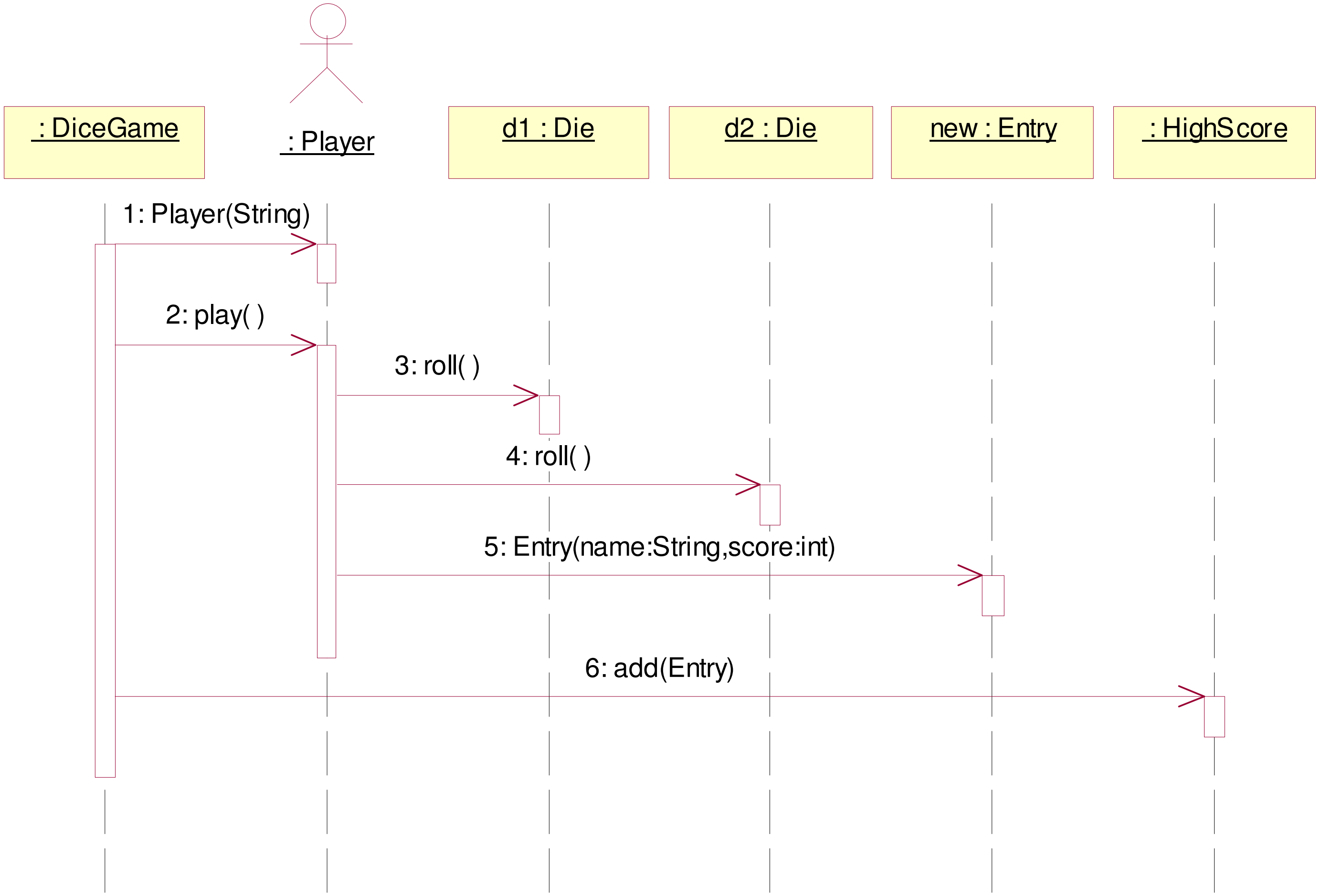
Partially handled

Not handled

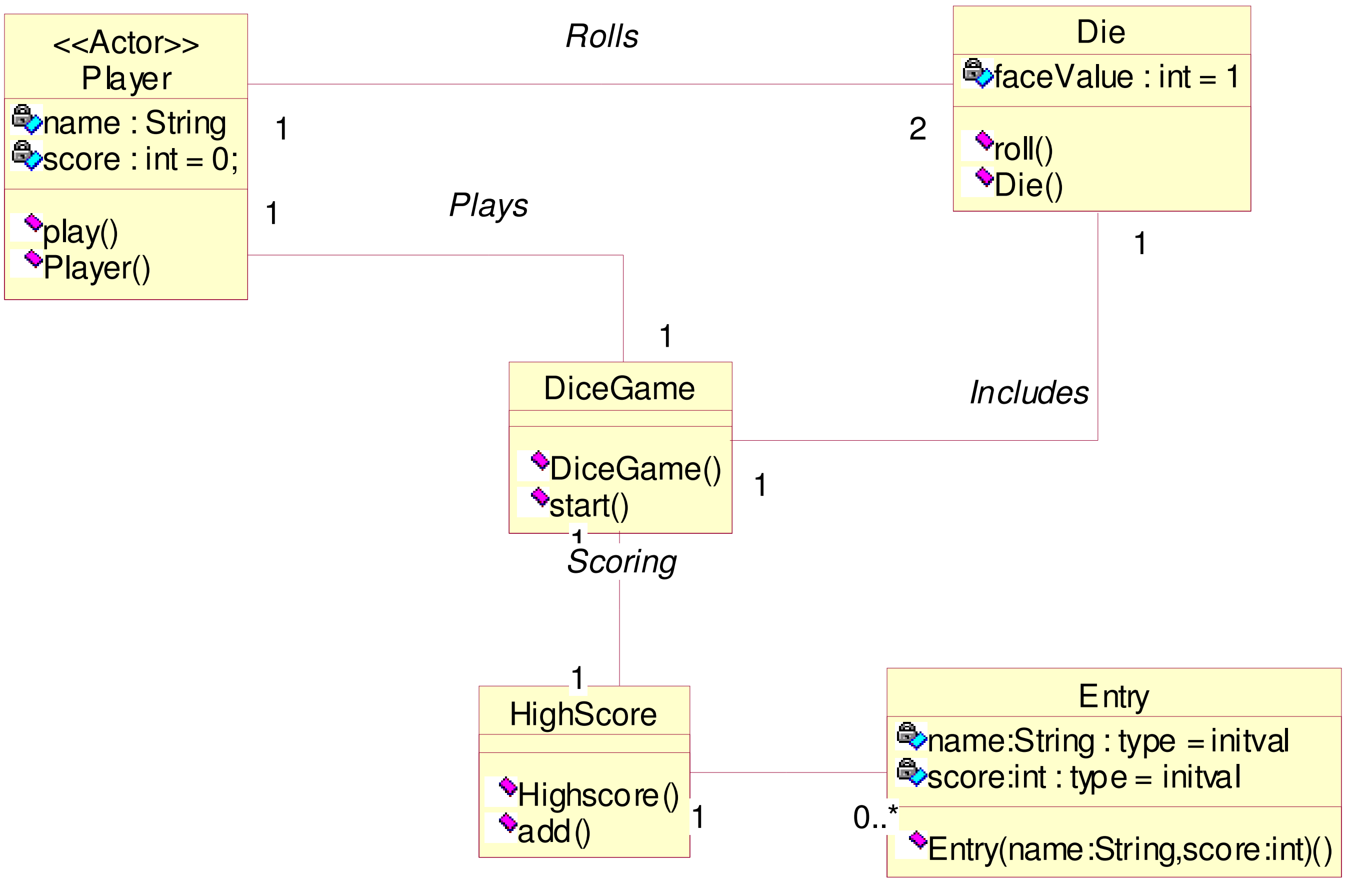
Sequence Diagram



Sequence Diagram



Class Diagram



<<Actor>>
Player
name : String
score : int = 0;
play()
Player()

Rolls

1

2

Die
faceValue : int = 1
roll()
Die()

Plays

1

1

1

DiceGame
DiceGame()
start()

Includes

1

1

Scoring

1

HighScore
Highscore()
add()

1

0..*

Entry
name:String : type = initval
score:int : type = initval
Entry(name:String,score:int)()

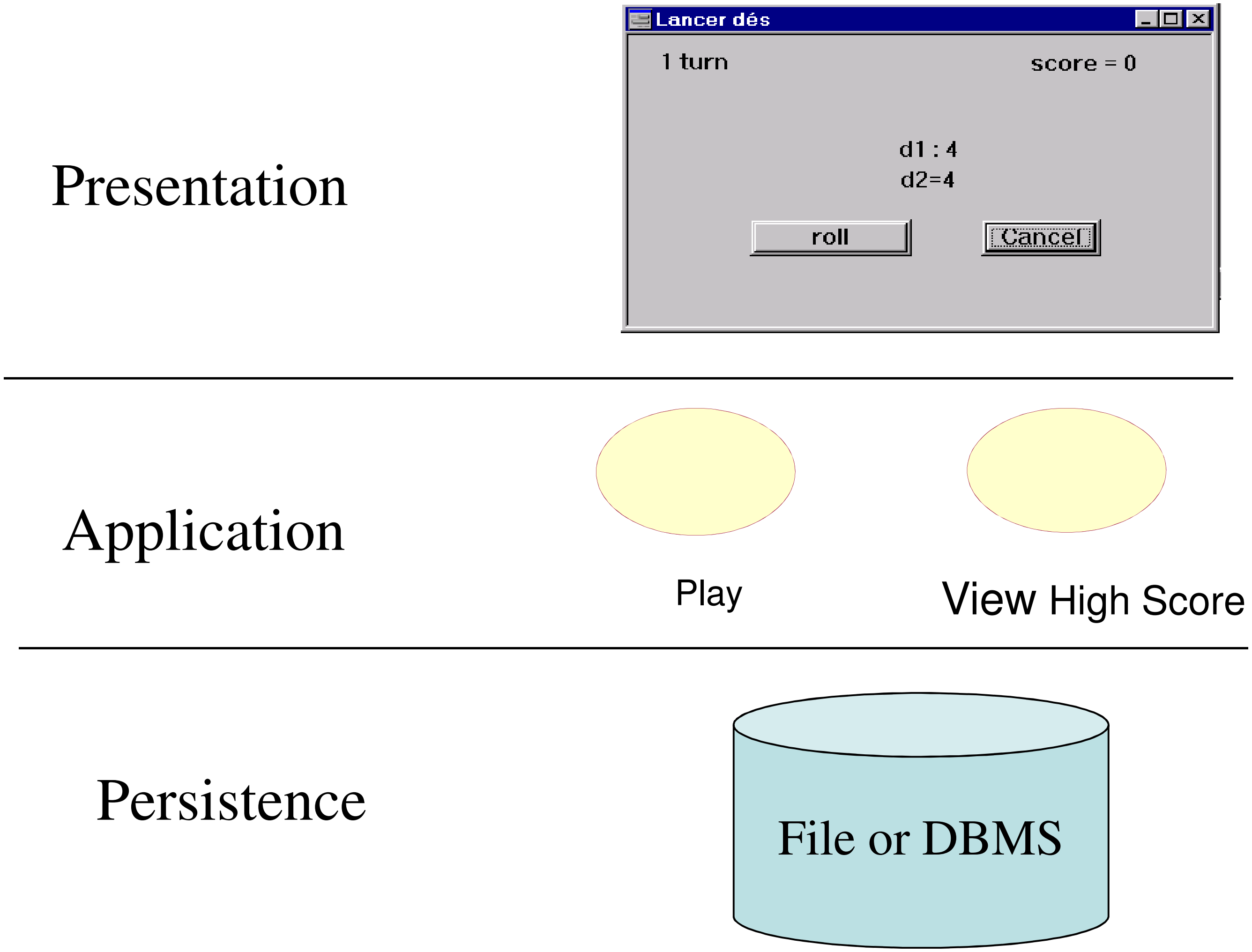
End of Analysis ?

- Coverage is « pretty » good
- Consistency across schemas is correct
- 14/20
  - » Dynamic model lacks detail (dynamic model for cancel?)
  - » Schemas are not described in enough detail…
  - » Sequence diagrams for the game are not detailed enough : a few methods are missing…

Design

- Take implementation into account
  - Handle the GUI portion
  - Handle highscores persistence
- Define a logical architecture
- Define a physical architecture
- Add technical classes allowing the implementation of such architecture !

Architecture Design



Presentation

Lancer dés

1 turn

score = 0

d1 : 4

d2=4

Application

Play

View High Score

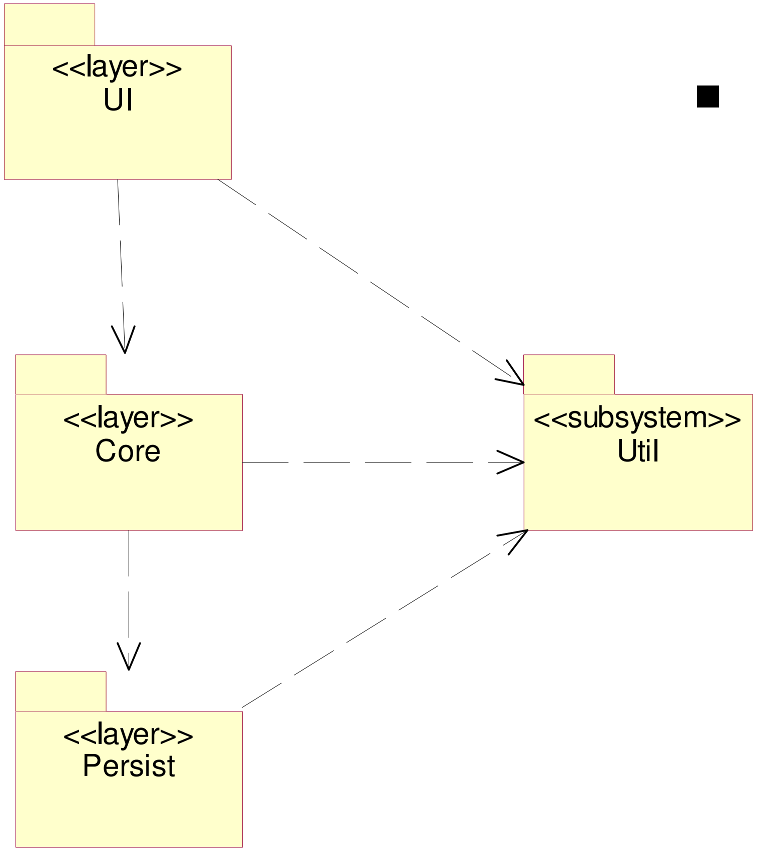
Persistence

File or DBMS

Layered Architecture...

- One possible architecture, others exist (seeo « A system of patterns » Bushcmann »)
- Layers must be as independent as possible
- « Separate » layers by relying on interfaces + abstract classes (design patterns)

Logical « packaging »

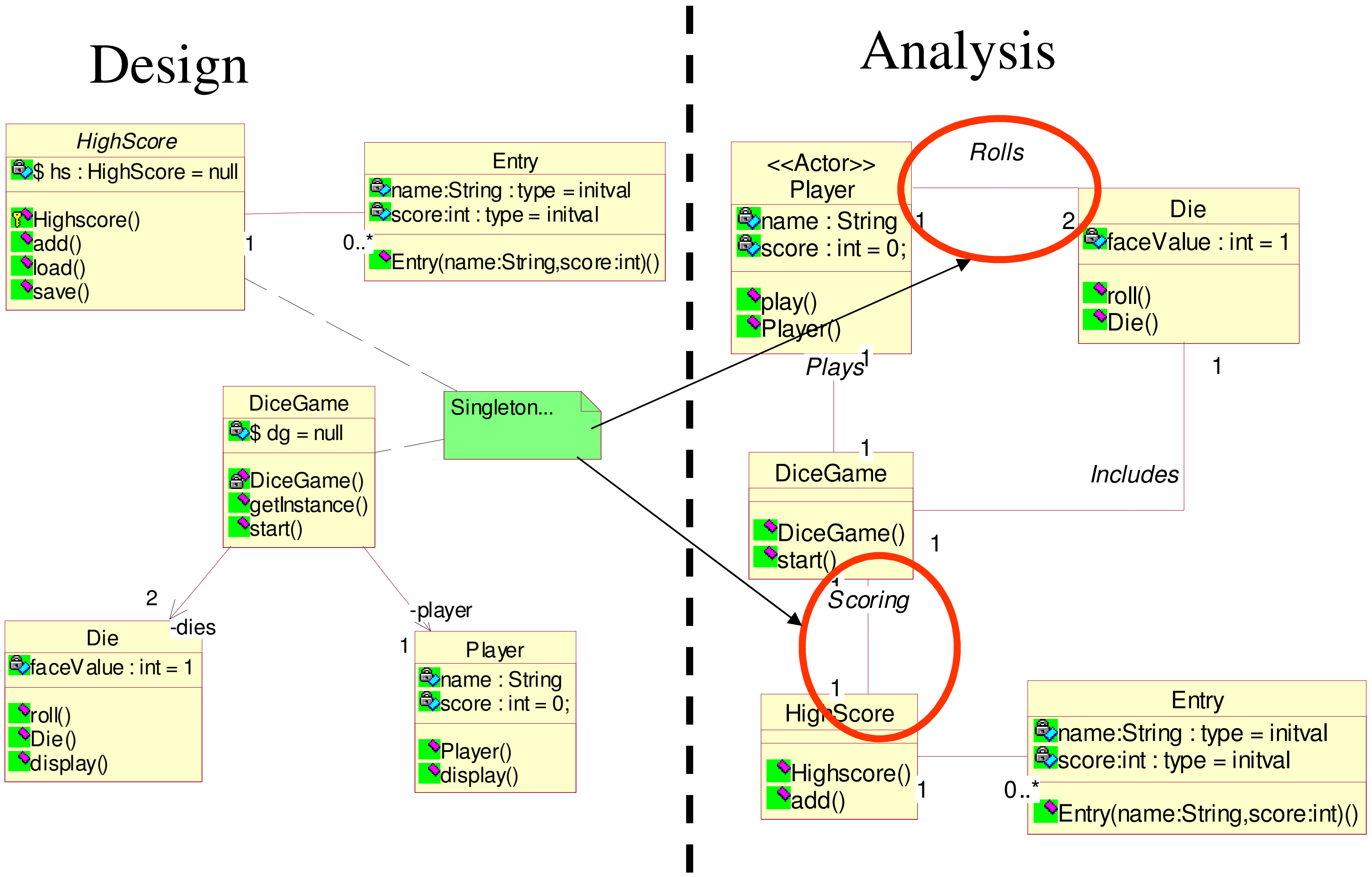


- Map the architecture on « layered » packages
- Express dependencies

« core » Layer

- Classes representing the application logic
- In fact, these are the analysis classes which are being revisited for realization purpose

Core « Layer »:First Diagram



Design

HighScore
$ hs : HighScore = null
Highscore()
add()
load()
save()

1

0..*

Entry
name:String : type = initval
score:int : type = initval
Entry(name:String,score:int)()

Singleton...

DiceGame
$ dg = null
DiceGame()
getInstance()
start()

2
-dies

Die
faceValue : int = 1
roll()
Die()
display()

-player
1

Player
name : String
score : int = 0;
Player()
display()

Analysis

<<Actor>>
Player
name : String
score : int = 0;
play()
Player()

Rolls
1
2

Die
faceValue : int = 1
roll()
Die()

Plays
1
1

1
Includes
1

DiceGame
DiceGame()
start()

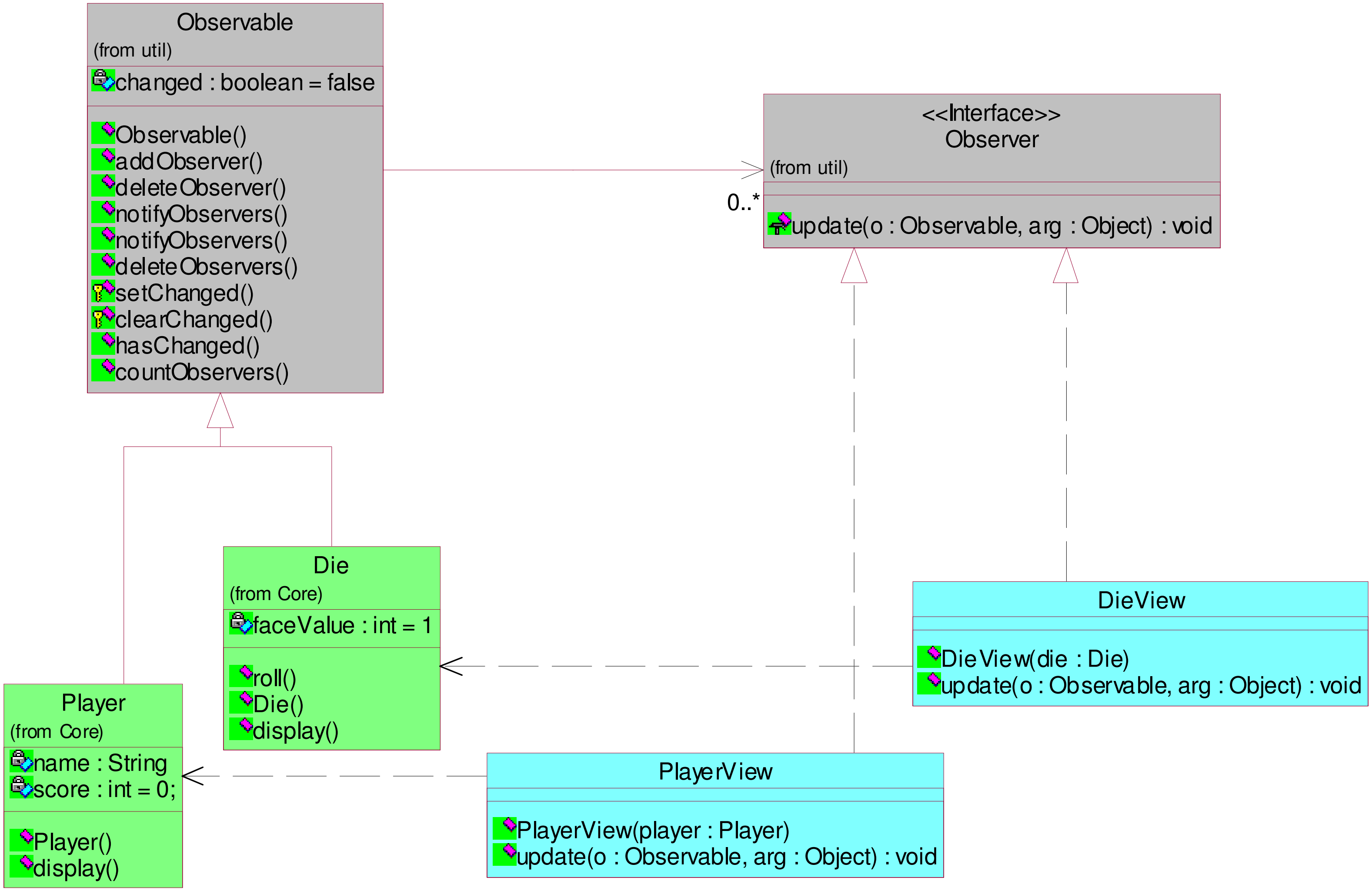
1
Scoring
1

HighScore
Highscore()
add()

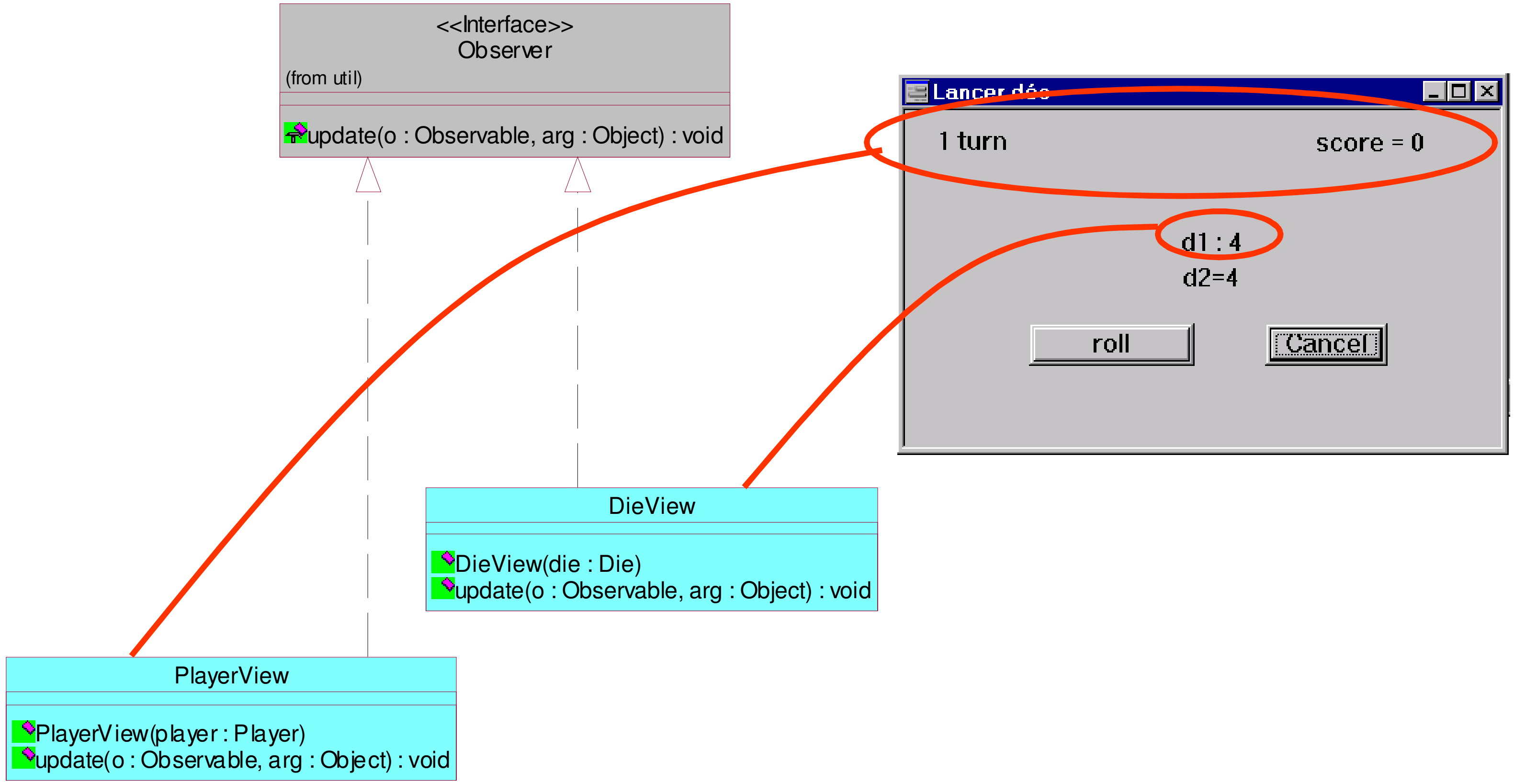
1
0..*

Entry
name:String : type = initval
score:int : type = initval
Entry(name:String,score:int)()

Graphical Interface Layering: MVC



Views ?



