Object-Oriented Design

- Objective: Develop logical solution that fulfills the requirements
  - define the classes that will be implemented in an object-oriented programming language.
  - Assign responsibilities to software components
  - Identify and apply design patterns.

Artifacts of Analysis/Architectural Modeling

- Conceptual Model
  - Static Structure diagram(s)
  - Sequence diagrams
  - Glossary
  - Other models and documents?

- Architectural Model
  - Stakeholder Needs
  - Architectural views
Larman’s Approach to Design

- Develop real use cases
- Create interaction diagrams
- Develop design class diagrams
- Key issues:
  - Allocation of responsibility
  - Identification/application of design patterns

Design--Real Use Cases

- Real use cases describe user interaction with the system in concrete terms.
  - User interactions with system interface(s)
  - System interaction with interfaces
- Requires some definition of interface details.
- Remember, the “user” may be another software system or a hardware system rather than a human.
Design--Modeling System
Interactions

- Interaction diagrams
  - sequence diagrams
  - collaboration diagrams
- Show the time-ordered interactions among system objects
- If contracts have been specified for system operations during analysis/architectural modeling, these provide a good starting point.

Sequence Diagrams

1: insert card
2: request password
3: enter password
4: verify account
5: verify card w. bank
6: bank account OK
7: account OK
8: request type
9: select type
10: request amount
11: enter amount
12: process transaction
14: bank trans. OK
15: transaction OK
16: dispense cash
Etc.
Collaboration Diagrams

Elements of Collaboration Diagrams

Classes/Objects/Actors:

<table>
<thead>
<tr>
<th>Classname</th>
<th>:Classname</th>
<th>c1:Classname</th>
</tr>
</thead>
<tbody>
<tr>
<td>A class</td>
<td>An instance of class Classname</td>
<td>A named instance of class Classname</td>
</tr>
</tbody>
</table>

Links:

: class1

: class2

Denotes an instance of an association between class1 and class2
Collaboration Diagram Elements--Continued

Messages:

`:class1` — `:class2`

```
 n:message_name(params)
```

Sequential order of this message.

Direction of message flow

May also designate a return value:

```
 n:return_value=message_name(params)
```

Collaboration Diagrams--Continued

- Additional Elements (see Chapter 17 of Larman for syntax and examples:
  - iterative messages
  - conditional messages
  - alternative paths
  - multiobjects
Design of Collaborations

• During the design of collaborations, important design decisions must be made.
  – Methods assigned to classes
    • operation
    • parameters
    • return values
  – Internal data (state) of objects is identified
  – Interactions among classes are identified.
  – Internal flow-of-control of objects is identified.
Using Patterns to Build Collaborations

- Design pattern: capture standard solutions (structures) that have evolved over time and have been successfully applied to previous problems.
- Why use patterns:
  - reuse
  - faster/more robust design
  - improved communication

Larman’s Design Patterns

- GRASP--General Responsibility Assignment Patterns
  - Expert
  - Creator
  - High Cohesion
  - Low Coupling
  - Controller
  - Polymorphism
  - Pure fabrication
  - Indirection
  - Don’t Talk to Strangers

Basic Patterns

Advanced Patterns
“Gang of Four” Patterns

- Design Patterns--Elements of Reusable Object Oriented Software, by Gamma, helm, Johnson, and Vlissides
  - Creational Patterns
    - Abstract Factory
    - Builder
    - Factory Method
    - Prototype
    - Singleton
  
“Gang of Four” Patterns--Continued

- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor
Overview of the GRASP Patterns

• Expert
  – Assign a responsibility to the information expert--ie. The class that has the necessary information to carry out the responsibility.
  – Basic idea:
    • To what class or object should a given responsibility be allocated--e.g. responsibility for authorizing an ATM transaction?
    • Identify the class that has the necessary information.
    • Assign a method to this class to carry out the responsibility.

Expert Pattern Example
ATM Transaction Authorization: Who is responsible?

[Diagram showing relationships between entities such as Consortium, Bank, Cashier, ATM, etc.]
Expert Pattern--Benefits and Liabilities

• Benefits:
  – Low coupling among objects--objects use their own information to carry out responsibilities
  – High cohesion--behavior is distributed across classes that have the required information.

• Liabilities:
  – May ignore higher-level structuring issues
  – Could result in “over-distribution” of responsibilities.
GRASP Patterns--Continued

- Creator Pattern: Assign class B the responsibility to create instances of class A under any of the following circumstances:
  - B aggregates objects of class A.
  - B contains objects of class A
  - B records instances of class A objects
  - B closely uses objects of class A.
  - B has initializing data for class A objects.

