

Object Oriented Design

Roadmap

So far –

- textual analysis + CRC cards provides a first breakdown into classes and responsibilities
 - a starting point, not a finished design

Next –

- design principles to determine if that breakdown is good
- strategies for improvement

Key Questions

Within a class –

- does everything in the class belong together?
(the answer should be yes)

Between classes –

- does this class depend on more of another class's details than it needs to?
(the answer should be no)

Why Bother With Good Design?

- *correct* and *well-designed* are different goals
 - a program can be fully correct and still badly designed

A well-designed program is –

- easier to get right
 - smaller, more focused pieces means fewer moving parts to keep track of – easier to verify that each piece is correct
- easier to understand
 - a familiar, well-organized structure lets a reader recognize what is going on
- easier to change
 - good design contains the ripple effect rather than letting it spread

Cohesion – One Class, One Purpose

- a class should represent a *single concept* and have a *single purpose*
- *high cohesion* = everything in the class is there for a reason connected to that one purpose
- *communicational cohesion* – the tasks refer to or update the same data structure
 - e.g. set of operations defined on a stack
- *procedural cohesion* – the tasks are the steps for achieving a goal
 - e.g. algorithm for decoding an encrypted message
- *logical cohesion* – the tasks perform a similar operation
 - e.g. error handling or input and output

stronger
reason for
grouping

weaker
reason for
grouping

5

Example

```
public class Book {
    private String title_;
    private String author_;
    private String content_;
    private int curpage_;
    private String location_;

    public String getTitle () { ... }
    public String getAuthor () { ... }
    public String getContent () { ... } // get content of the entire book
    public String getPage ( int page ) { ... } // get content of the specified page
    public String getPage () { ... } // get content of the current page
    public void turnPage () { ... } // advance to the next page
    public Location getLocation () { ... } // location in the library
}
```

- three concepts bundled together – low cohesion
 - the book's basic info
 - how content is read page-by-page
 - where it is kept in the library

CPSC 329: Software Development • Fall 2026

6

Recognizing Low Cohesion

- warning signs
 - a class you can't summarize without "and"
 - instance variables that don't all relate to each other
 - the *god class* – one class doing too much of the whole program's data and/or logic
- code smells
 - large class – long, can't summarize in one sentence
 - noticed through size and responsibility count
 - divergent change – one class has to change for many unrelated reasons
 - e.g. Game class edited when a rule changes and when the console output format changes and when the save/load logic changes
 - noticed through history – unrelated commits touching the same class

CPSC 329: Software Development • Fall 2026

7

Separation of Concerns

- distinct functionality should be split into different classes
- separation of concerns vs cohesion
 - cohesion is about individual classes – does this class have one job?
 - separation of concerns is about choosing where the job boundaries are program-wide – what tasks are part of this job?
- drawing the boundaries well is what makes high cohesion possible
 - facilitates division of labor
 - e.g. one person works on the rules engine, one works on the UI, one works on the save/load logic
 - facilitates reuse
 - e.g. separating title/author/content from page-tracking and shelf location allows the core book concept to be used in places those things don't apply – a bookstore, a citation manager, etc

CPSC 329: Software Development • Fall 2026

8

Information Hiding

- design decisions should be segregated so the rest of the program isn't affected by them
 - e.g. should anything outside `TableauColumn` care if an `ArrayList` or a `LinkedList` was used to store the cards in the column?
- prioritize hiding the decisions most likely to change or which have the most impact if they do
 - not just “make everything private”

Encapsulation

- bundle data together with the methods that manipulate it
- encapsulation vs information hiding
 - information hiding is the *goal* – design decisions don't leak out
 - encapsulation is the *mechanism* – bundling data and methods together into a class
- advantages
 - reduced complexity – you only need to read a class's public methods, not its internal details, to use it
 - and code isn't written that depends on those internal details
 - protected integrity – state can only change through methods that enforce rules
 - robustness – limited interdependencies mean a bug in one place is less likely to corrupt something unrelated
 - flexibility – you can change the internals without breaking the callers

Recognizing Information Hiding Issues

- warning signs
 - non-private instance variables
 - revealing the internal representations – getters and setters (or a whole public interface) tied to how the data is stored
 - unjustified getters and setters
 - a collection or mutable object is returned directly from a getter with no copy or wrapper
 - a constructor stores a reference to a mutable parameter without copying it
- code smells
 - insider trading – a class accessing another's internal fields or methods directly
 - message chains – a long sequence of calls drilling through several objects
 - e.g. `order.getCustomer().getAddress().getCity()`
 - global data – data that is modifiable from anywhere in the code base
 - data class – a class with only fields and getters/setters