<<Interface>>
Observer
(from util)

update(o : Observable, arg : Object) : void

DieView

DieView(die : Die)
update(o : Observable, arg : Object) : void

PlayerView

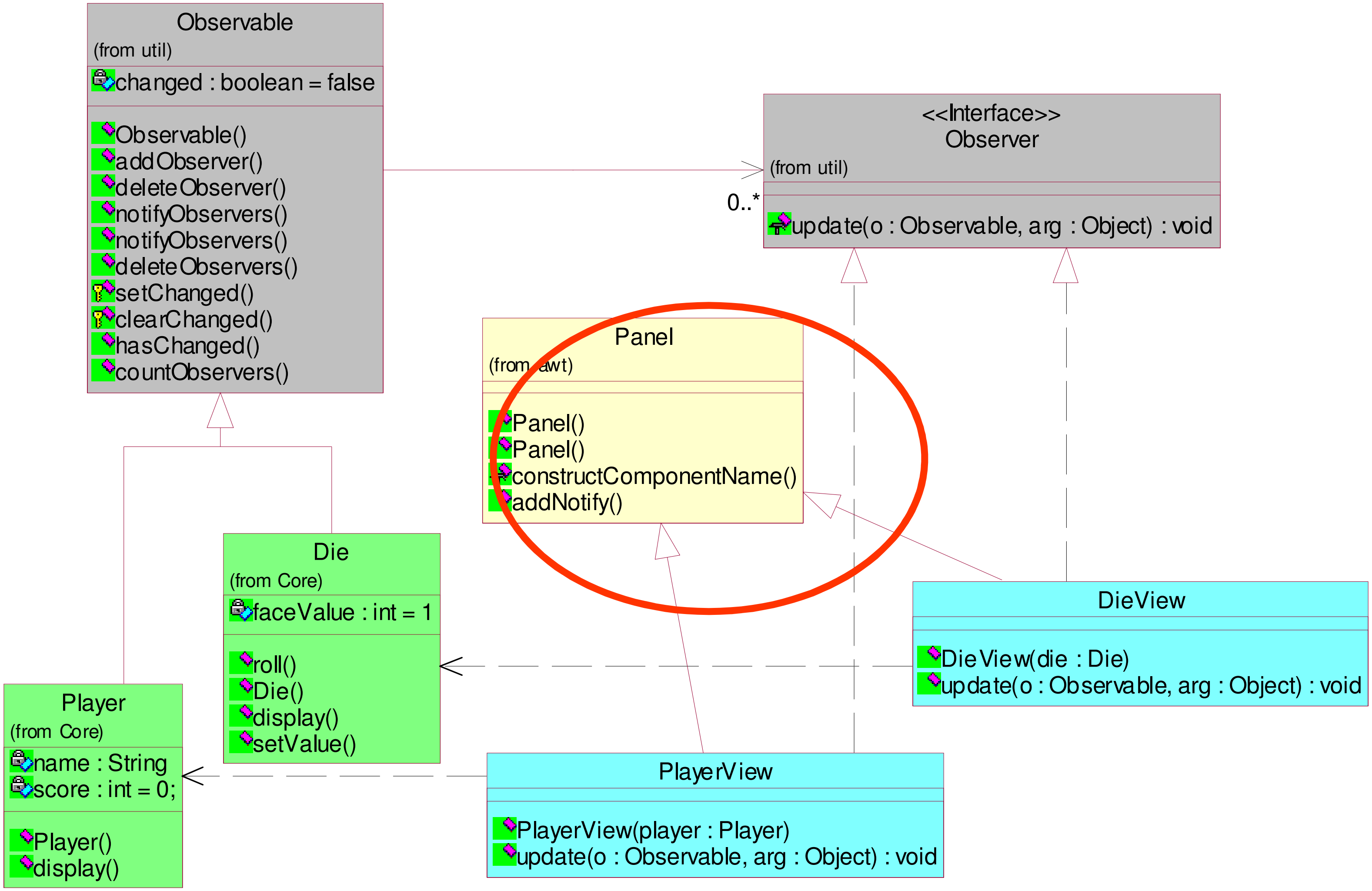
PlayerView(player : Player)
update(o : Observable, arg : Object) : void

Lancer dés

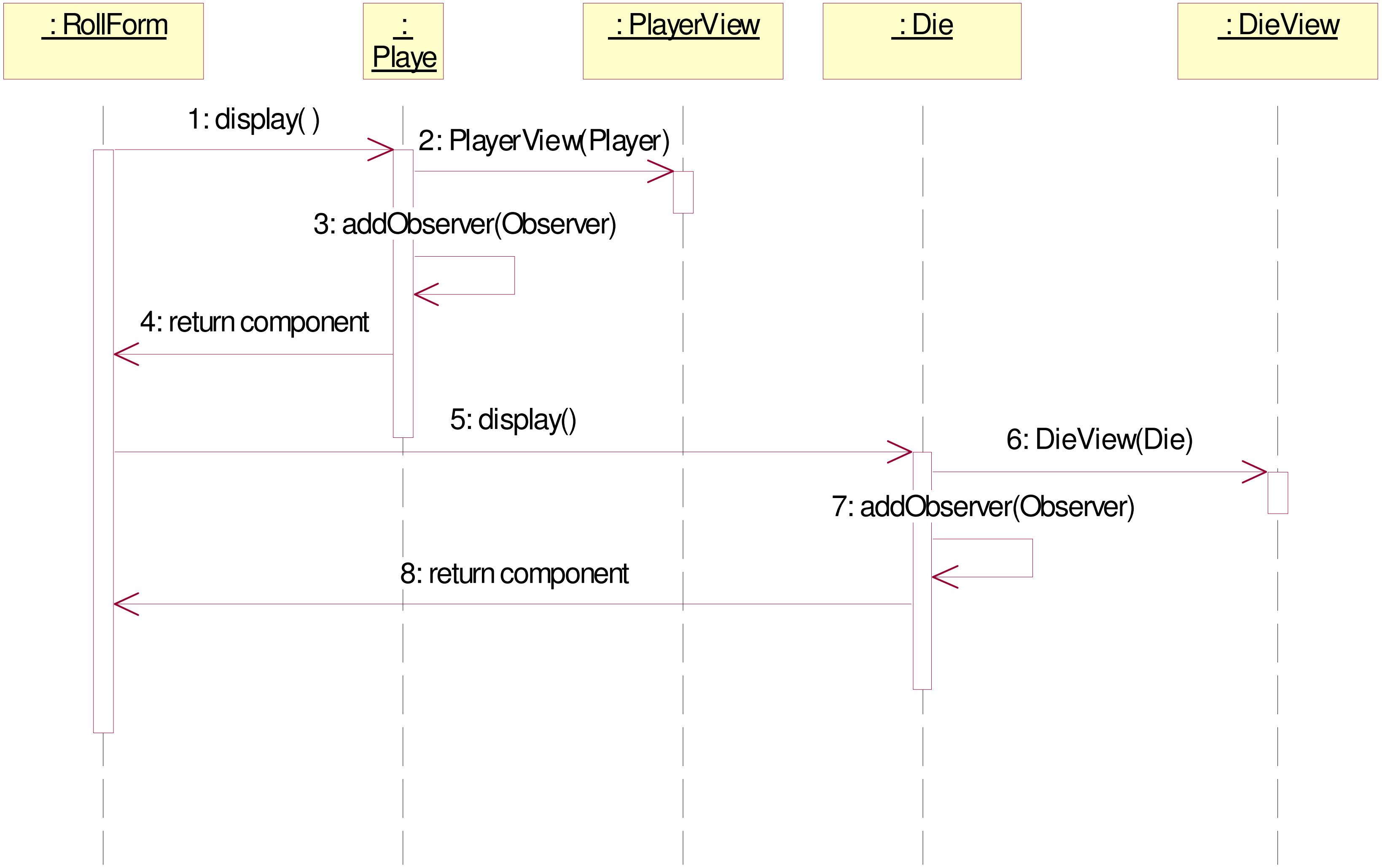
1 turn score = 0

d1 : 4
d2=4

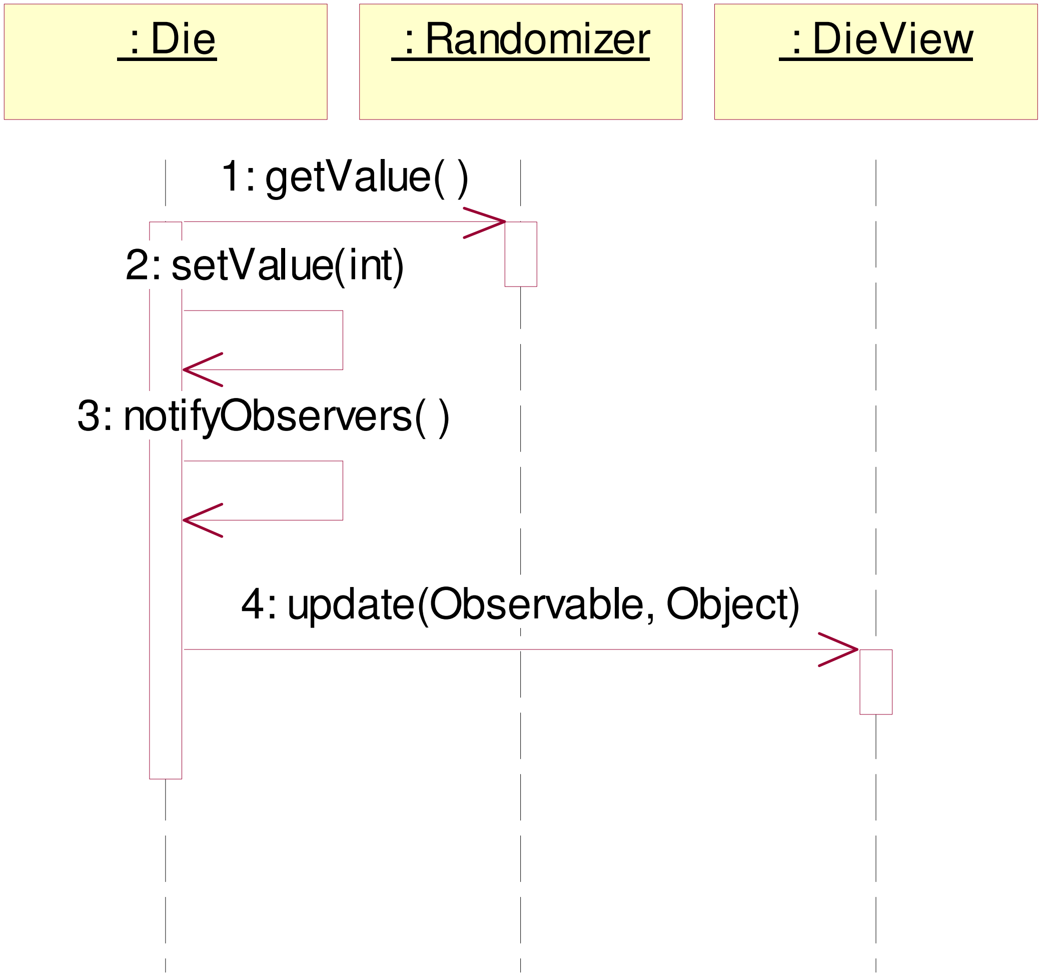
Attention ...


Observable
(from util)
changed : boolean = false
Observable()
addObserver()
deleteObserver()
notifyObservers()
notifyObservers()
deleteObservers()
setChanged()
clearChanged()
hasChanged()
countObservers()
<<Interface>>
Observer
(from util)
0..*
update(o : Observable, arg : Object) : void
Panel
(from awt)
Panel()
Panel()
constructComponentName()
addNotify()
Die
(from Core)
faceValue : int = 1
roll()
Die()
display()
setValue()
DieView
DieView(die : Die)
update(o : Observable, arg : Object) : void
Player
(from Core)
name : String
score : int = 0;
Player()
display()
PlayerView
PlayerView(player : Player)
update(o : Observable, arg : Object) : void

MVC in action: 1 Setup



MVC in action: 2 state change



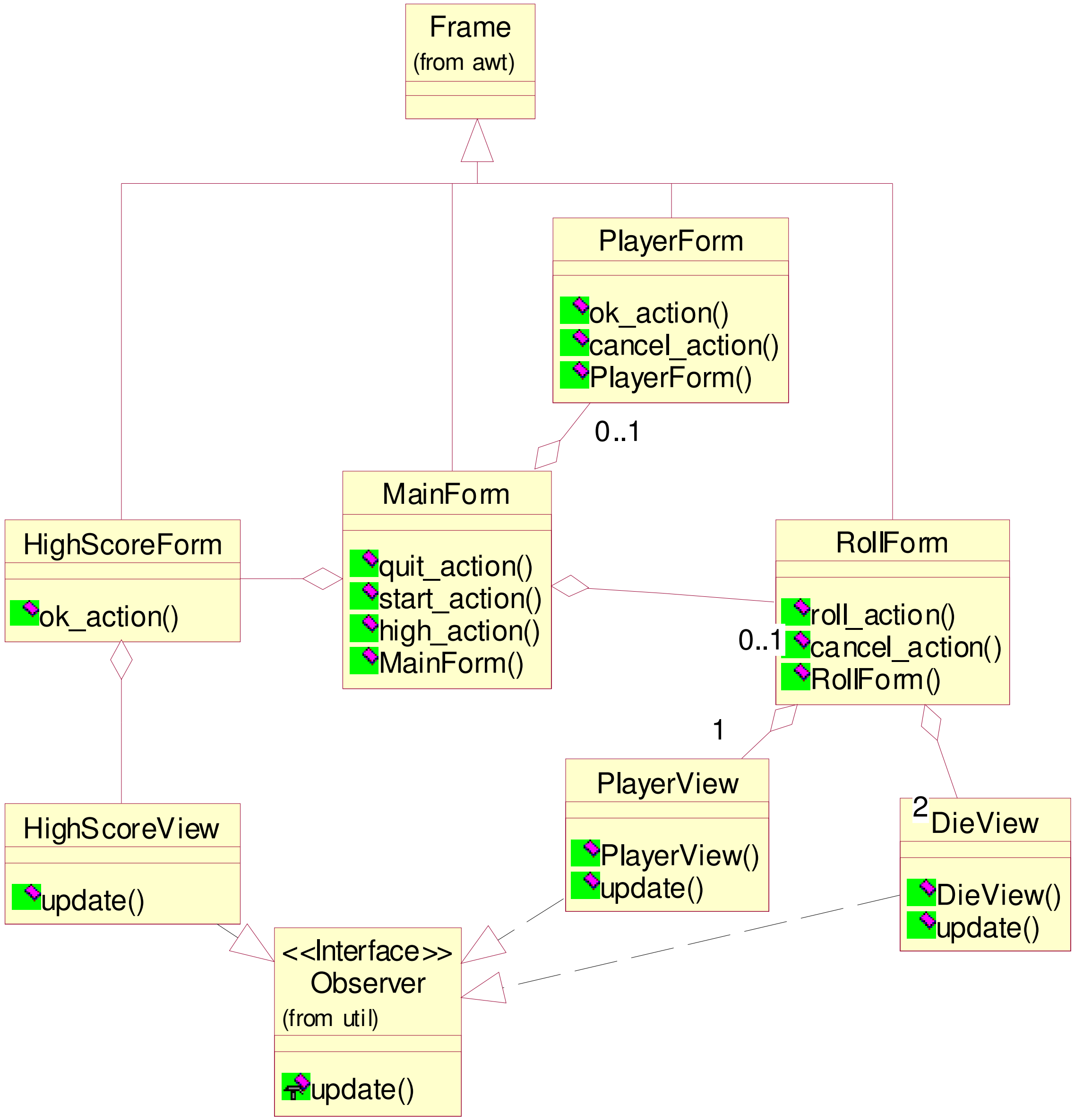
MVC

- Java AWT Java: Delegation Model
  - » Event propagation from user interface to core application classes
- MVC:
  - » State change propagation to graphical objects

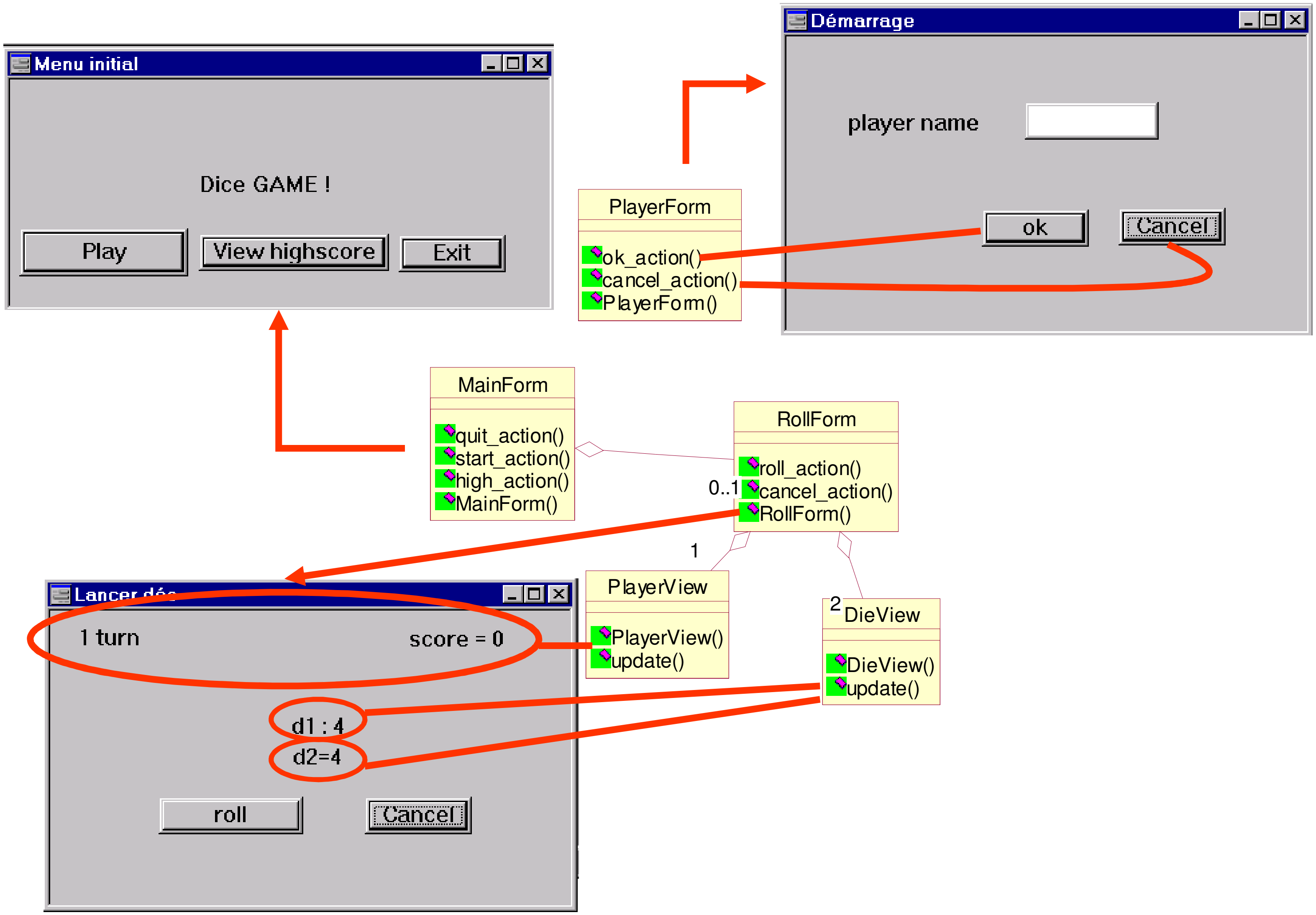
Put the views in the « forms »

- Implement graphical interfaces containing views as needed…
- The UI « layer » ...

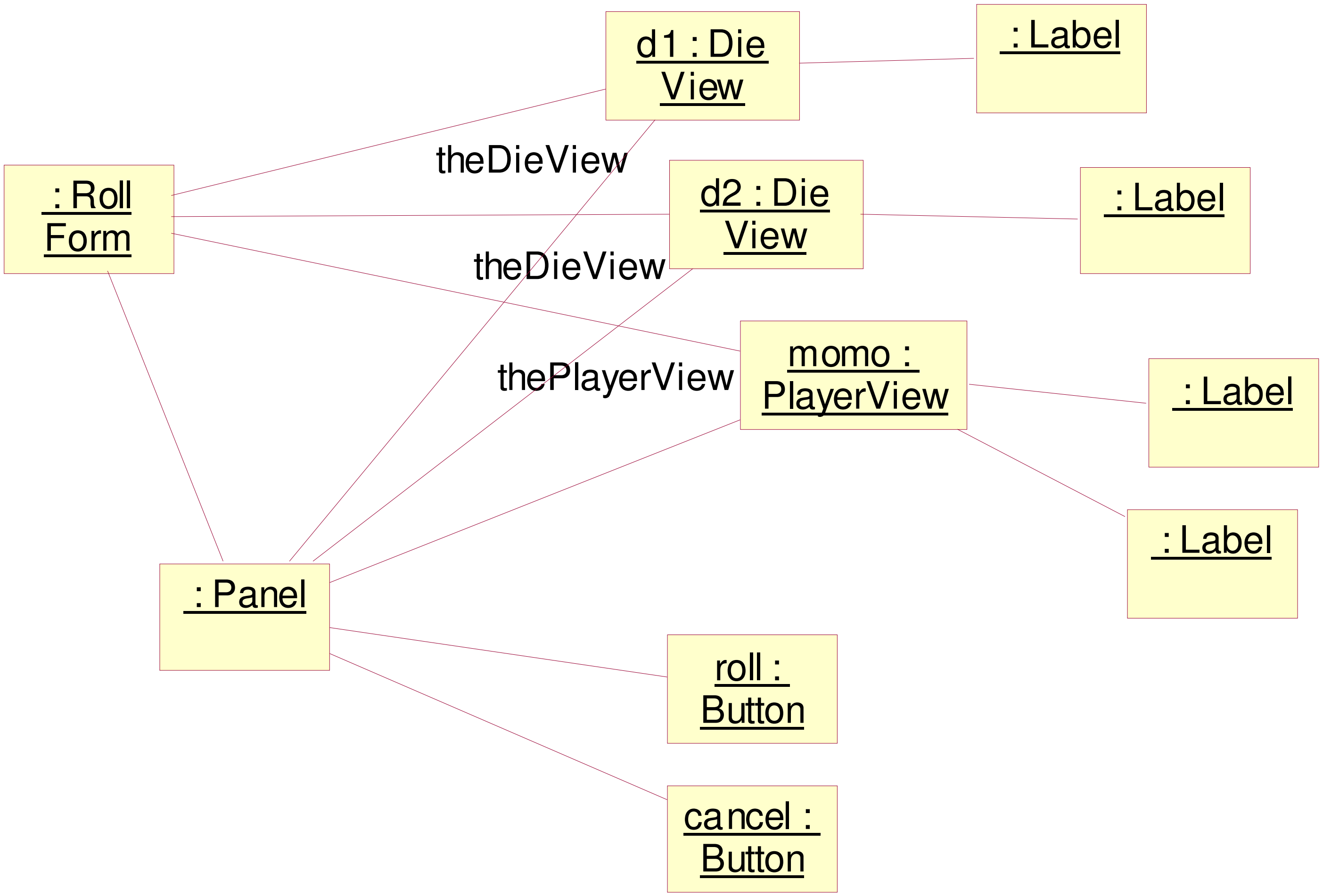
UI « Layer »