Creation Pattern Example

Who should create CardAuthorization objects?

```
name

::Bank

CardAuthorization {new}
```

```
::CreateAuthorization{...}

::Create(...)
```
Creator Pattern--Benefits and Liabilities

• Benefits:
  – Low Coupling--since creator already has associations with created class

• Liabilities:
  – No real drawbacks--this is just common sense.
  – Choice of creator may not always be unique.

GRASP Patterns--Continued

• Low Coupling Pattern: Assign responsibility so that coupling remains low.
• Coupling: degree of interaction among objects
• (Potential) advantages of low coupling:
  – reduced complexity
  – more opportunities for reuse
  – easier to modify
GRASP Patterns--Continued

• High Cohesion Pattern: Assign responsibilities so that cohesion is high.
• Cohesion: The degree of interaction (relatedness) among responsibilities within as class.
• (Potential) advantages of high cohesion:
  – Good “packaging” of functionality
  – Enhances reuse.
  – Enhances maintainability

More About Coupling and Cohesion

• An analogy: Consider the design of a computer to be partitioned across three chips.
• Approach 1:
Design of a 3-Chip CPU--Second Approach:

Which approach makes more sense? Why?

Cohesion

• Meyers Defined Seven Levels of Cohesion
  – 7. Functional Cohesion
  – 7. Informational Cohesion
  – 5. Communicational Cohesion
  – 4. Procedural Cohesion
  – 3. Temporal Cohesion
  – 2. Logical Cohesion
  – 1. Coincidental Cohesion

(GOOD)
Types of Cohesion

• Coincidental
  – module performs multiple, unrelated actions
  – This amounts to arbitrary modularization

• Logical
  – module performs a set of related actions, one of which is selected by the calling module.
  – E.g., a module performing all input/output functions for a complex system.

• Temporal
  – module performs a series of actions related in time.
  – E.g., module containing all system initialization actions.

Types of Cohesion--Continued

• Procedural
  – module performs a set of weakly-connected actions corresponding to the sequence of steps in some operation
  – E.g., all of the operations involved in an ATM transaction

• Communicational
  – module performs a sequence of steps, related to some operation, which operate on the same data.
  – E.g., update a database, record update to audit trail, print the update.
Types of Cohesion--Continued

• Informational
  – module performs a set of independent actions, all of which operate on the same data structure
  – E.g., implementation of an Abstract Data Type

• Functional Cohesion
  – module performs one coherent action or achieves a single objective
  – E.g., “calculate sales commission.”

A Cohesion Example

Compute average daily temperatures at various sites.

Initialize sums and open files

Create new temperature record

Store temperature record

Close files and print average temperatures

Read in site, time, and temperature

Store record for specific site

Edit site, time, or temperature field
Coupling

• Five levels of coupling:
  – 5. Data Coupling (GOOD)
  – 4. Stamp Coupling
  – 3. Control Coupling
  – 2. Common Coupling (BAD)

Types of Coupling

• Content Coupling
  – one module directly references the content of the other.
  – E.g. module A branches to a local label of module B.

• Common Coupling
  – two modules share access to the same global data
  – E.g., modules use global variables to pass arguments
Types of Coupling--Continued

• Control Coupling
  – one module explicitly controls the logic of another
  – E.g. a control switch is passed as an argument

• Stamp Coupling
  – a data structure is passed as an argument but called module only operates on some individual components of the data structure
  – E.g., an employee record is passed to a module which only needs the salary field.

Data Coupling

• Data Coupling
  – all data exchanged by modules are homogeneous data items.
  – I.e., either simple data values or data structures in which all elements are used by the called module.
GRASP Patterns--Continued

- Controller Pattern
  - Assign responsibility for handling a system event to one of the following *controller classes*:
    - One representing the overall “system”, business, or organization
      - façade controller
    - One that represents an active real-world entity that might be responsible for the task
      - role controller
    - One that represents an artificial hander of all system events associated with some collaboration
      - use-case controller
Controller Pattern--Continued

- System event--generated by external actor
  - associated with system operations.
  - E.g. user selecting a function on ATM screen.
- Controller--object responsible for handling a system event.
- Possible choices for ATM transaction
  - System or ATM--façade controller
  - Teller--role controller
  - ATMTransactionHandler--use-case controller

Controller Classes--Which Type to Use:

- Façade controller
  - places all system event handling in a single class
  - may become too complex and incohesive if the number and range of system events is high.
- Role controller
  - attempts to mimic behavior of a human agent
  - may suffer from imperfect or awkward analogy
- Use-case controller
  - allocates controller responsibility on a per-collaboration basis
  - best choice if system has many events spread across several operations.
Controller Pattern Example

A Compiler:

Controller Classes--Additional Issues

- Separation of presentation (interface objects) from event-handling responsibility
  - E.g. GUI objects shouldn’t process user input events.
  - GUI object may select the appropriate controller class to handle a given event.