1. Object-oriented software development

Procedure-orientation
- We view work to be done mainly as a process (procedure)
- Implicit in the process are entities upon which the process is carried out.
- But procedure does not capture the entities well

Object-orientation
- We still view the work to be done as a process.
- But participants (entities) of the process are made explicit.
- The participants interact to get the work done.

Form and Function (I)
- Software has two sides: form and function

Form and Function (II)
- In mainstream OO processes (e.g., RUP), function and form grow together.
  - Baseline architecture takes shape early as primary use cases are built.
- In light-weight processes (e.g., XP), form is still important
  - “The simplest design that could possibly work.”
  - If design gets bogged down, refactor to fix it.

Process weights
- Agile methods: XP, Crystal, JAD, ...
- RUP, OMT, Fusion, ...
- Light weight: Few artifacts, Hot communication, Small team
- Heavy weight: Many artifacts, Cold communication, Large team
Proven process elements
- Use cases (requirement capture)
- Increments (staging strategy)
- Iterations (rework scheduling strategy)
- Patterns
- Unit tests
- Continual integration
- Review

A process and its workflow
1. Write use cases
2. For each use case
   1. Find domain model
   4. Find design model
   5. Implement and do unit tests
   6. Integration
   7. Use case tests
   8. Review

Workflow elements, roles and responsibilities

<table>
<thead>
<tr>
<th>Writing use cases</th>
<th>Analysts/domain experts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain modeling</td>
<td>Analysts/developers</td>
</tr>
<tr>
<td>Design modeling</td>
<td>Developers/analysts</td>
</tr>
<tr>
<td>Implementation and unit tests</td>
<td>Developers</td>
</tr>
<tr>
<td>Integration and tests</td>
<td>Developers</td>
</tr>
<tr>
<td>Use case tests</td>
<td>Team</td>
</tr>
<tr>
<td>Review</td>
<td>Team</td>
</tr>
</tbody>
</table>

An Example
- Problem: You are writing a program to display the movement of certain 2D particles, which move synchronously. The application hosts two presentations:
  - A 2D geometric view depicting the particles movement
  - A bar chart of number of particles by quadrants
- Moreover, certain events can cause the particle movement to reset:
  - Mouse clicked in the 2D view
  - Certain quadrant become devoid of particles

Increments and iterations
- There will be two increments:
  - Increment 1 will take cares of the random particles and the 2D view (I)
  - Increment 2 will add the bar chart view onto the existing design
    - Iteration 1: add bar chart view (II)
    - Iteration 2: fix the lost update problem (III)

Design strategy
- We will use straightforward design that works
- At the end of an iteration, we will examine the design by the following OOD principles
  - Program to an interface, not an implementation
  - Favor object composition over class inheritance.
  - Reuse
- We then make design changes using design patterns so that the principles are not violated
Increment I Design

Increment I Review
- Program works
- No immediate concern for reuse

Increment II, Iteration 1 Review
- (Function) Presentation generated change is not immediately propagated to peer. (III)
- Domain logic in UI component:
  - process() in BarChartFrame
Increment II, Iteration 1 fixes

- In `init()` of random points, notify the application object to notify views of the change
  - Add association from random points to the application object [Coupling increases]
  - Add an operation to the application, which will notify the views.

Solution: Observer

- Random points
  - manages the dependent views
  - Notify dependent views for change
  - Implement an interface
- Views
  - Depends the interface which random points implements
  - Delegate random points processing to an algorithm object (Strategy)*
- The application object
  - Create random points and views
  - Manage application level events (application exit)

Increment II, Iteration 1 Review (II)

- Program works correctly
- Design problems
  - The application object is doing too much (low cohesion)
    - It creates all relevant objects and associate them
    - It serves as the middleman for propagating view-initiated changes
  - The random particles is dependent on the application object for notifying the changes (high coupling)
  - Views depends on the implementation of random particles (reuse not possible)
Consequences

- The application object is made more cohesive
- Dependency from random points to the application object is reduced
- Random points and views are more cohesive
- Views can be reused
  - RandomPointFrame can serve as a view for subjects that produce 2D points.