Frame
(from awt)
PlayerForm
ok_action()
cancel_action()
PlayerForm()
0..1
MainForm
quit_action()
start_action()
high_action()
MainForm()
HighScoreForm
ok_action()
RollForm
roll_action()
cancel_action()
RollForm()
0..1
1
PlayerView
PlayerView()
update()
2
DieView
DieView()
update()
HighScoreView
update()
<<Interface>>
Observer
(from util)
update()

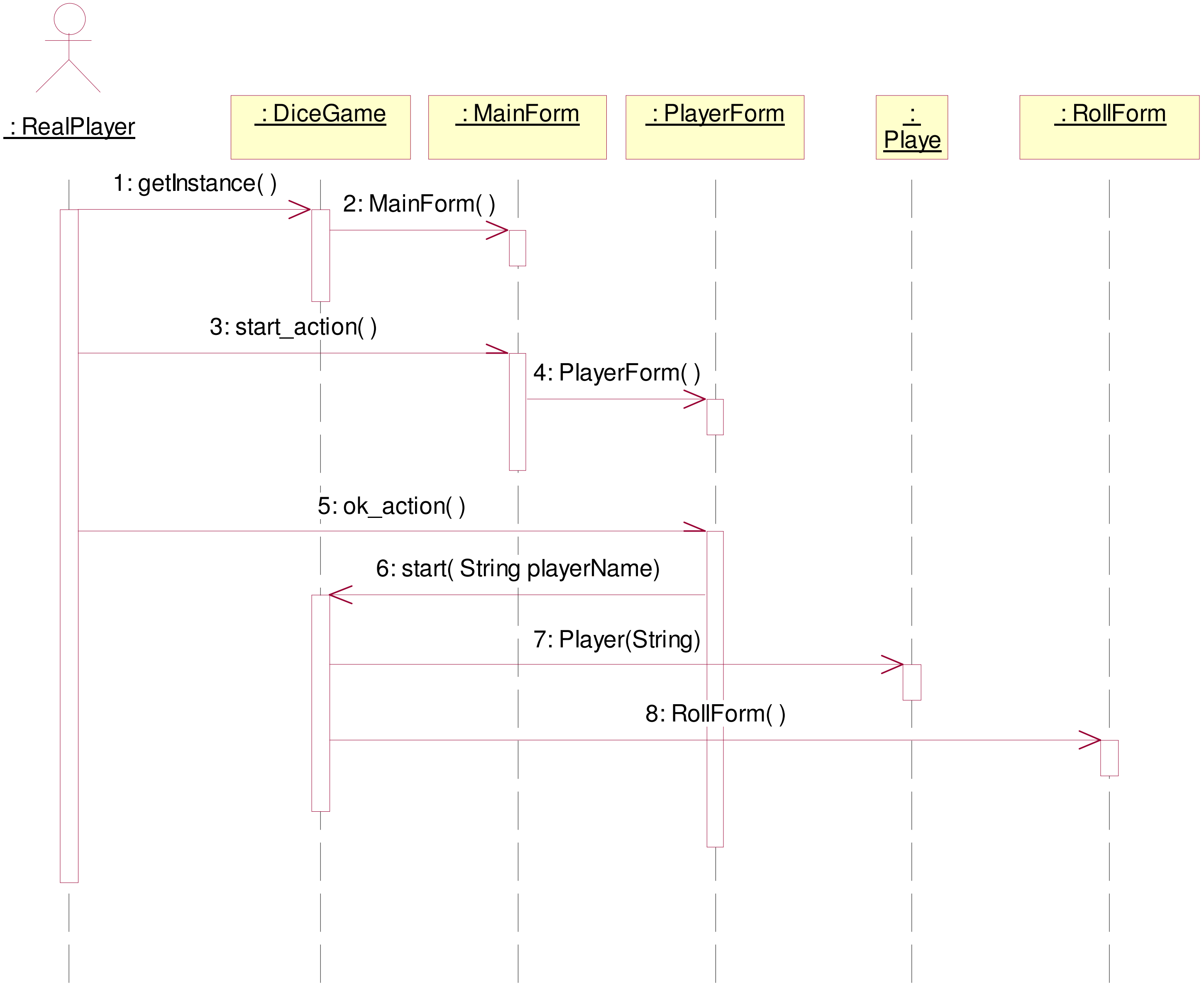
UI Mapping Class, UI



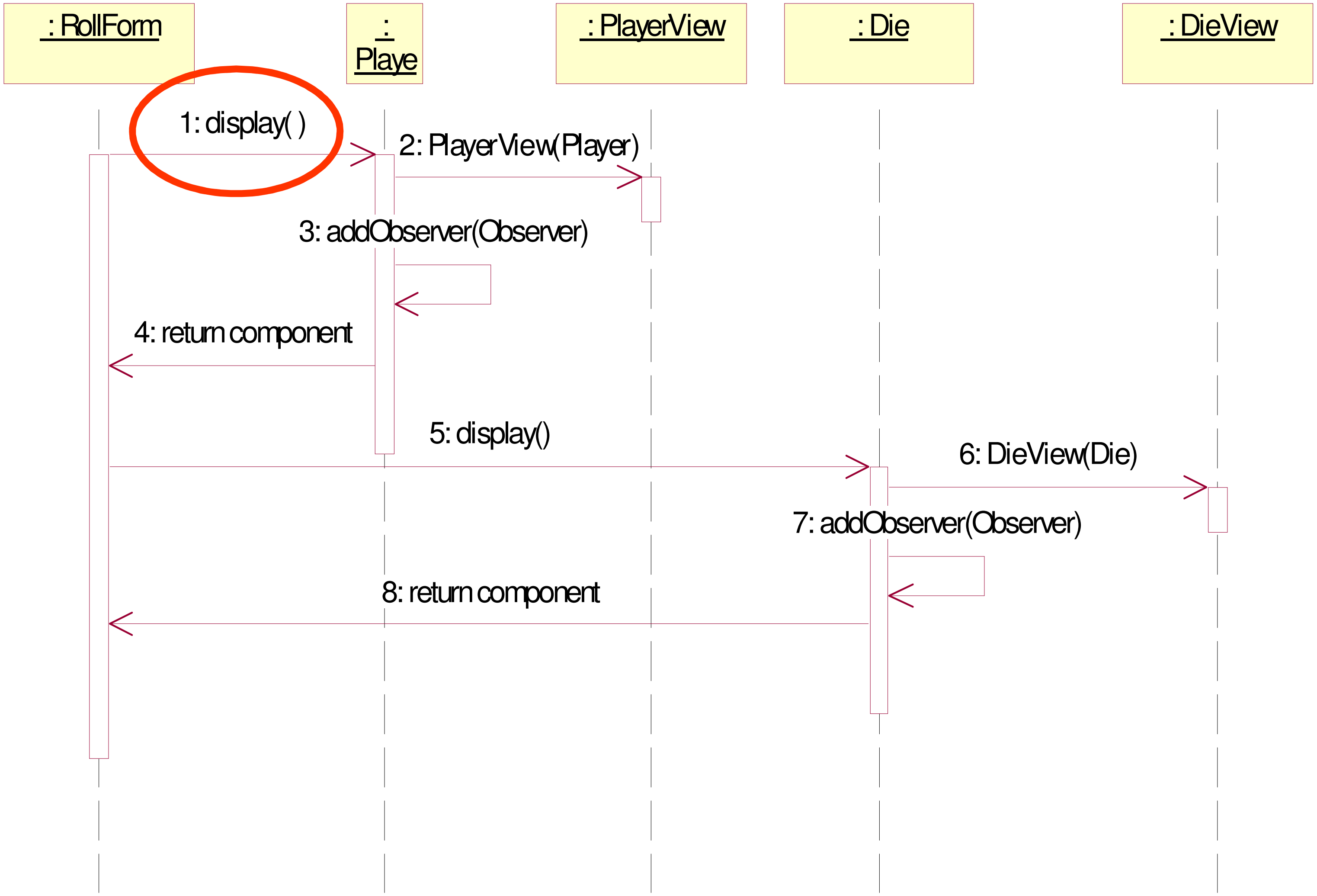
Object Diagram: rollform



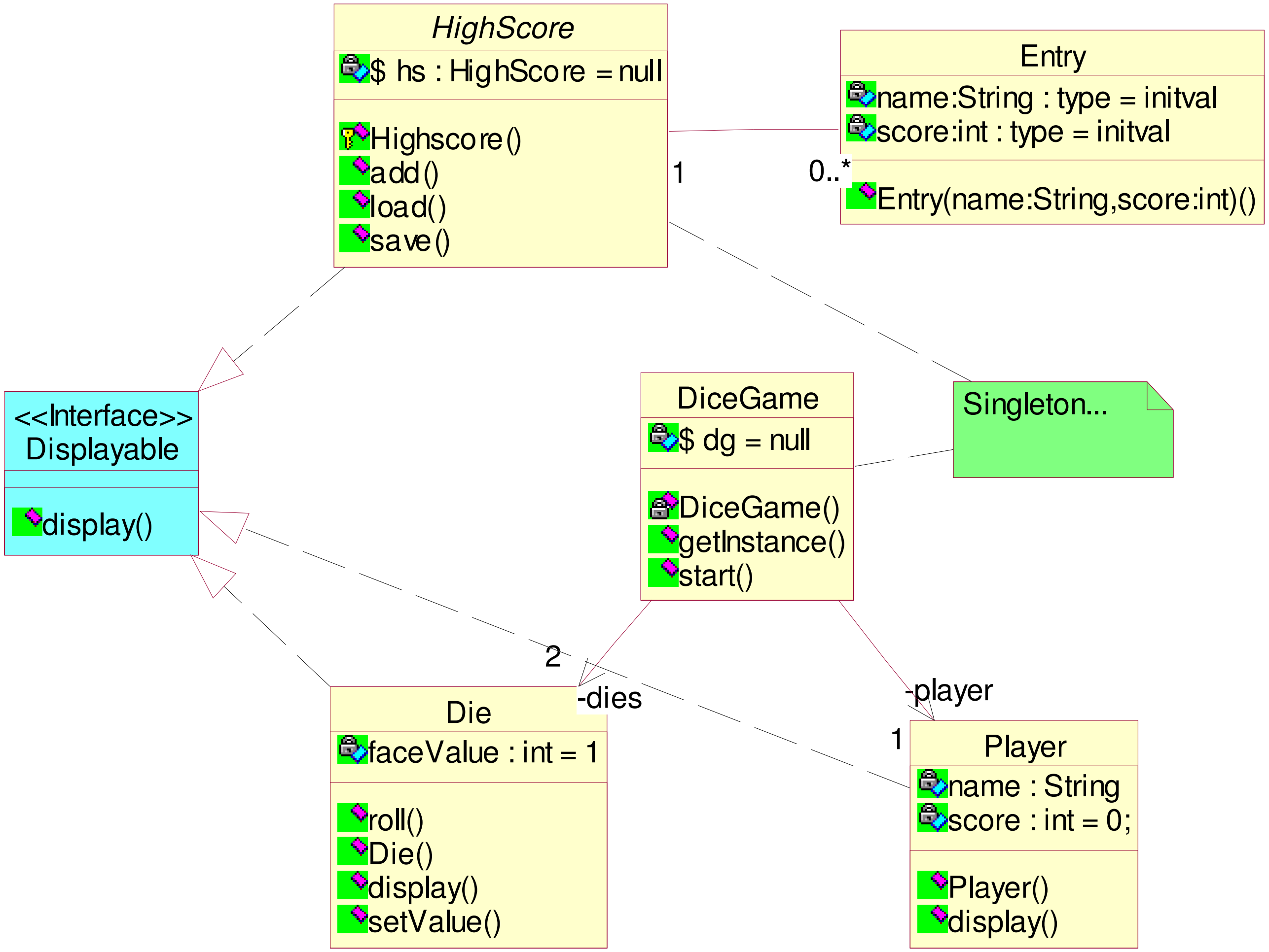
Sequence Diagram: rollform



Sequence Diagram: rollform



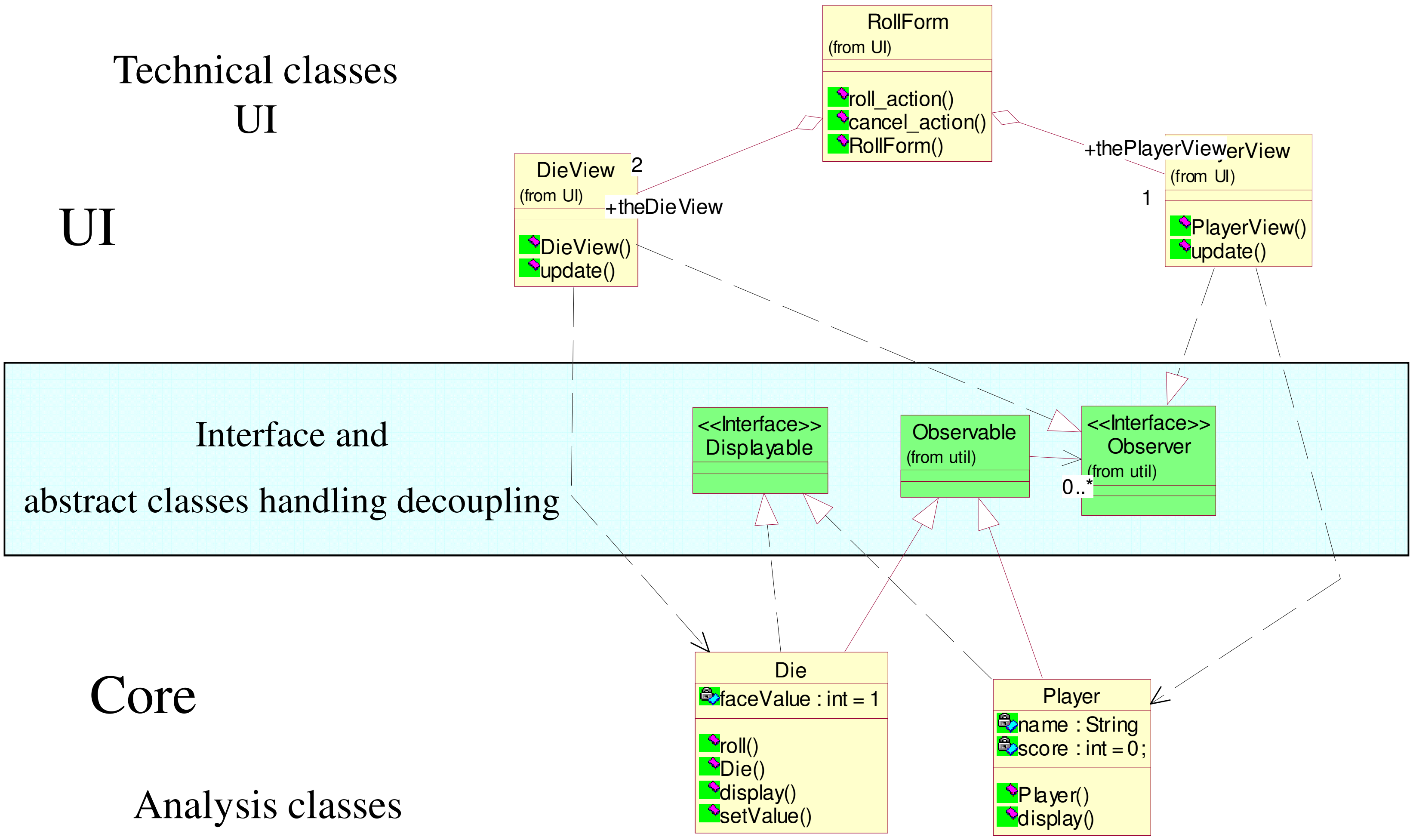
UI/Core Separation...



Layered Architecture...

Technical classes
UI

UI



Interface and

abstract classes handling decoupling

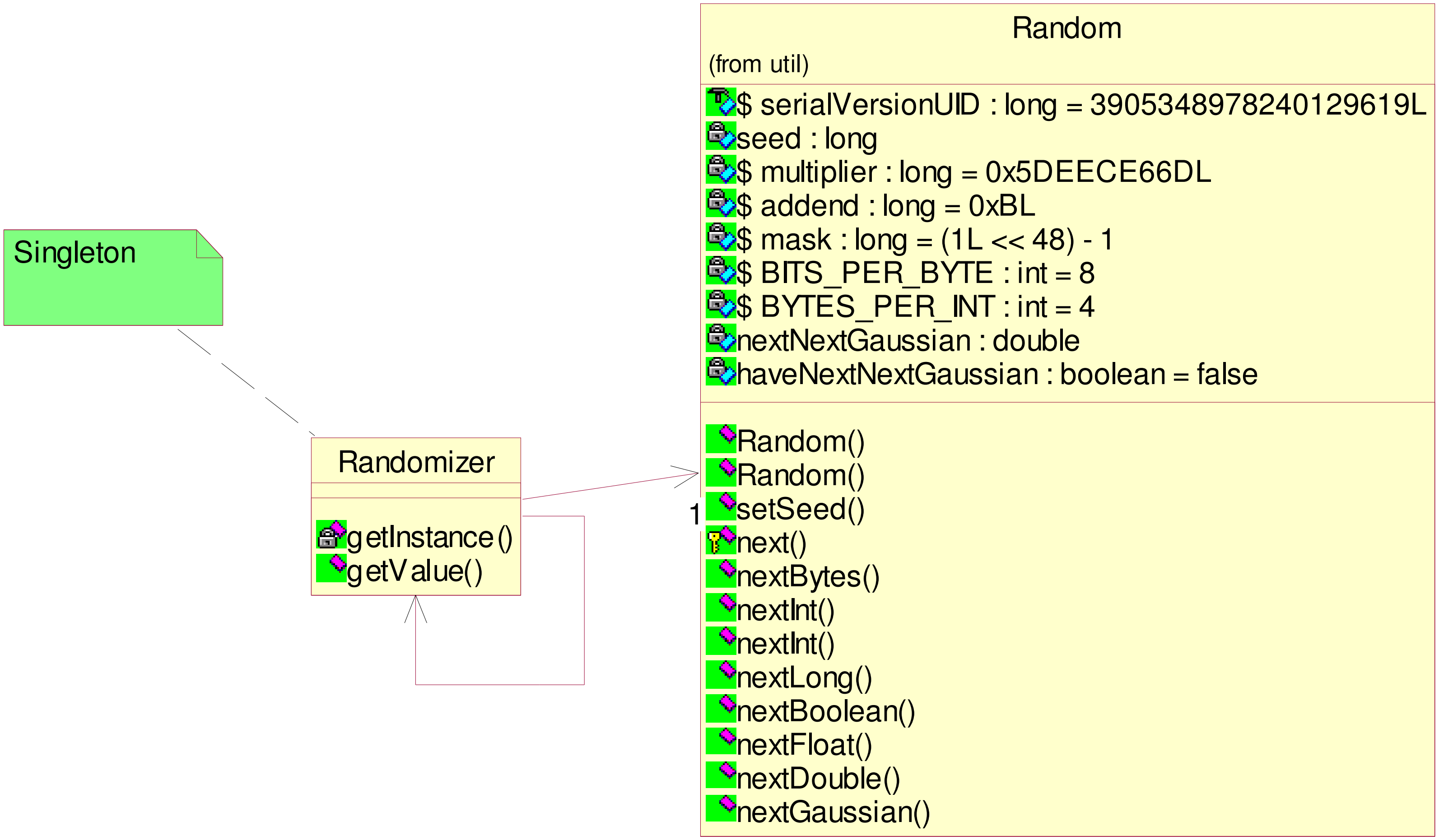
Core

Analysis classes

Util « subsystem »

- Need for random numbers
- Use java.util « random » functionality…
- The random number generator is shared by the die…
- Another singleton...

Subsystem « util »



Singleton

Randomizer

getInstance()
getValue()

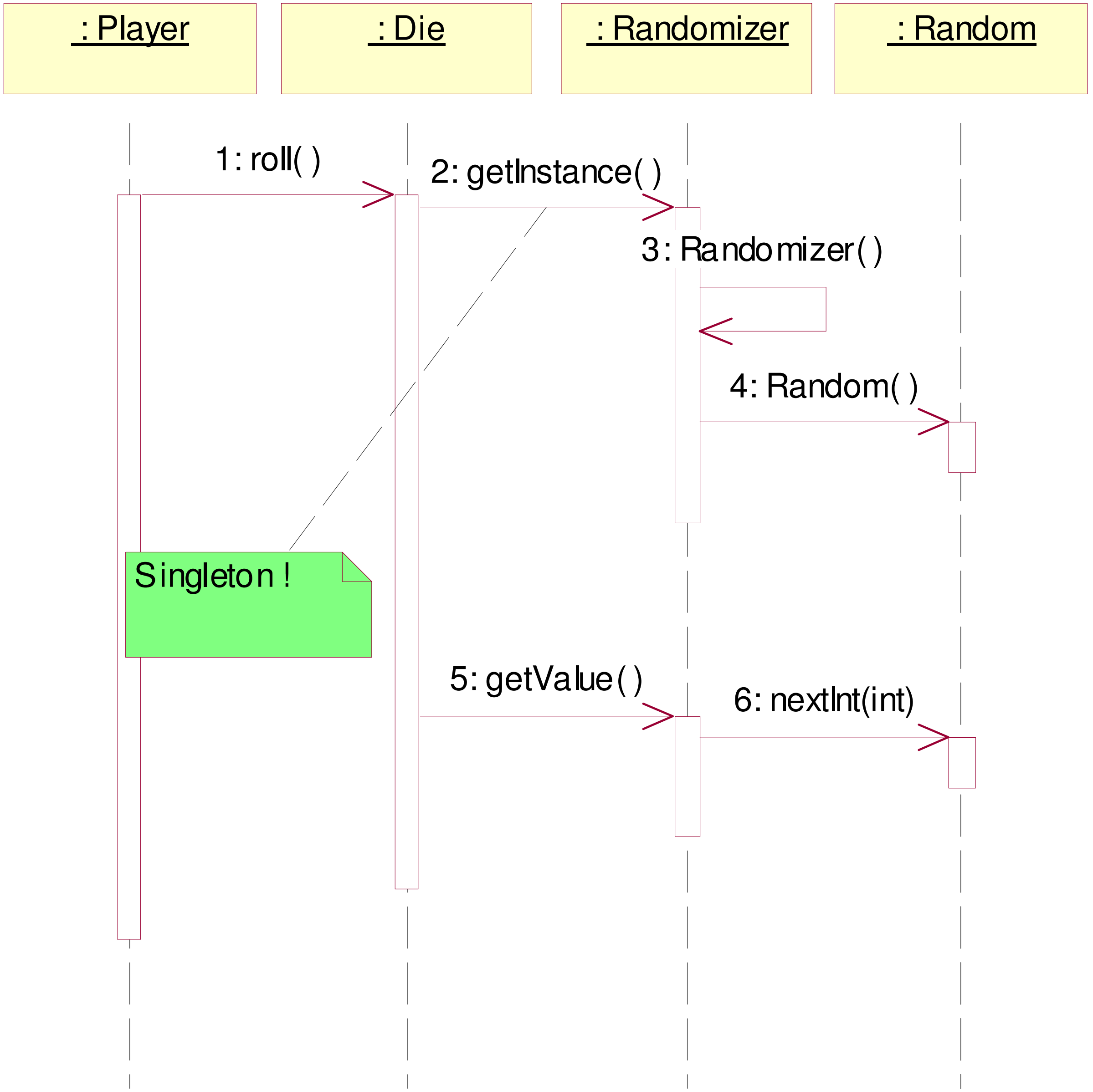
1

Random
(from util)

$ serialVersionUID : long = 3905348978240129619L
seed : long
$ multiplier : long = 0x5DEECE66DL
$ addend : long = 0xBL
$ mask : long = (1L << 48) - 1
$ BITS_PER_BYTE : int = 8
$ BYTES_PER_INT : int = 4
nextNextGaussian : double
haveNextNextGaussian : boolean = false

Random()
Random()
setSeed()
next()
nextBytes()
nextInt()
nextInt()
nextLong()
nextBoolean()
nextFloat()
nextDouble()
nextGaussian()

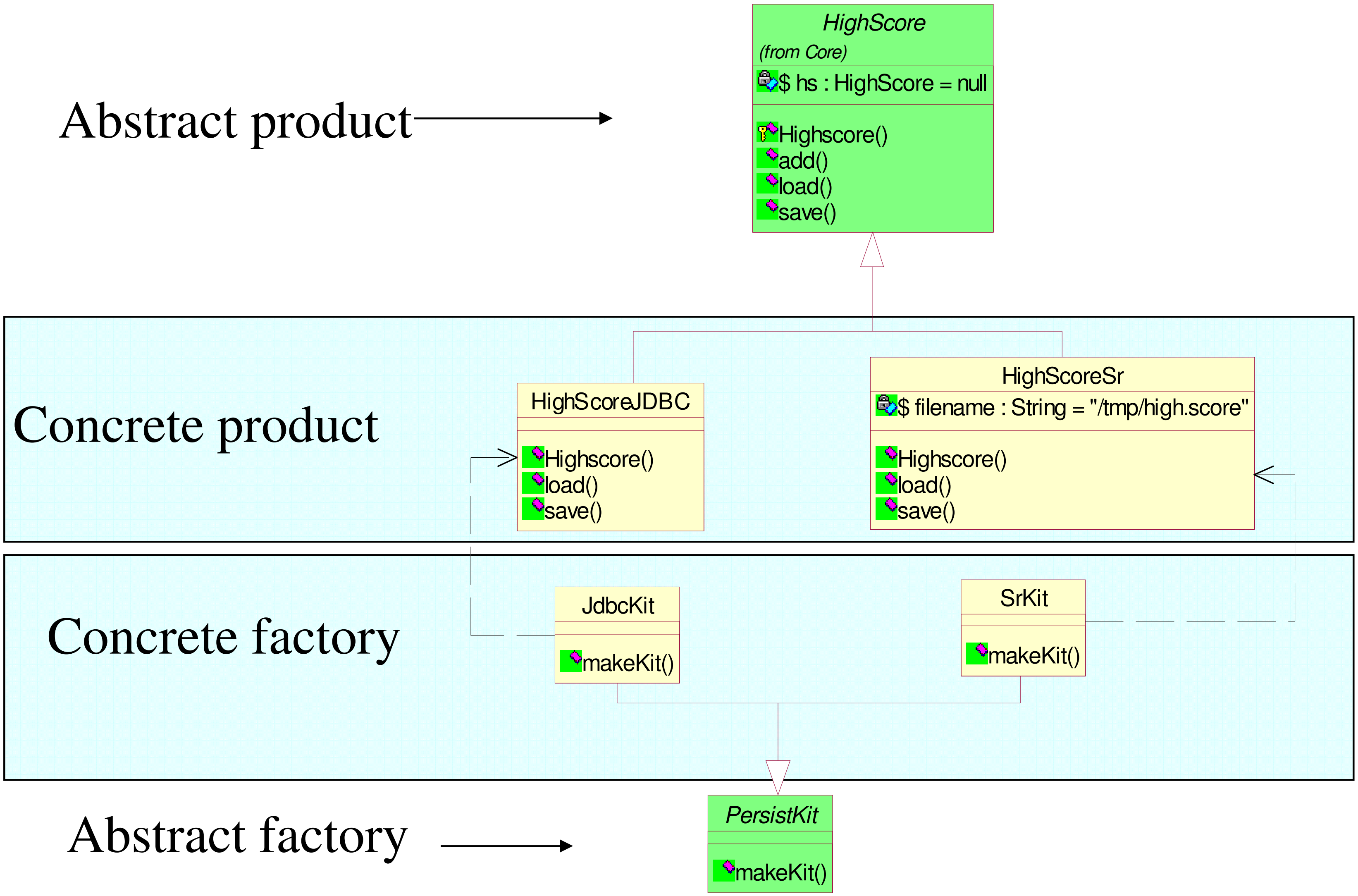
« Util » Dynamic Model



« Persist » Layer

- Technical classes used for persistence
- Ensures Core/Persist independence
  - » Ability to switch « persistent engine »
- For example:
  - » Persistence by « Serialization »
  - » Persistence via a relational DBMS (JDBC)

Isolation : Pattern Fabrication



Abstract product

HighScore
(from Core)
$ hs : HighScore = null
Highscore()
add()
load()
save()

Concrete product

HighScoreJDBC
Highscore()
load()
save()

HighScoreSr
$ filename : String = "/tmp/high.score"
Highscore()
load()
save()

Concrete factory

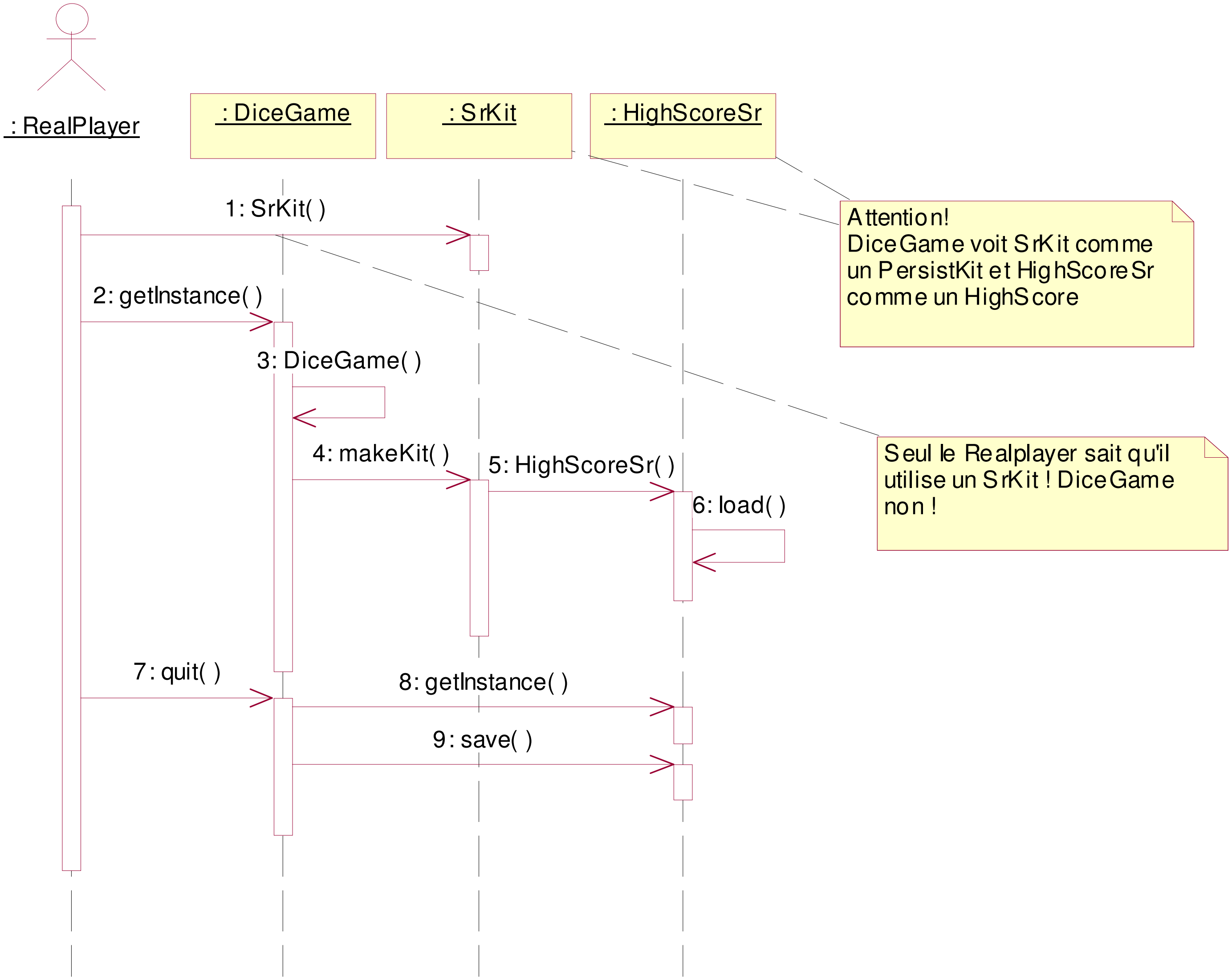
JdbcKit
makeKit()

SrKit
makeKit()

Abstract factory

PersistKit
makeKit()

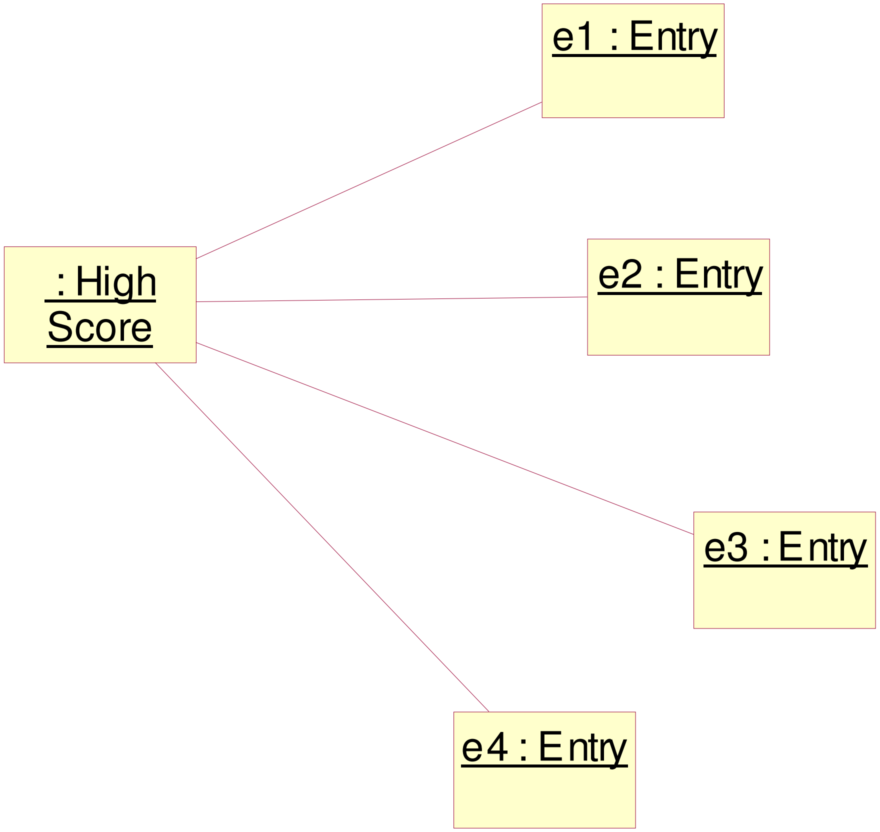
« Persist » Dynamic Model



Attention!
DiceGame voit SrKit comme un PersistKit et HighScoreSr comme un HighScore

Seul le Realplayer sait qu'il utilise un SrKit ! DiceGame non !

- Persistence propagation...



A little bit of code…that’s all folks

```
class HighScoreSr extends HighScore implements Serializable {
...
public void save() throws Exception {
  FileOutputStream ostream = new FileOutputStream(filename);
  ObjectOutputStream p = new ObjectOutputStream(ostream);

  p.writeObject(this); // Write the tree to the stream.
  p.flush();
  ostream.close();    // close the file.
}

 public void load() throws Exception {
   FileInputStream istream = new FileInputStream(filename);
   ObjectInputStream q = new ObjectInputStream(istream);

   HighScoreSr hsr = (HighScoreSr)q.readObject();
}
}
```

JdbPersist... Dynamic Model

- A table must be created in a relational DBMS
- Upon creation of HighScoreJDBC:
  Connection to DBMS via JDBC
- **save**:
  - Perform « inserts » for each « entry »
- **load**:
  - Select * from ...,
  - Follow the result,
  - create « entry » objects

A little bit of code...

```
public class HighScoreJDBC extends HighScore {
    public static final String url="jdbc:odbc:dice";
    Connection con=null;

    public HighScoreJDBC() {
       try {
           //loads the driver
           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           con=DriverManager.getConnection(url,
                         "molli","");
       } catch (Exception e) {
           e.printStackTrace();
           new Error("Cannot access Database at"+url);
       }
       hs=this; // register unique instance !
       this.load();
    }
```

Jdbc Load

```
public void load() {
       try {
           Statement select=con.createStatement();
           ResultSet result=select.executeQuery
              ("SELECT Name,Score FROM HighScore");
           while (result.next()) {
              this.add(new Entry(result.getString(1),
                                 result.getInt(2)));
           }
       } catch (Exception e) {
           e.printStackTrace();
       }
    }
```

Jdbc Save

```
public void save() {
      try {
          for (Enumeration e = this.elements() ;
               e.hasMoreElements() ;) {
             Entry entry=(Entry)e.nextElement();
             Statement s=con.createStatement();
             s.executeUpdate(
                 "INSERT INTO HighScore (Name,Score)"+
                "VALUES('"+entry.getName()+"',"+
                          entry.getScore()+")");
          }
      } catch (Exception e) {
          e.printStackTrace();
      }
   }
```

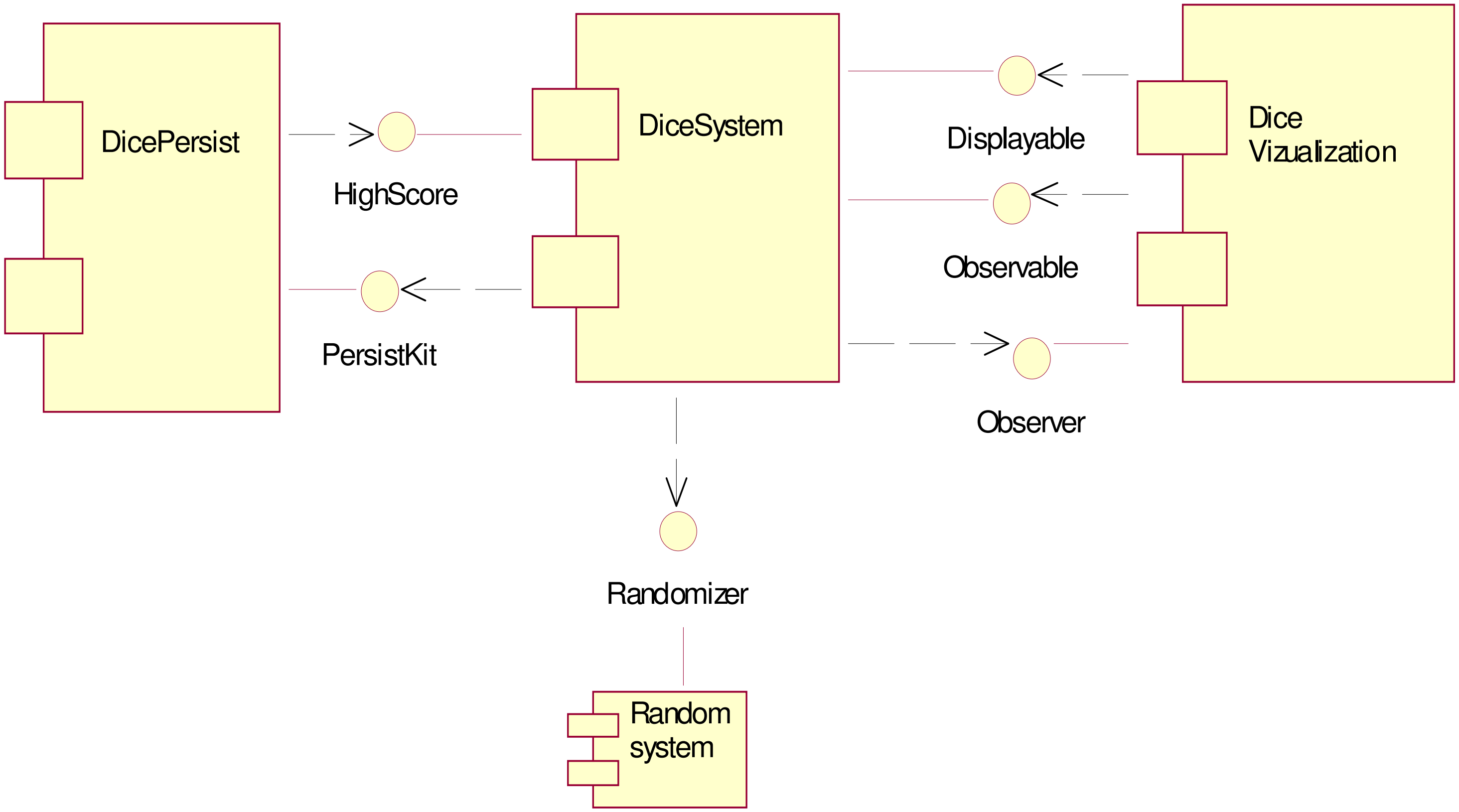
Component diagram...

- A component is a « non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture »
- A component conforms to and provides the physical realization of a set of interfaces.

Component diagram

- « Realize » : implement interfaces
- « Depend » : Use interfaces
- Interfaces isolate components

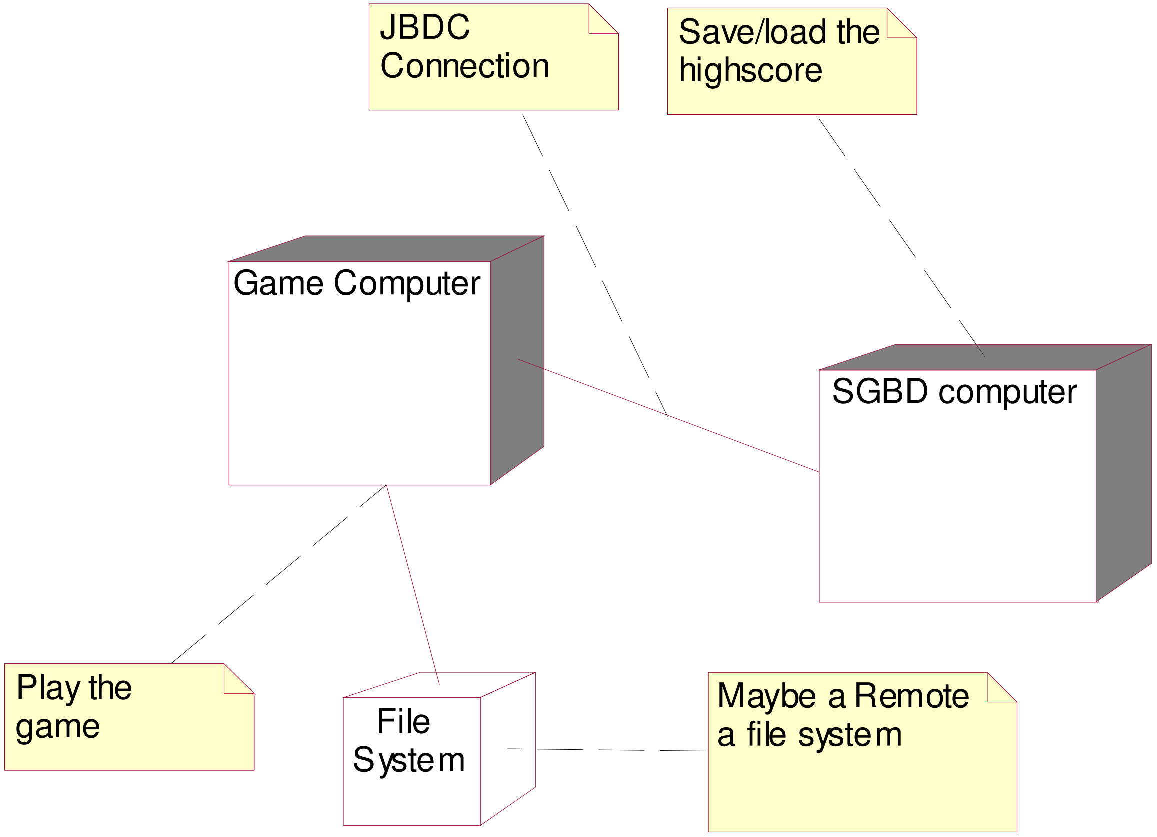
Component diagram...



Deployment Diagrams

- Display the physical architecture
- Associer execution units to associated handlers
- Identify connections between execution units

Deployment Diagrams



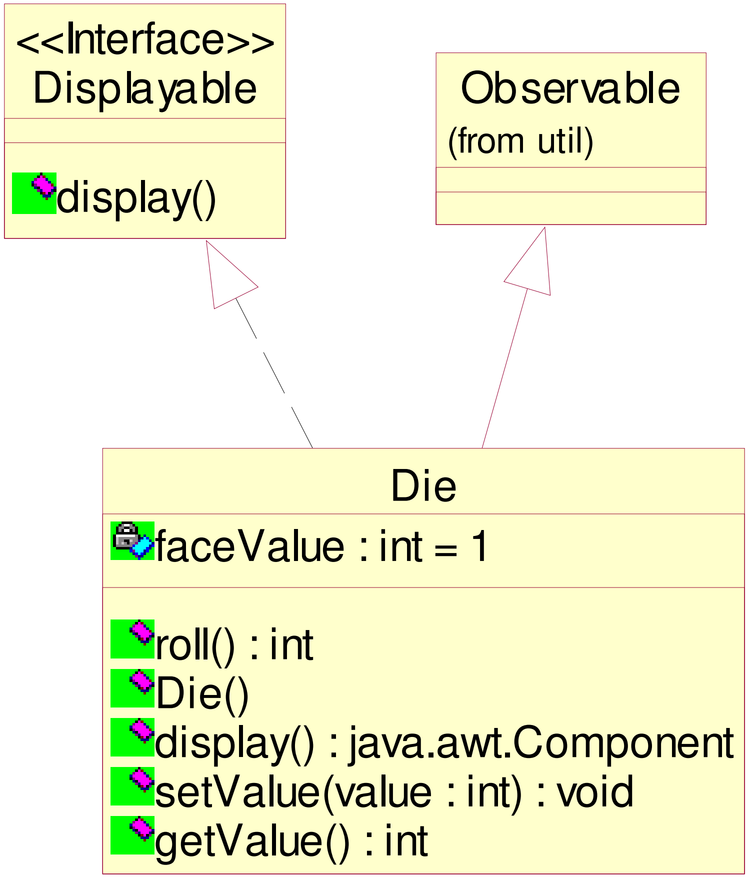
Design Complete ?

- Functionality coverage : compare Use-case and activity diagrams…
- Consistency between diagrams ??
  - » Some inconsistencies… UI vs Core
  - » Core/Persist independence partially modeled...

Generate code : code mapping

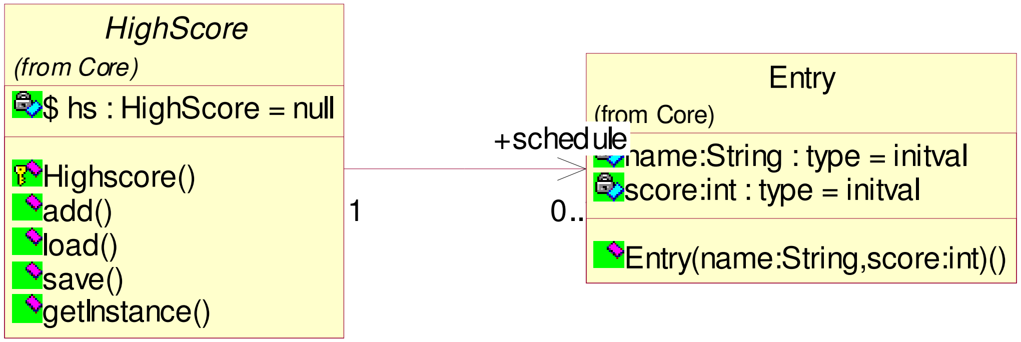
- Map into any programming language !
- OO languages : java, C++, smalltalk
- Or: VB, C, Fortran
- As well as: SQL, Cobol...

Java Mapping...



```
package Core;

import Util.Randomizer;
import UI.DieView;
import java.util.*;
import java.awt.Component;

public class Die extends Observable
implements Displayable {
private int faceValue = 1;

public int roll() {

setValue(Randomizer.getInstance().
          getValue());
          return getValue();
    }
public java.awt.Component display()
{
          Component c=new
DieView(this);
          this.addObserver((Observer
)c);
          return c;
    }
public void setValue(int value) {
          faceValue=value;
          this.setChanged();
          this.notifyObservers();
    }
public int getValue() {       return
faceValue;}
}
```

Relationships

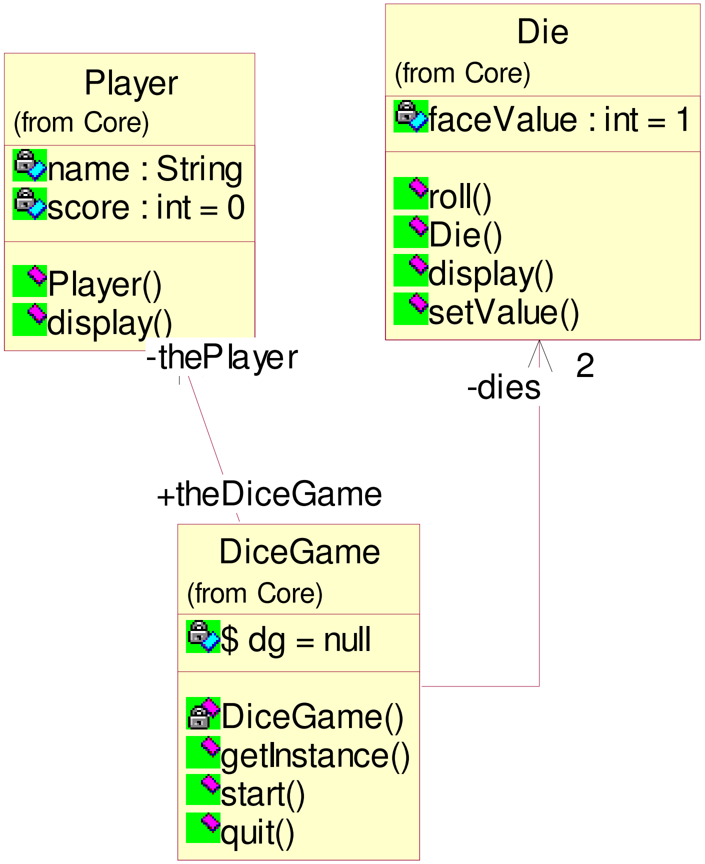


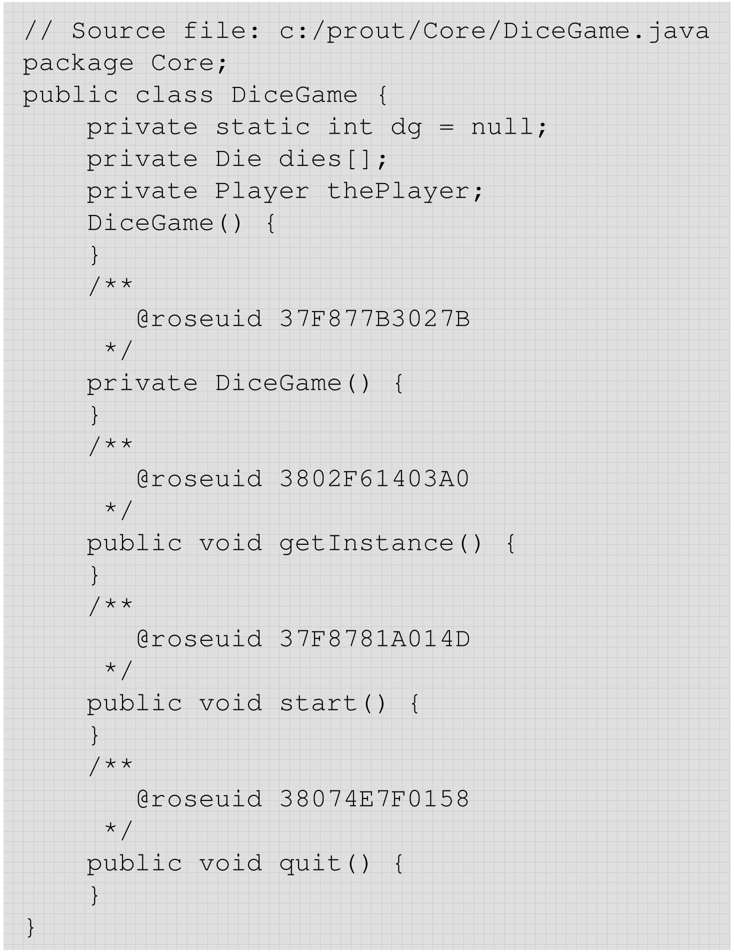
```
package Core;
import java.util.*;
import java.awt.Component;
import UI.HighScoreView;
public abstract class HighScore
extends Observable implements
java.io.Serializable, Displayable {
    protected static HighScore hs = null;
    public Vector entries=new Vector();
public void add(Entry entry) {
          entries.addElement(entry);
          this.setChanged();
          this.notifyObservers();
    }

    public Enumeration elements() {
          return entries.elements();
    }
public abstract void load();
public abstract void save();
public Component display() {
 Component c=new HighScoreView(this);
 this.addObserver((java.util.Observer)c);
 return c;
}
public static HighScore getInstance() {
 if (hs==null) {
     new Error("No Persist Kit declared");
 }
 return hs;}
```

Programming...

- Use « forward engineering » functionality provided by tools
- Then « reverse engineering »
- To achieve « round trip engineering » ;-D
- Ensure Programming/Design/Analysis consistency...

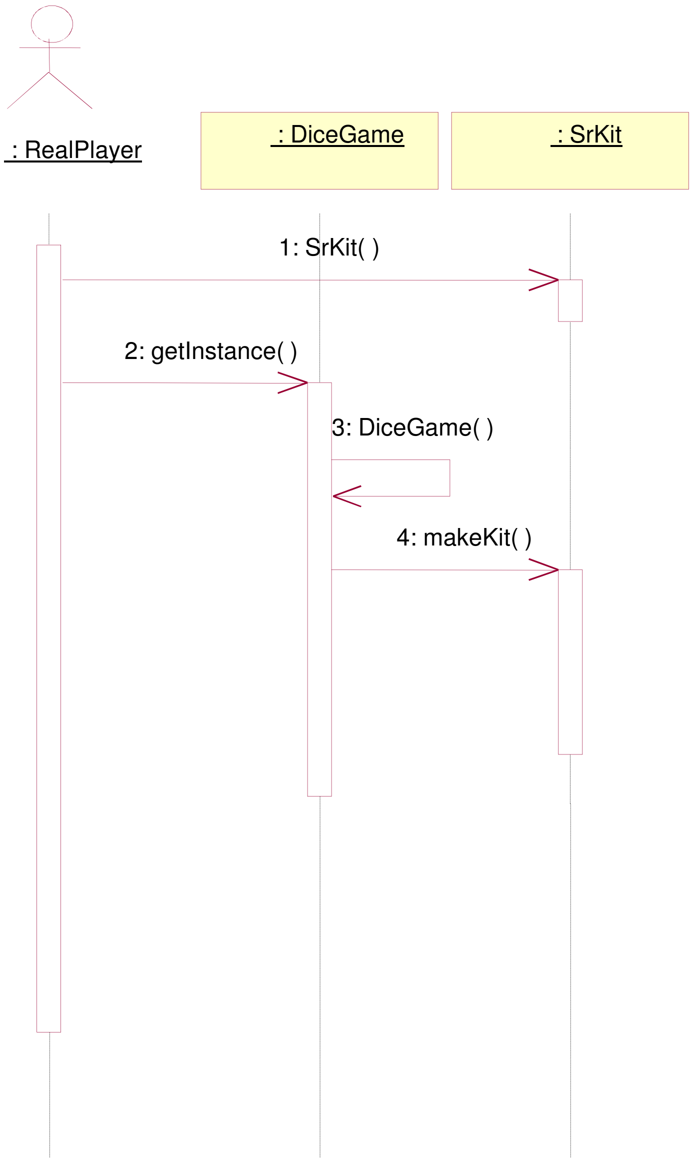
Forward engineering





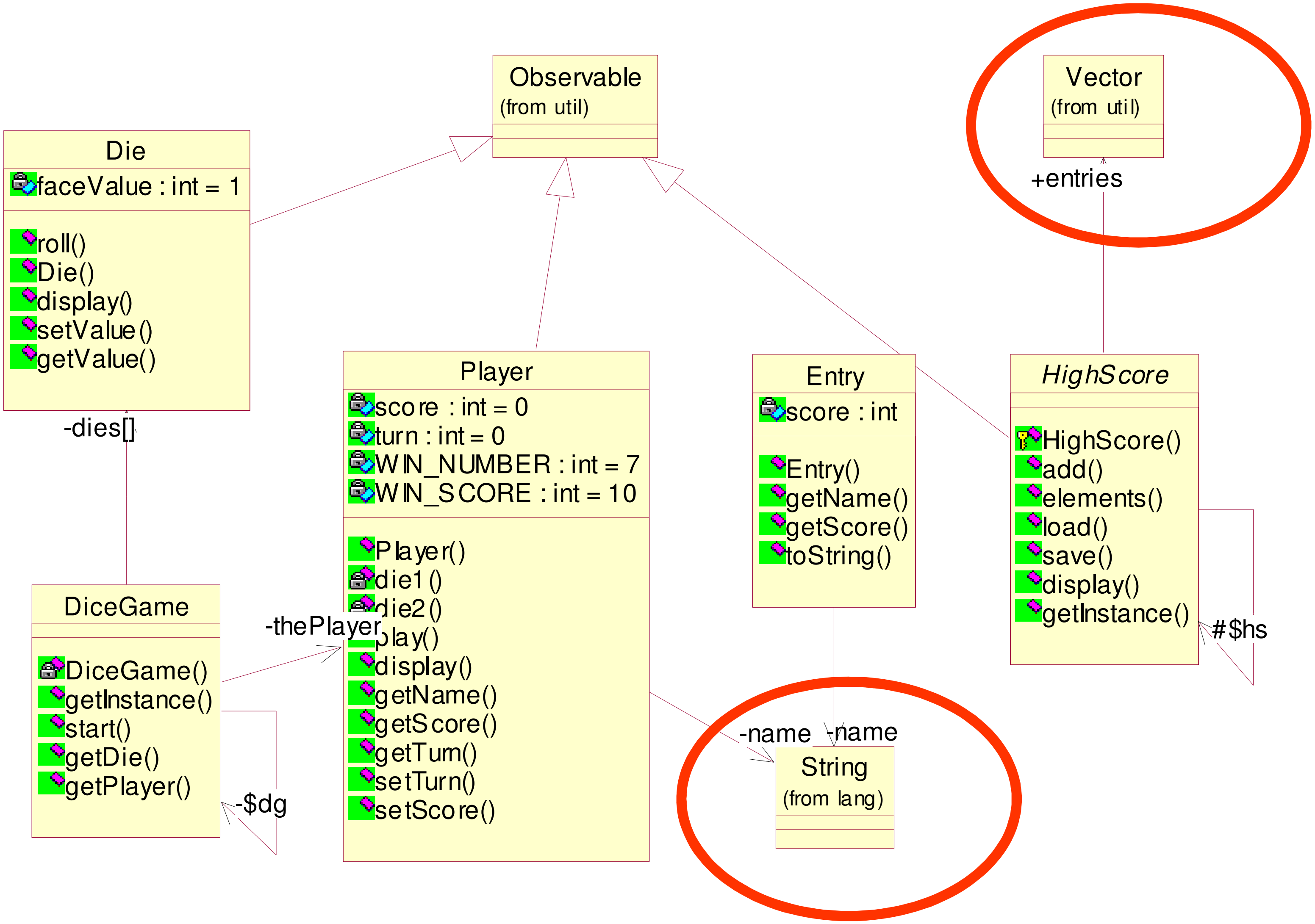
```
// Source file: c:/prout/Core/DiceGame.java
package Core;
public class DiceGame {
     private static int dg = null;
     private Die dies[];
     private Player thePlayer;
     DiceGame() {
     }
     /**
        @roseuid 37F877B3027B
      */
     private DiceGame() {
     }
     /**
        @roseuid 3802F61403A0
      */
     public void getInstance() {
     }
     /**
        @roseuid 37F8781A014D
      */
     public void start() {
     }
     /**
        @roseuid 38074E7F0158
      */
     public void quit() {
     }
}
```

Forward Engineering



```
package Core;

import UI.MainForm;
import Persist.*;
import java.awt.*;

class Main {
    public static void main(String args[]) {
        // SrKit srk=new SrKit();
        JdbcKit srk=new JdbcKit();
        DiceGame dg=DiceGame.getInstance();
        Frame f=MainForm.getInstance();
        f.setSize(300,300);
        f.show();
    }
}
```

Reverse Engineering...



Observable
(from util)

Vector
(from util)

+entries

Die
faceValue : int = 1
roll()
Die()
display()
setValue()
getValue()

-dies[]

Player
score : int = 0
turn : int = 0
WIN_NUMBER : int = 7
WIN_SCORE : int = 10
Player()
die1()
die2()
play()
display()
getName()
getScore()
getTurn()
setTurn()
setScore()

Entry
score : int
Entry()
getName()
getScore()
toString()

HighScore
HighScore()
add()
elements()
load()
save()
display()
getInstance()

#$hs

DiceGame
DiceGame()
getInstance()
start()
getDie()
getPlayer()

-thePlayer

-$dg

-name

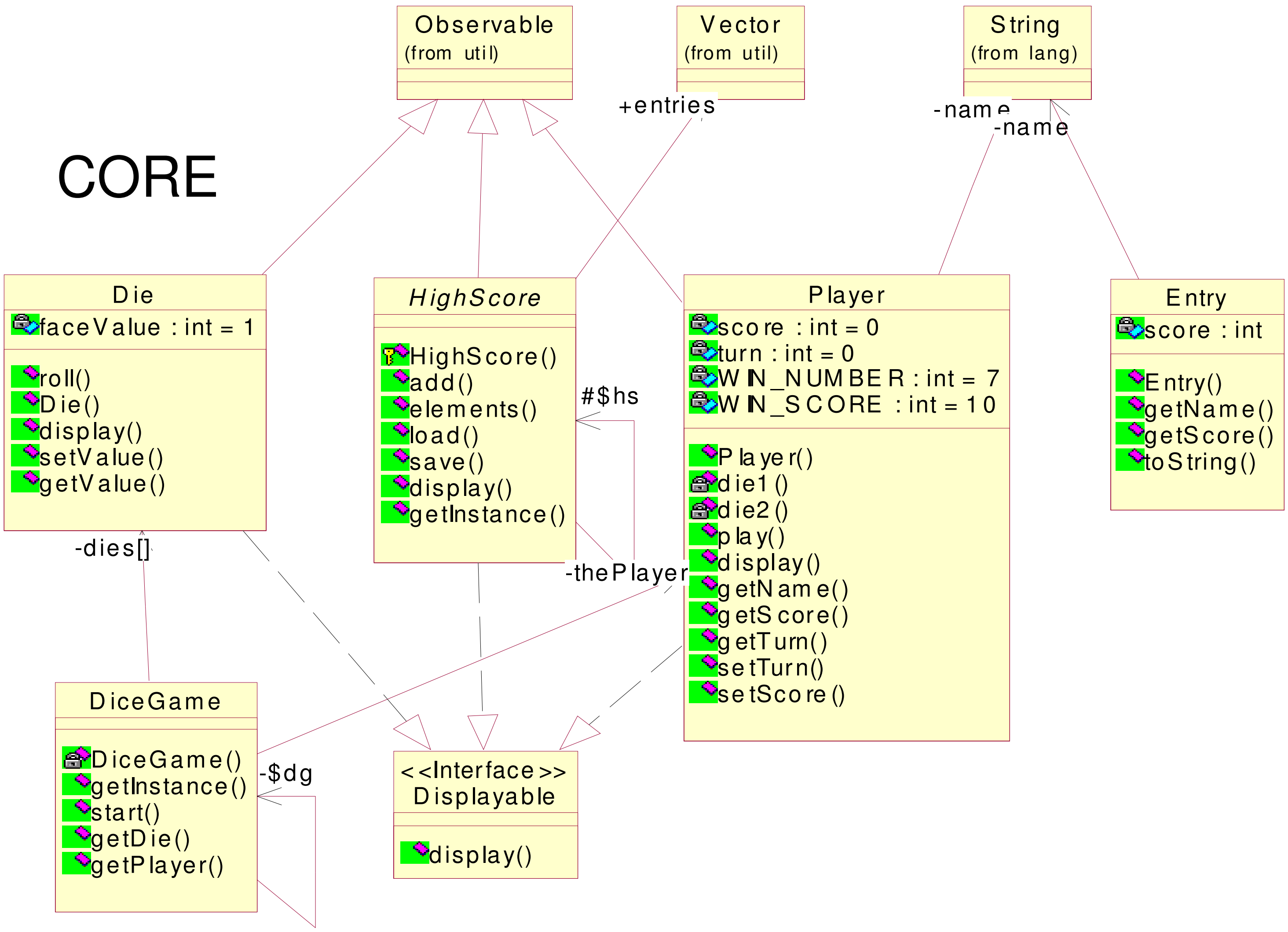
-name

String
(from lang)

Reverse engineering

- Does not apply to the dynamic model !
- Handle forward+modification+reverse issues
- Nothing miraculous !

Design/Implementation Comparison



CORE

Observable
(from util)

Vector
(from util)

String
(from lang)

+entries

-name

-name

Die
faceValue : int = 1
roll()
Die()
display()
setValue()
getValue()

HighScore
HighScore()
add()
elements()
load()
save()
display()
getInstance()

#$hs

Player
score : int = 0
turn : int = 0
WIN_NUMBER : int = 7
WIN_SCORE : int = 10
Player()
die1()
die2()
play()
display()
getName()
getScore()
getTurn()
setTurn()
setScore()

Entry
score : int
Entry()
getName()
getScore()
toString()

-dies[]

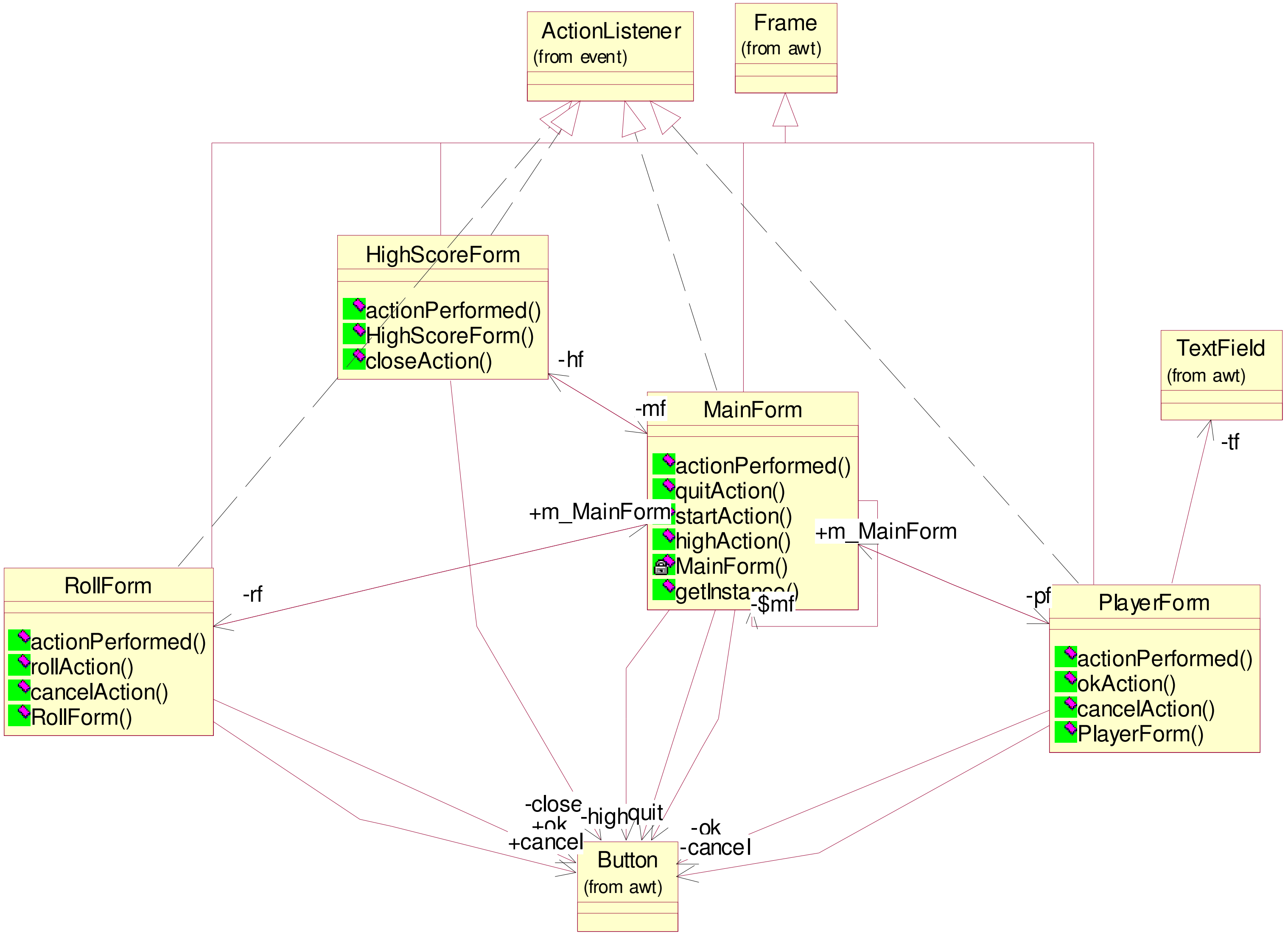
-thePlayer

DiceGame
DiceGame()
getInstance()
start()
getDie()
getPlayer()

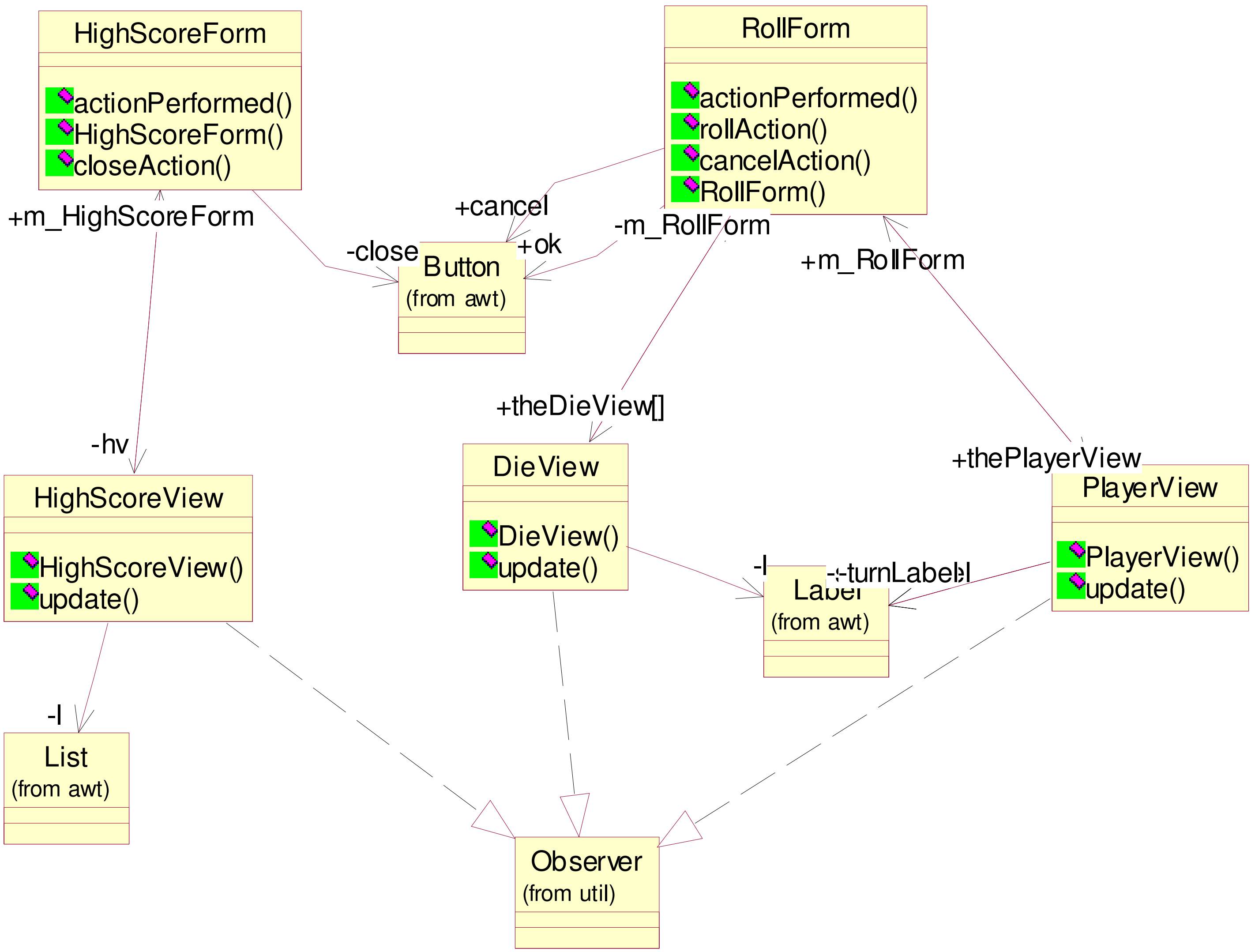
-$dg

<<Interface>>
Displayable
display()

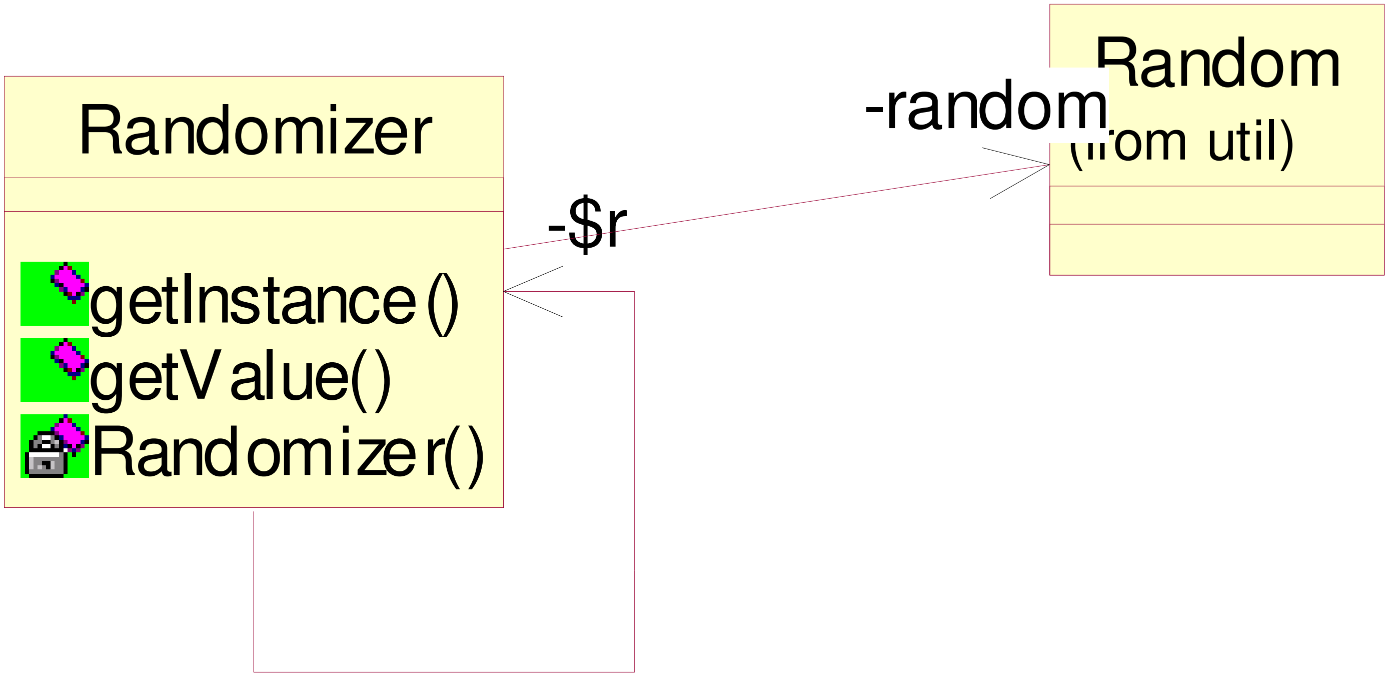
UI(1)


ActionListener
(from event)
Frame
(from awt)
HighScoreForm
actionPerformed()
HighScoreForm()
closeAction()
-hf
-mf
MainForm
actionPerformed()
quitAction()
startAction()
highAction()
MainForm()
getInstance()
-$mf
+m_MainForm
+m_MainForm
TextField
(from awt)
-tf
RollForm
actionPerformed()
rollAction()
cancelAction()
RollForm()
-rf
-pf
PlayerForm
actionPerformed()
okAction()
cancelAction()
PlayerForm()
-close
+ok
+cancel
-high
quit
-ok
-cancel
Button
(from awt)

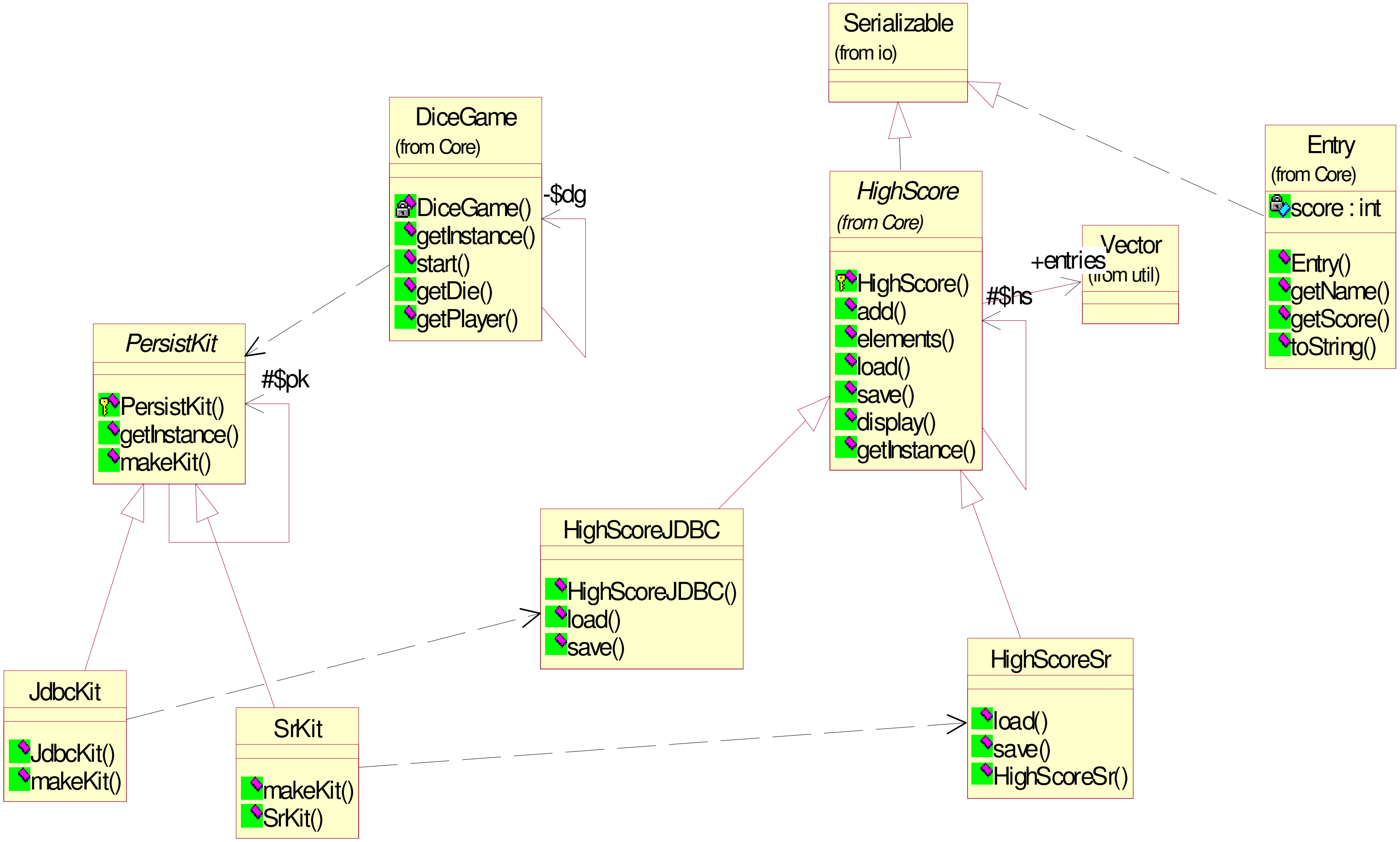
UI(2)


HighScoreForm
actionPerformed()
HighScoreForm()
closeAction()
RollForm
actionPerformed()
rollAction()
cancelAction()
RollForm()
+m_HighScoreForm
+cancel
-close
Button
(from awt)
+ok
-m_RollForm
+m_RollForm
+theDieView[]
-hv
HighScoreView
HighScoreView()
update()
DieView
DieView()
update()
+thePlayerView
PlayerView
PlayerView()
update()
-l
-turnLabel
-l
Label
(from awt)
-l
List
(from awt)
Observer
(from util)

Util


Randomizer
getInstance()
getValue()
Randomizer()
-$r
-random
Random
(from util)

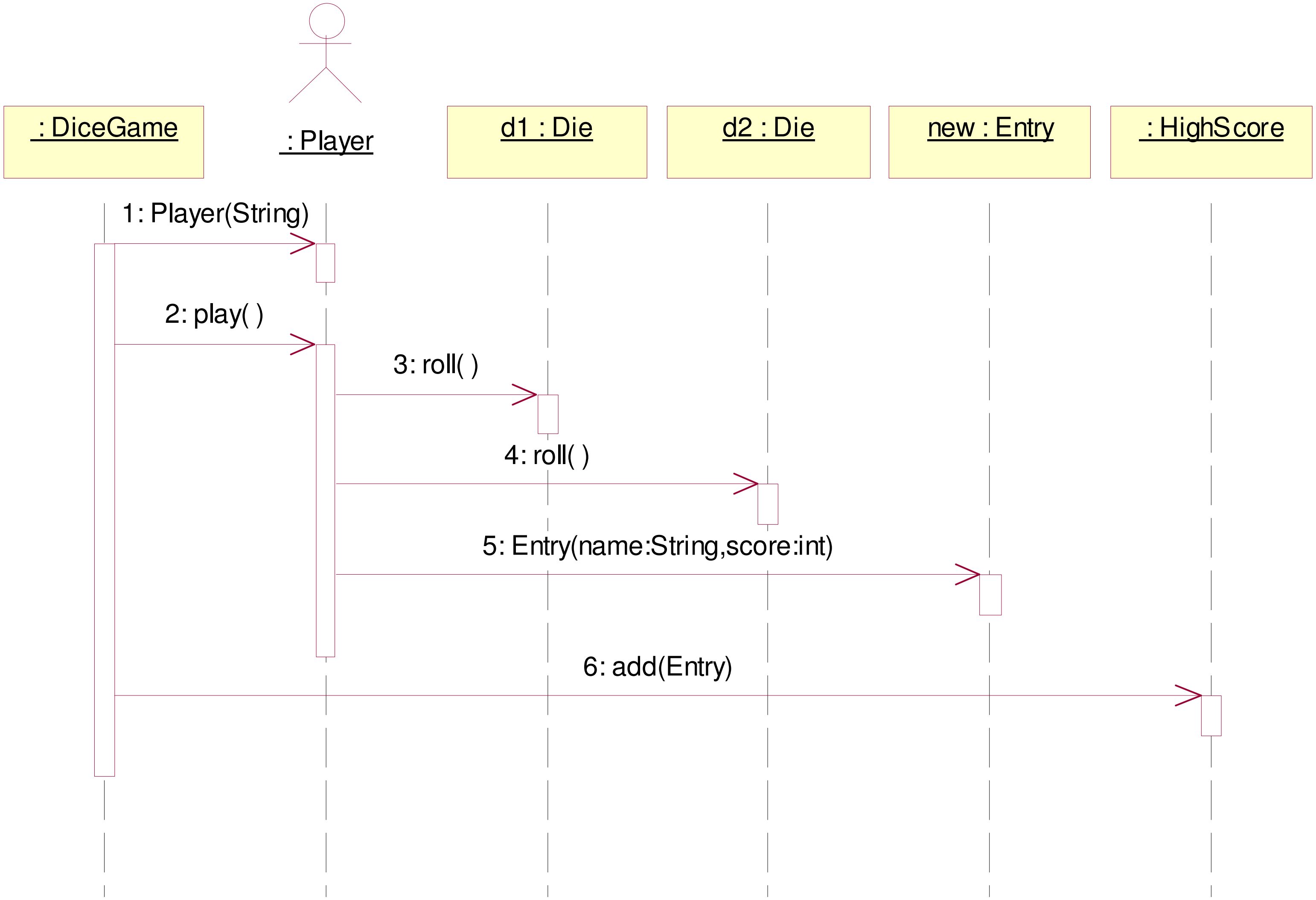
Persist


Serializable
(from io)
DiceGame
(from Core)
DiceGame()
getInstance()
start()
getDie()
getPlayer()
-$dg
HighScore
(from Core)
HighScore()
add()
elements()
load()
save()
display()
getInstance()
+entries
Vector
(from util)
#$hs
Entry
(from Core)
score : int
Entry()
getName()
getScore()
toString()
PersistKit
PersistKit()
getInstance()
makeKit()
#$pk
HighScoreJDBC
HighScoreJDBC()
load()
save()
JdbcKit
JdbcKit()
makeKit()
SrKit
makeKit()
SrKit()
HighScoreSr
load()
save()
HighScoreSr()

Problems Found

- Dynamic model to handle turns is not designed properly !
- Who really tests for the end of the game ?
- Design flaw !

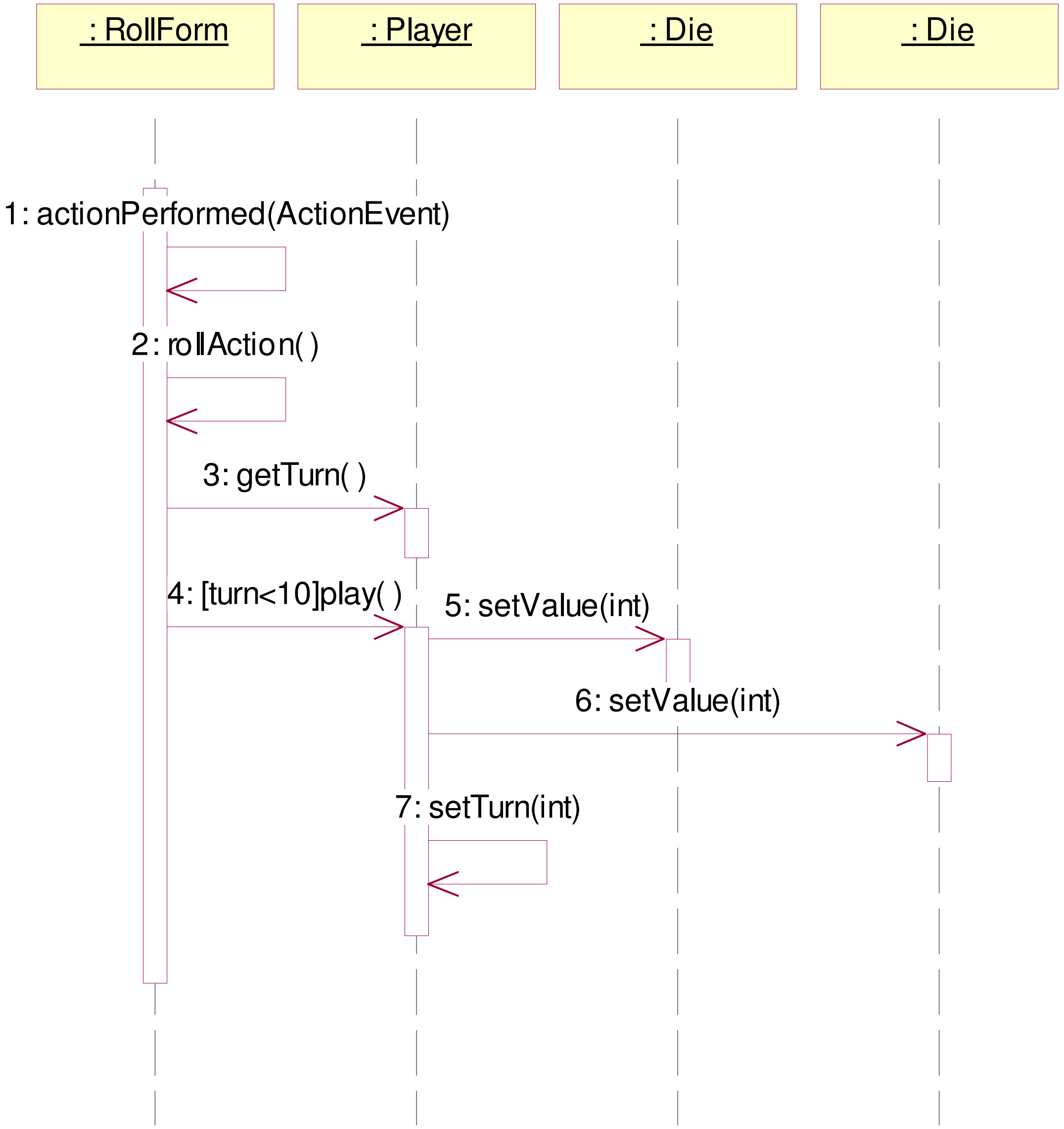
Here ! Analysis Diagram !!



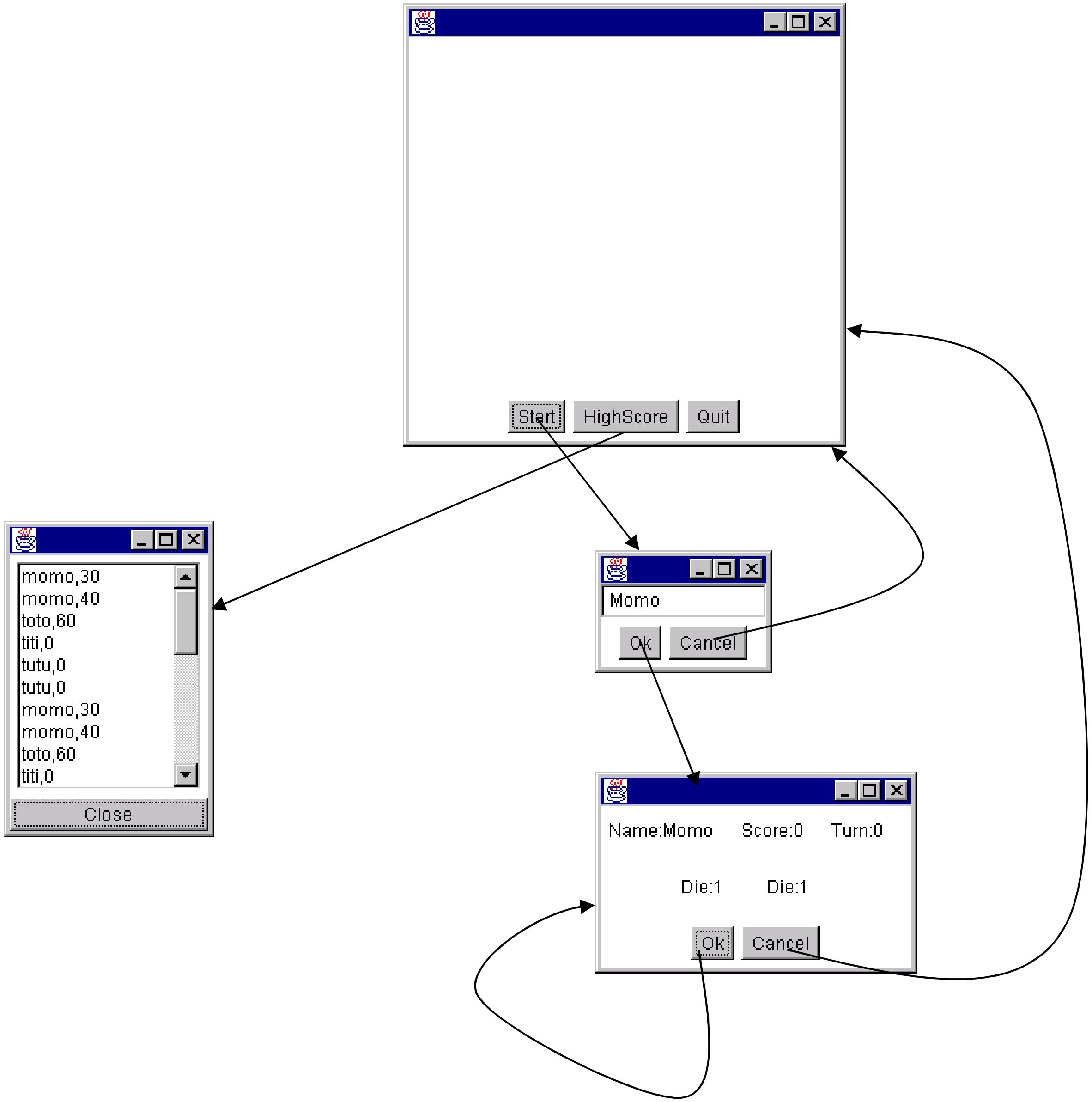
Problem !

- Not formal enough !
- This analysis diagram was not reviewed at design time !!!
- (-4)

Redo !


: RollForm
: Player
: Die
: Die
1: actionPerformed(ActionEvent)
2: rollAction( )
3: getTurn( )
4: [turn<10]play( )
5: setValue(int)
6: setValue(int)
7: setTurn(int)

Finally !



```
momo,30
momo,40
toto,60
titi,0
tutu,0
tutu,0
momo,30
momo,40
toto,60
titi,0
```

Momo

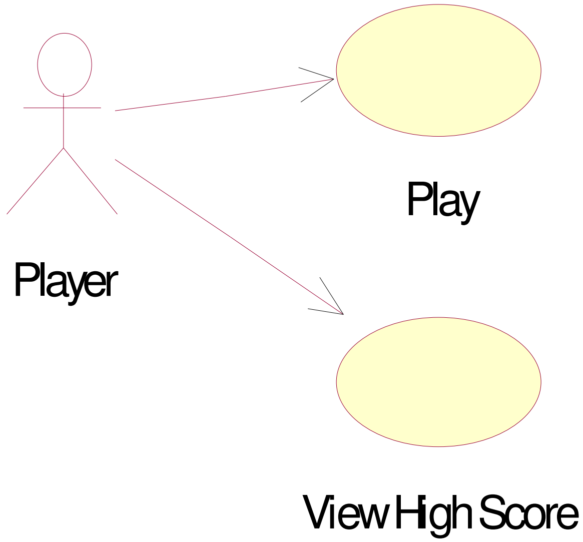
Name:Momo Score:0 Turn:0

Die:1 Die:1

Does it work ? Testing

- Unit testing : test each class and each method at a time
  - » Class diagram
- Integration tests :
  - » Component diagram
- System test :
  - » Use Case + Activity diagram

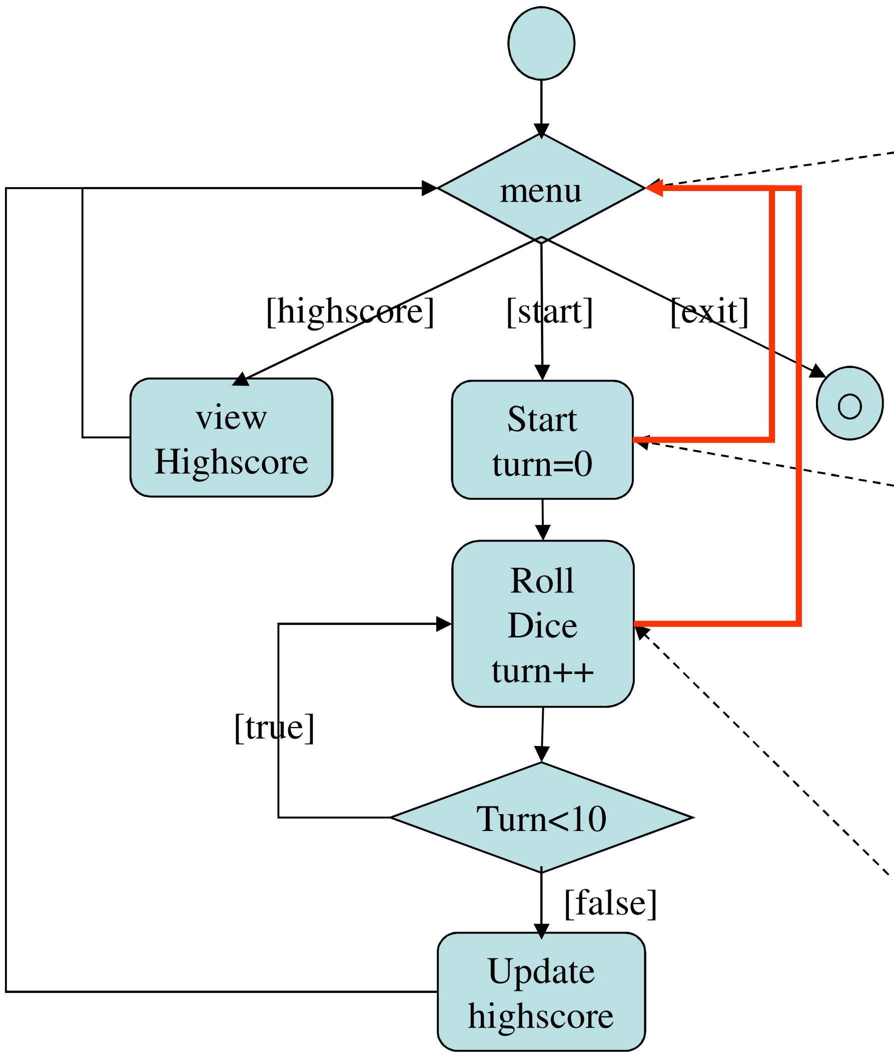
System Test

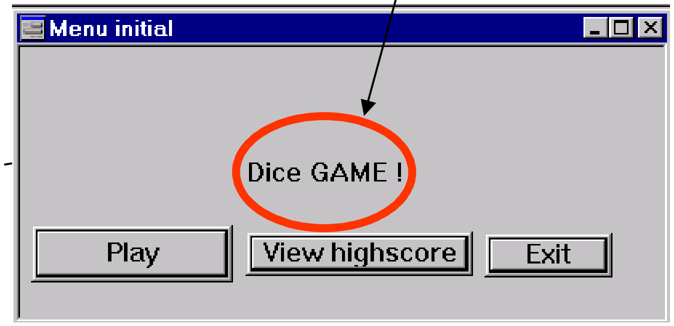


- Ok, functionality is there ...
- And conforms to the description of the use case !
- >8->

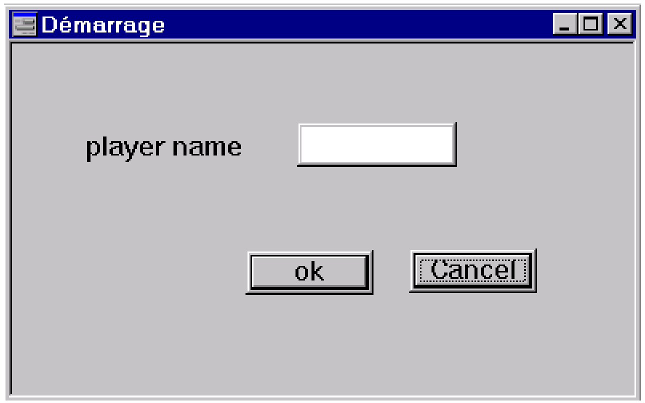
System Test

I forgot this one !

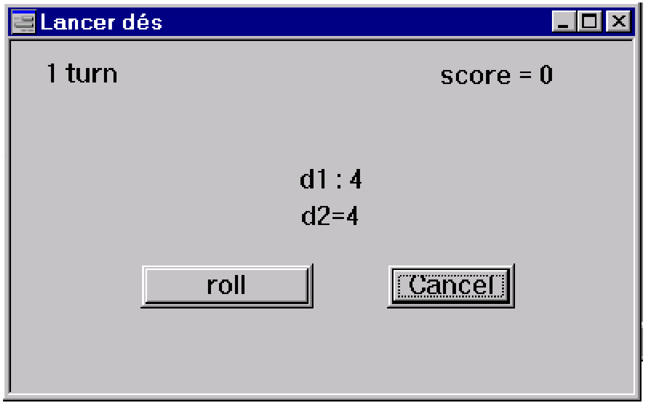




Dice GAME !



player name



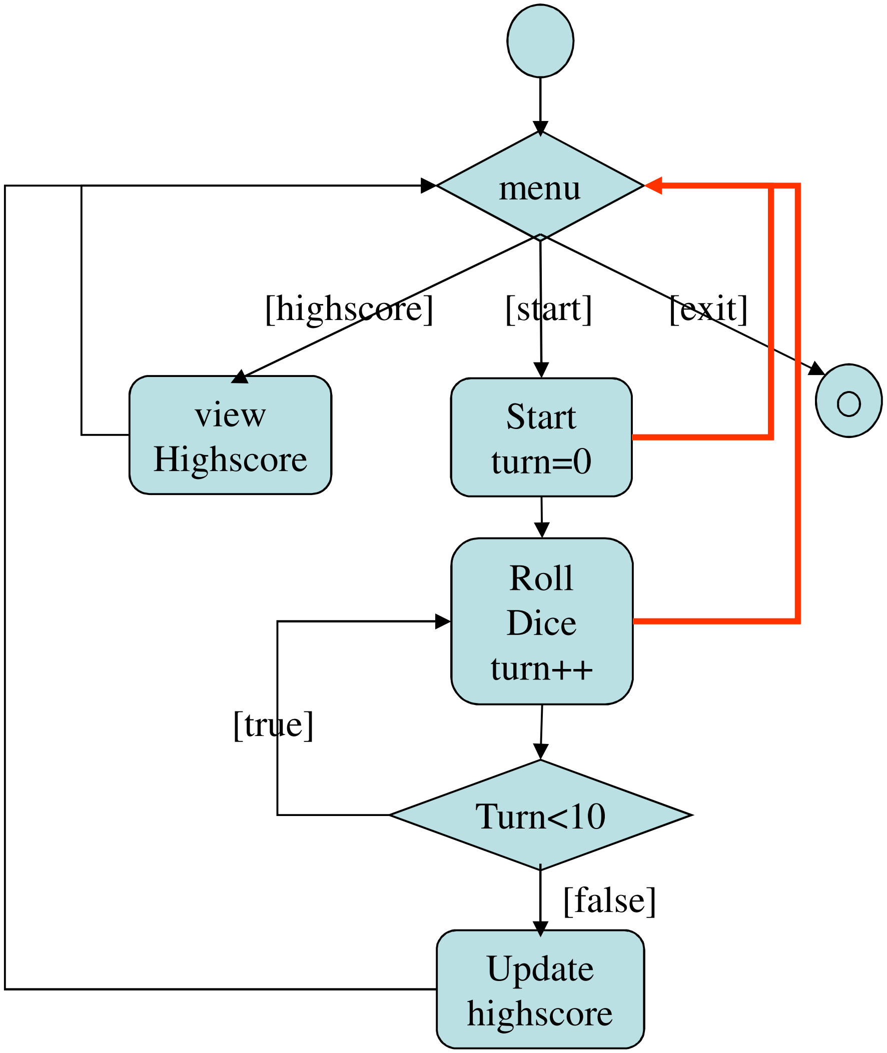
1 turn

score = 0

d1 : 4

d2=4

System Test



- Test all possible paths !
- Ex:
  - 1/Start
  - 2/ roll
  - 3/ cancel
  - 4/ highscore
  - 5/ exit

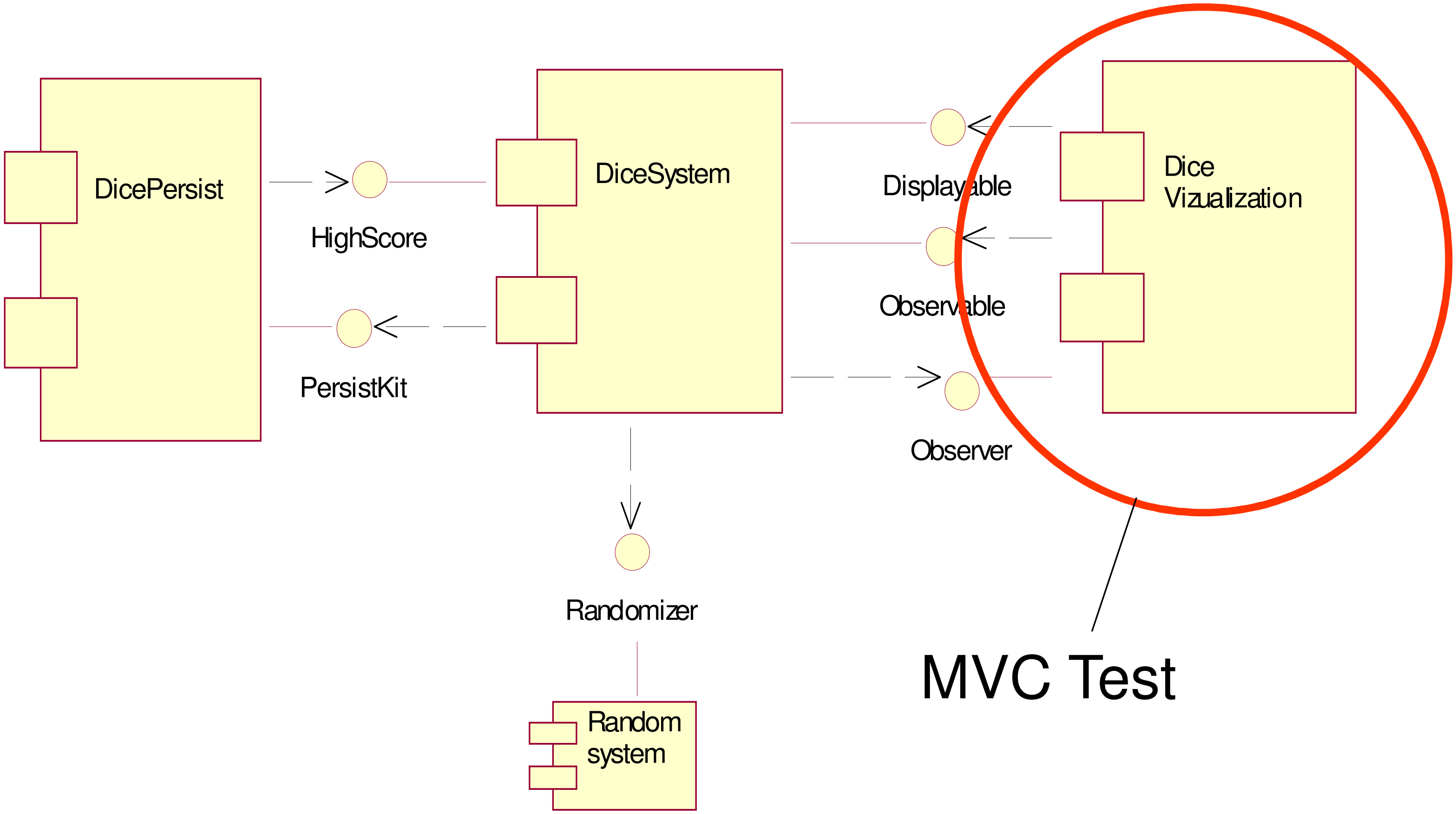
Problems Found

- Scenario 1 :
  - » start, roll*, highscore, quit : OK
- Scenario 2:
  - » highscore, : ko ! Bug
  - » Design bug:
    - DiceGame creates Highscore (start)
    - If Highscore before start : bug

Solution

```
package Core;

import UI.MainForm;
import Persist.*;
import java.awt.*;

class Main {
    public static void main(String args[]) {
       // SrKit srk=new SrKit();
       JdbcKit srk=new JdbcKit();
       DiceGame dg=DiceGame.getInstance();
       Frame f=MainForm.getInstance();
       f.setSize(300,300);
       f.show();
    }
}
```

Integration Test



Test Scenario

- Highscore, start, roll*
- If the MVC works properly, entry of a new highscore leads to redisplaying the list which is already opened !!
- Ok, it works…
- It is good to design well ...

Summary of this Application Design

- Requirements Analysis
  - Use-case + description
  - Activity diagram
  - UI prototyping
- Analysis
  - Dynamic Model : Collaboration, Sequence, state diagrams
  - Static Model : Class Diagram

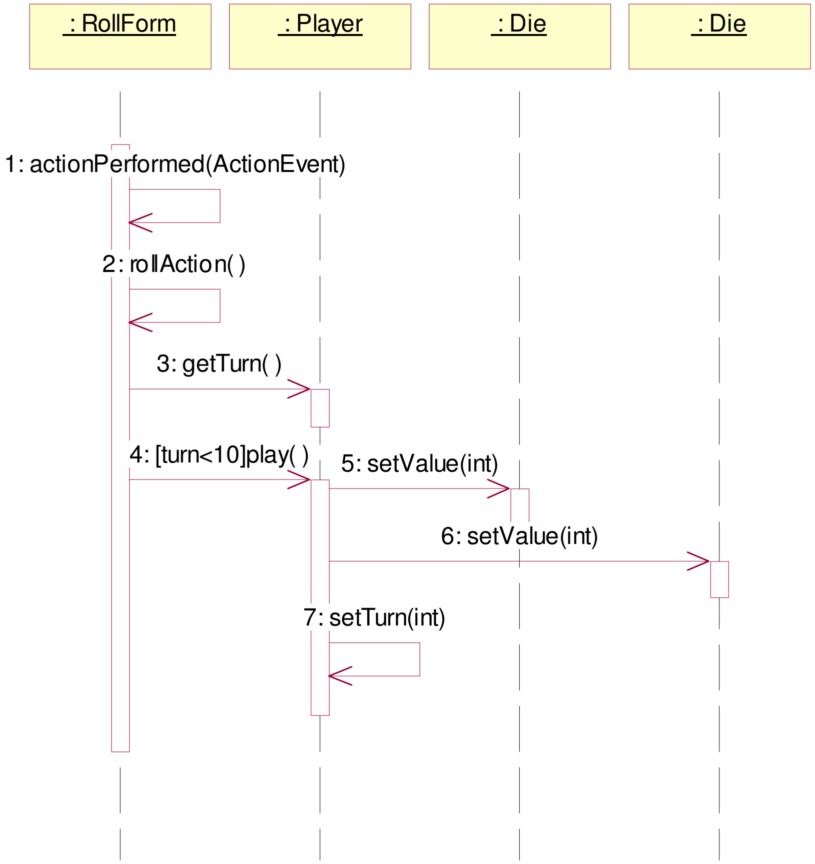
Design

- Architecture design (layer)
  - » Package diagram, component diagram, deployement diagram
- Technical classes used to ensure isolation :
  - » MVC pattern,
  - » Factory pattern
- Technical classes UI and persistence
  - » *Forms, Highscore*

Programming

- Simple conversion of design to Java
- For each UML model, it is possible to build a translation towards any target language
- Use « round-trip engineering » tools
- Programming PB : Need to update the analysis/design artifacts !!!

Feedback Problems Found at Coding !



- « auto-critique » to find the reason behind the problem.
- Improve process for next time !
- Here : analysis diagrams have not been redone !
- A software process must emphasize quality !

Testing

- Functionality control : Use-case diagram
- Conformance control : Activity diagram
- Integration tests : Component diagram
- Unit tests : not done

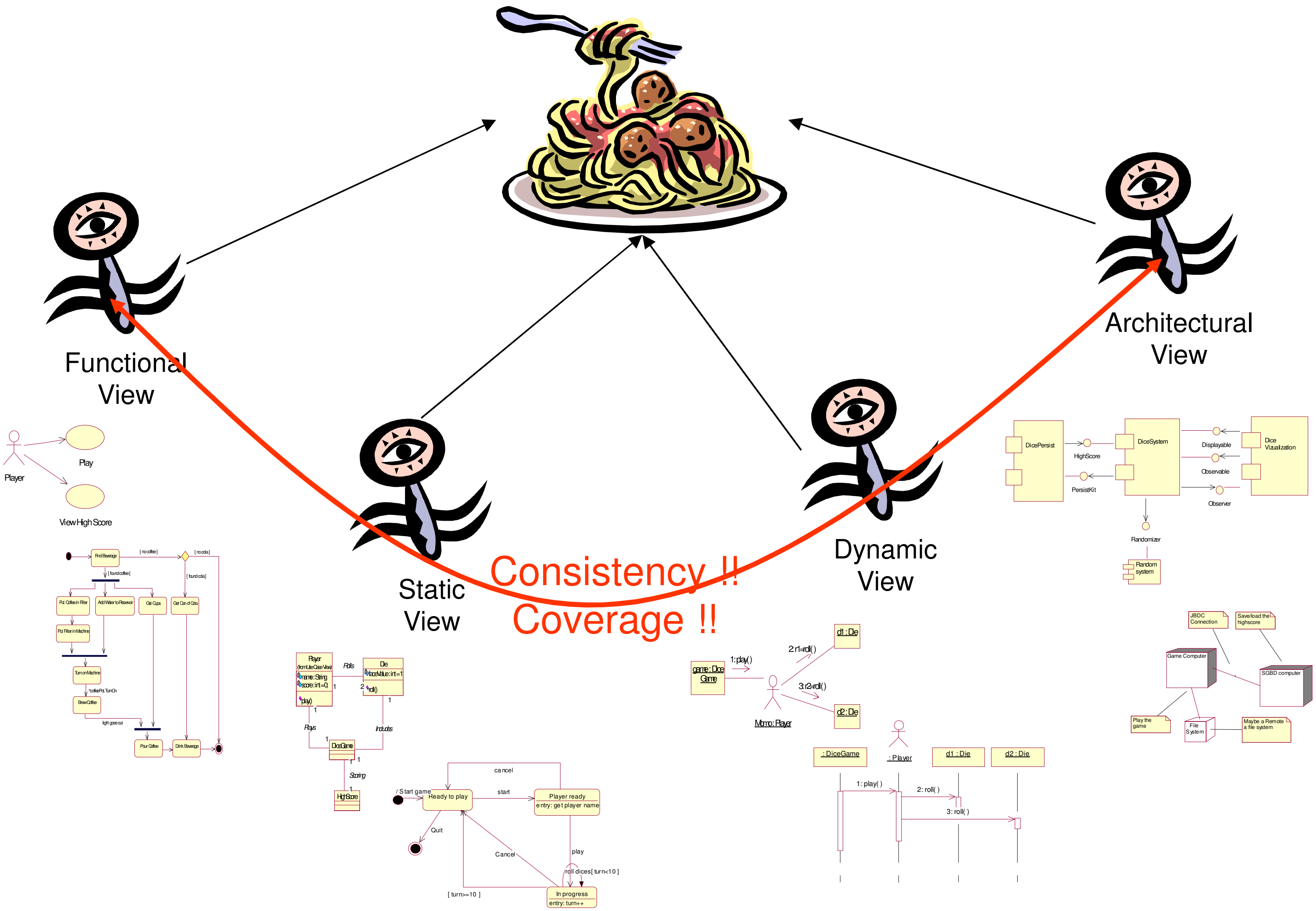
Paying Attention to Testing !

- Current vision is too simplistic !
- Must integrate testing as part of a quality management process (change management)
- Regression testing, automated testing (test suites, test generation, tools!!)

Conclusion for this Application

- Phase:
  - » Requirements analysis, analysis, design, implementation, etc.
- For each phase:
  - » Put together views for the same problem
  - » Static, dynamic, fonctional, architectural views

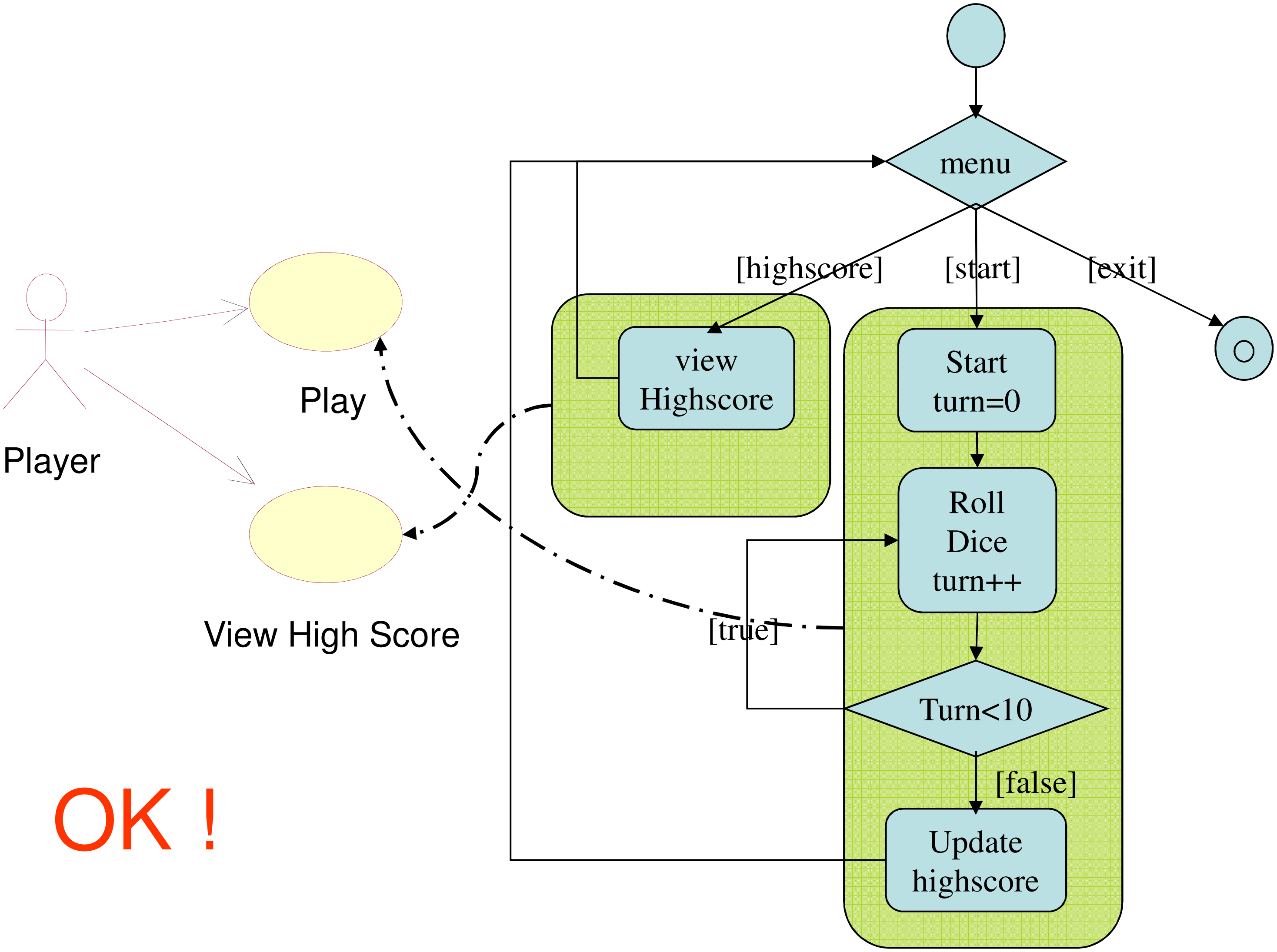
UML Usage Guideline: Consistency & Coverage



Consistency / Coverage

- Use-cases/Activity Diagrams
  - » An activity must always be assigned to a use-case
  - » All use cases must be implemented in the activity diagrams

Use-case / Activity



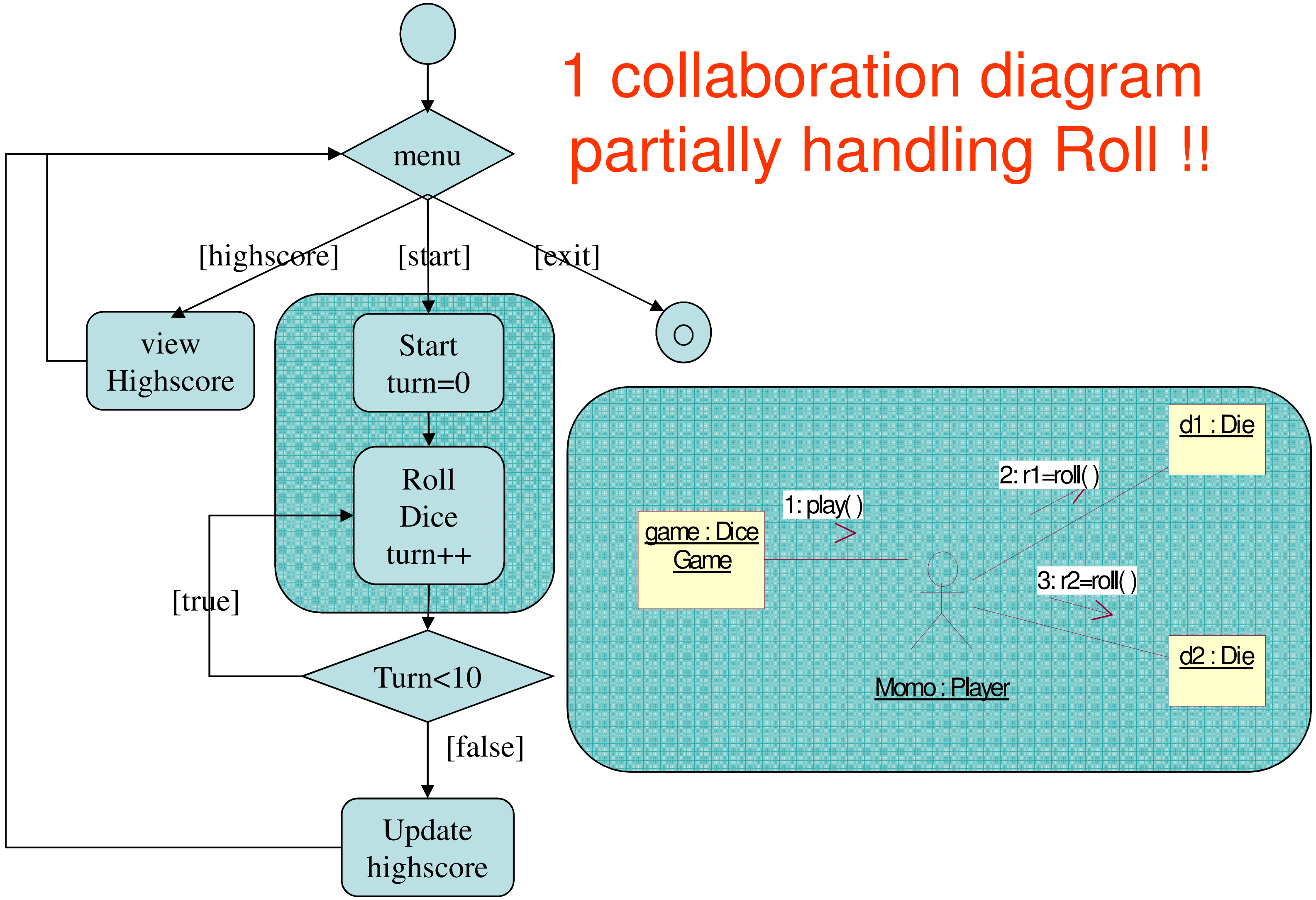
OK !

Activity / Collaboration

- All possible paths in the activity diagrams may be represented using collaboration diagrams !
- Beware of over-analysis !
- Only represent the most relevant scenarios !

Activity / collaboration

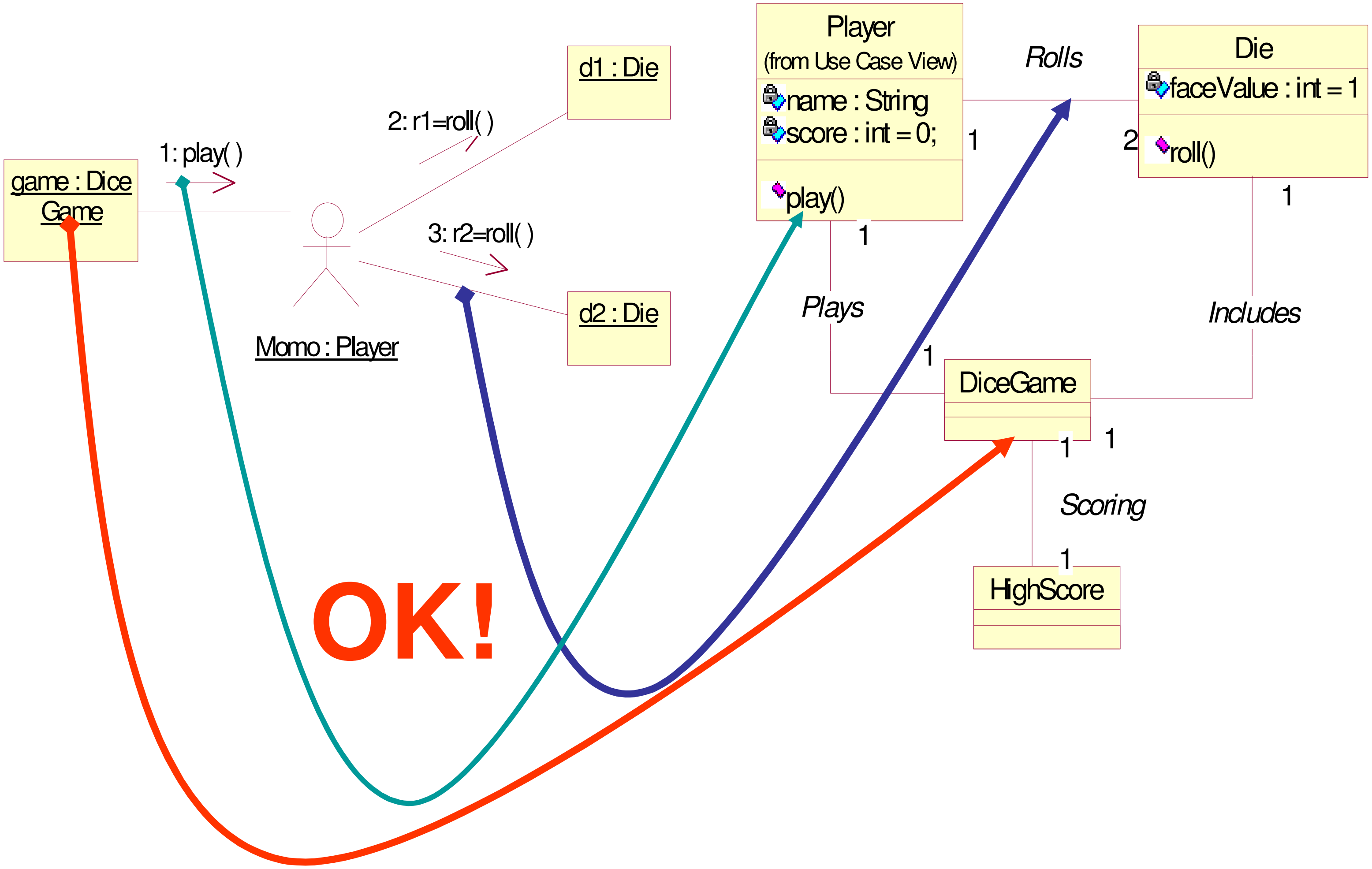
1 collaboration diagram partially handling Roll !!



Collaboration / Class diagram

- All the objects in a collaboration diagram have a type: the class diagram Class
- All the relationships in a collaboration diagram must exist or may be derived from the class diagram !
- Messages exchanged are methods in the class diagram !

Collaboration / Class Diagram

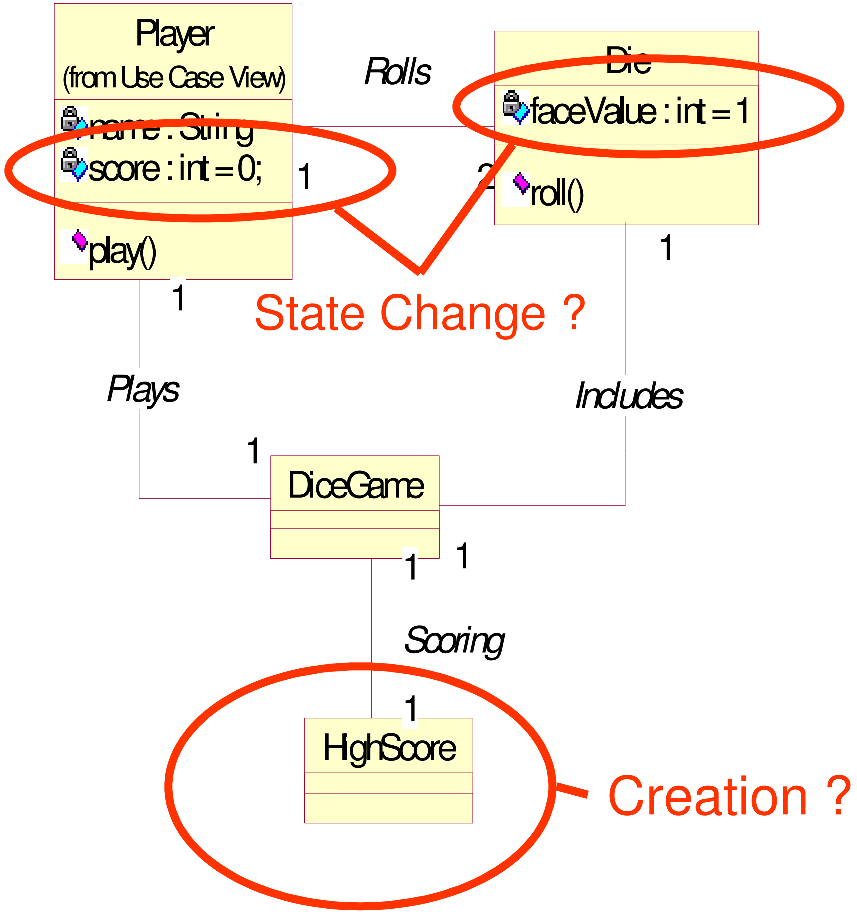


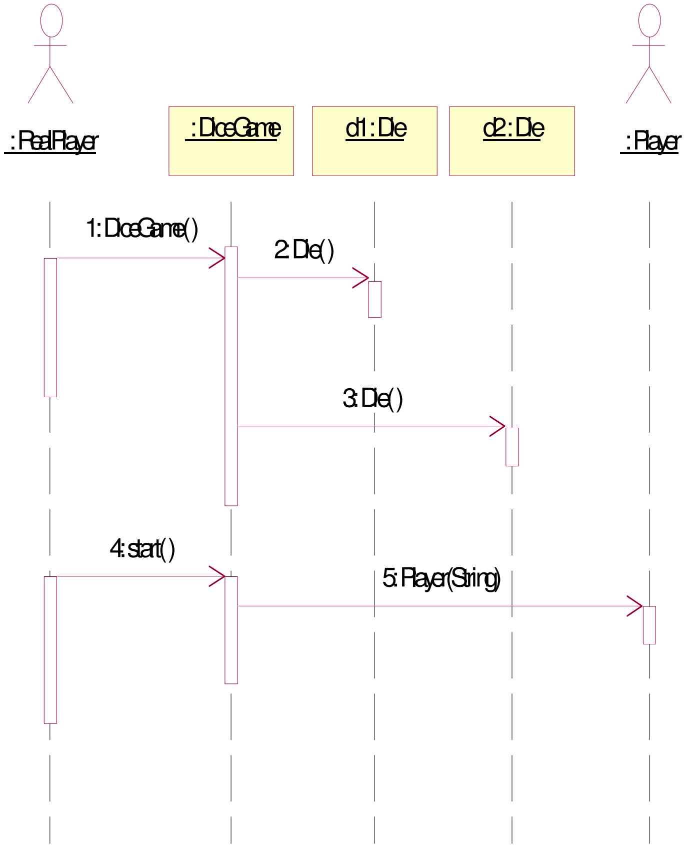
Class diagram / collab / sequence

- The complete dynamic model for relations must appear in at least one sequence or activity diagram
- Any attribute change must be represented in at least one activity or sequence diagram
- Object creation or destruction must appear in at least one dynamic diagram !

Class / Sequence

KO!





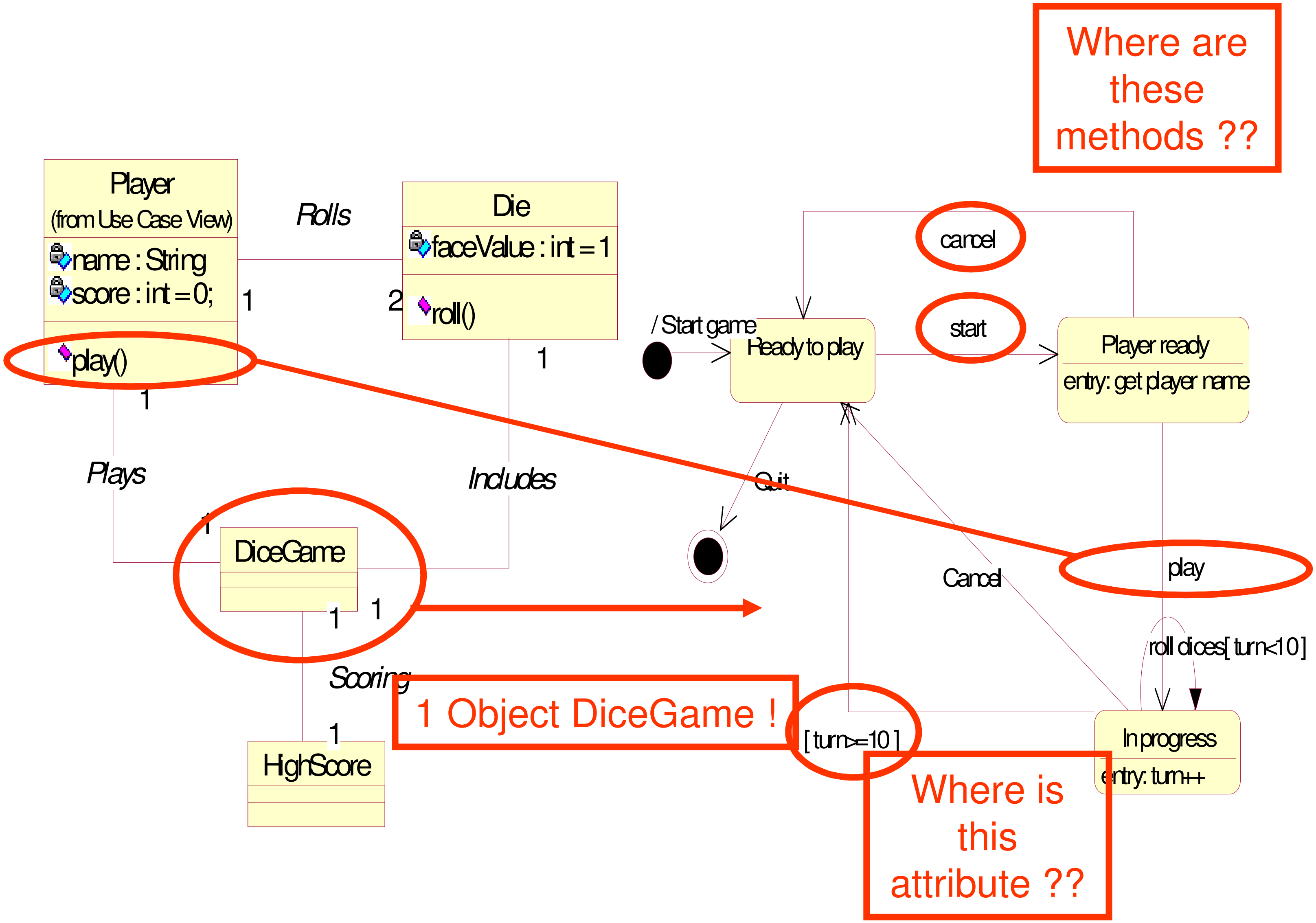
Class / Sequence

- A « good » solution:
  - » Follow the activity diagram to generate scenarios
  - » Follow the possible paths in the activity diagram
  - » If the activity diagram is not covered:
    - Granularity is not fine enough (it is the case here)
    - Class diagram over-specified

Class / State diagram...

- For each class, ask yourself if its state evolves with time ?
- If so, put together a state diagram…
- Every transition in the state diagram must be verified !

Class / State diagram !



Where are these methods ??

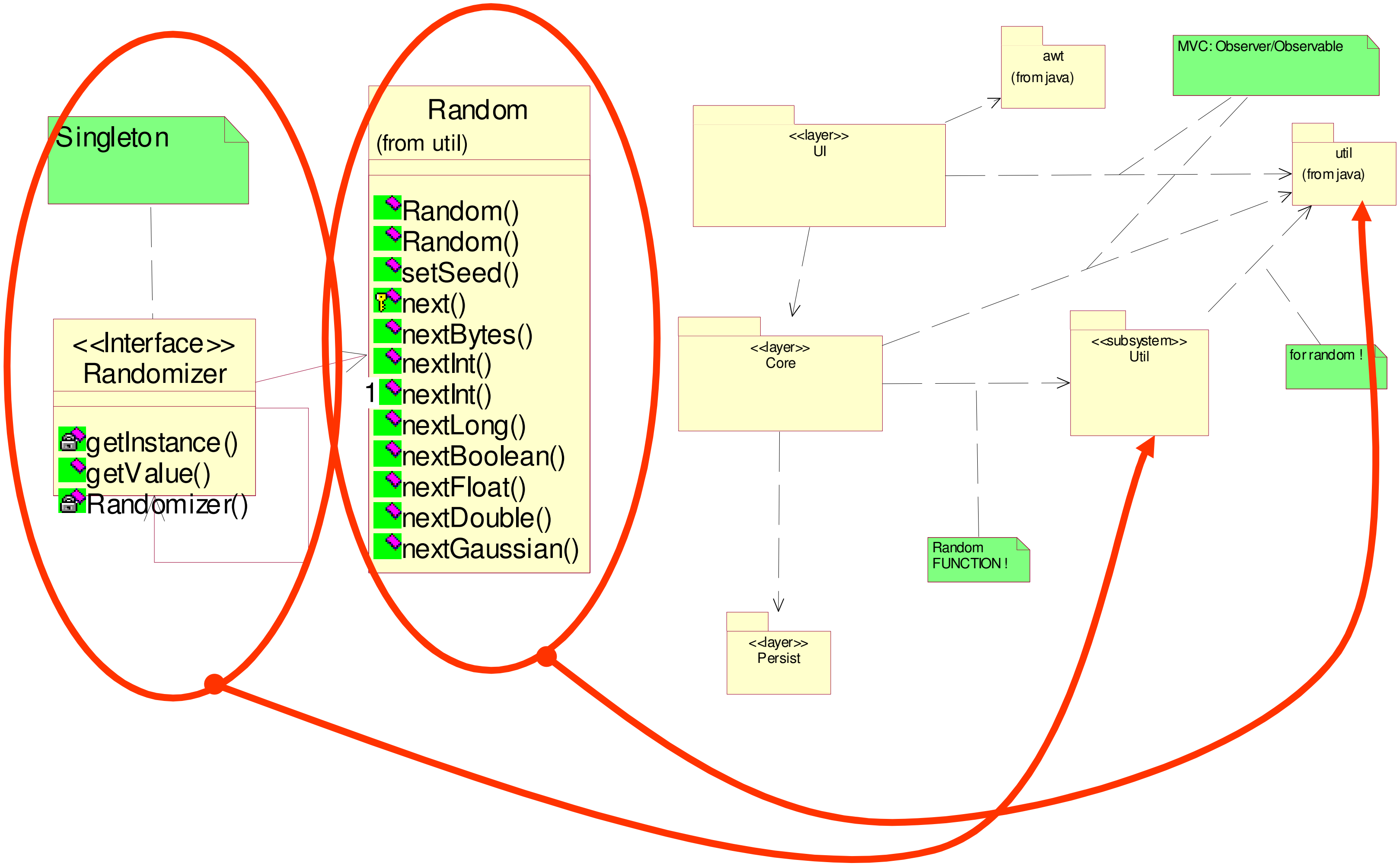
1 Object DiceGame !

Where is this attribute ??

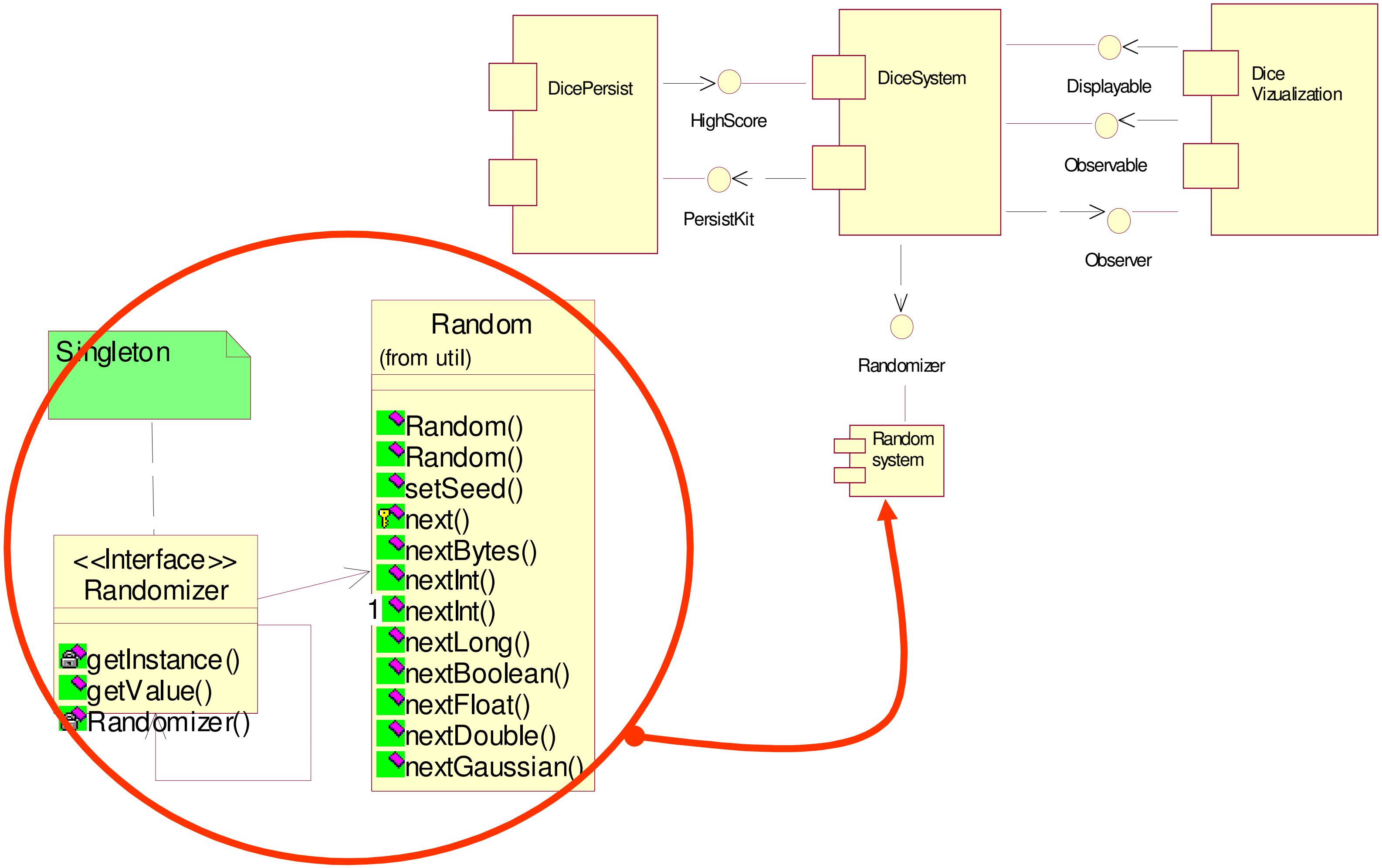
Class/Package/Component diagram (Design)

- Each class must be allocated to one package, which is itself part of the overall architecture
- Every class must also be part of a component that implements a set of functionalities in that architecture !!
- Otherwise the class is not part of the architecture !

Class / Package



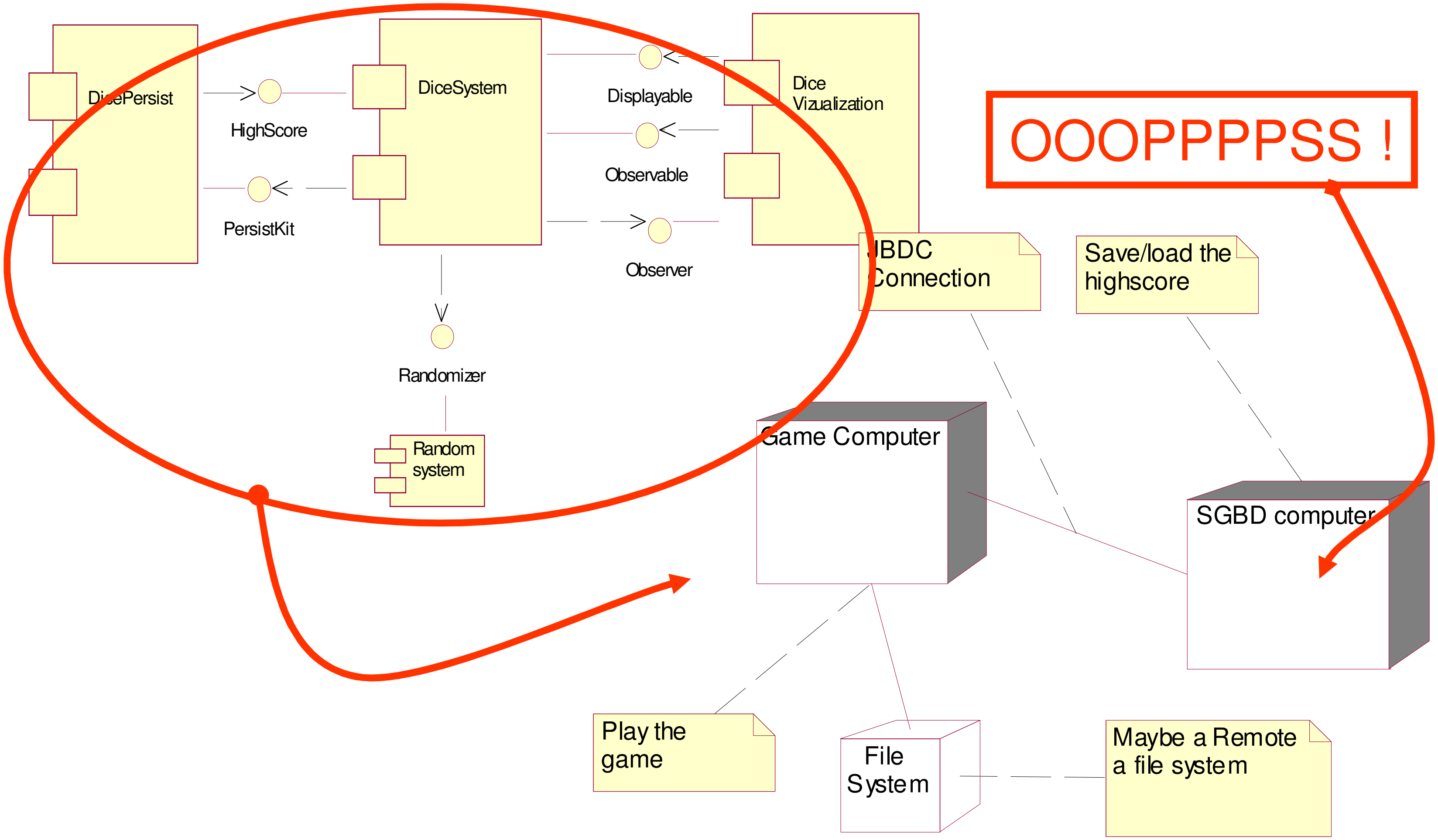
Class / Component



Component / Deployment

- Each component must be allocated to one execution unit in the deployment diagram !
- In general, a component cannot be part of two execution units…
- Every execution unit must have at least one component...

Component Deployment !



General Conclusion on Dice

- How different is it from a « directly coded » application ??

Bibliographie

- UML Distilled *Fowler&Scott*
- UML Toolkit *Eriksson&Penker*
- Applying UML and Patterns *Larman*
- Design pattern *GOF*
- System of patterns *Buschman&al*
- Penser objet avec UML&Java *Lai*
- Object oriented Analysis *Spadounakis*
- UML Specification *www.rational.com*

Agenda

1 **Introduction**

2 **Requirements Analysis**

3 **Requirements Modeling**

4 **Design Concepts**

5 **Sample Analysis and Design Exercise Using UML**

➔ 6 **Summary and Conclusion**

Summary – Key Detailed-Level Analysis and Design Objectives

- Requirements Models
  - Scenario-based (system from the user's point of view)
  - Data (shows how data are transformed inside the system)
  - Class-oriented (defines objects, attributes, and relationships)
  - Flow-oriented (shows how data are transformed inside the system)
  - Behavioral (show the impact of events on the system states)
- Requirements modeling covers different dimensions via flow-oriented models, behavioral models, and patterns
- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product
- Design principles establish and overriding philosophy that guides the designer as the work is performed
- Design concepts must be understood before the mechanics of design practice are applied
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight)
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve

Course Assignments

- Individual Assignments
  - Reports based on case studies / class presentations
- Project-Related Assignments
  - All assignments (other than the individual assessments) will correspond to milestones in the team project.
  - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
  - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  - A sample project description and additional details will be available under handouts on the course Web site

Team Project

- Project Logistics
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may <u>not</u> be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

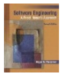
Team Project Approach - Overall

- Document Transformation methodology driven approach
  - Strategy Alignment Elicitation
    - Equivalent to strategic planning
      - i.e., planning at the level of a project set
  - Strategy Alignment Execution
    - Equivalent to project planning + SDLC
      - i.e., planning a the level of individual projects + project implementation
- Build a methodology Wiki & partially implement the enablers
- Apply transformation methodology approach to a sample problem domain for which a business solution must be found
- Final product is a wiki/report that focuses on
  - Methodology / methodology implementation / sample business-driven problem solution

Team Project Approach – Initial Step

- Document sample problem domain and business-driven problem of interest
  - Problem description
  - High-level specification details
  - High-level implementation details
  - Proposed high-level timeline

Assignments & Readings

- Readings
  - Slides and Handouts posted on the course web site
  -  Textbook: Part Two-Chapters 6-8
- Individual Assignment (assigned)
  - See Session 5 Handout: “Assignment #2”
- Team Project #1 (ongoing)
  - Team Project proposal (format TBD in class)
  - See Session 2 Handout: “Team Project Specification” (Part 1)
- Team Exercise #1 (ongoing)
  - Presentation topic proposal (format TBD in class)
- Project Frameworks Setup (ongoing)
  - As per reference provided on the course Web site

Any Questions?

Next Session: From Analysis and Design to Software Architecture