Patterns and Anti Patterns

DONT s

1. Control Class Bloat Anti Pattern
2. Anemic Model Anti Pattern
3. GOD Class Anti Pattern

DO s

1. CRC Technique
2. Inheritance versus Composition
3. Composite Pattern

Anti Pattern: Control Class Bloating

Types of Classes : Boundary, Entity, Control

Anti Pattern: Bloating of Control Class

Typical of Procedural Code Migration

Symptom: Separation of Data and Behavior

Anemic Model Anti Pattern

- Result of using Controller with lots of logic
-
-

Anti Pattern: Behavioral GOD Class

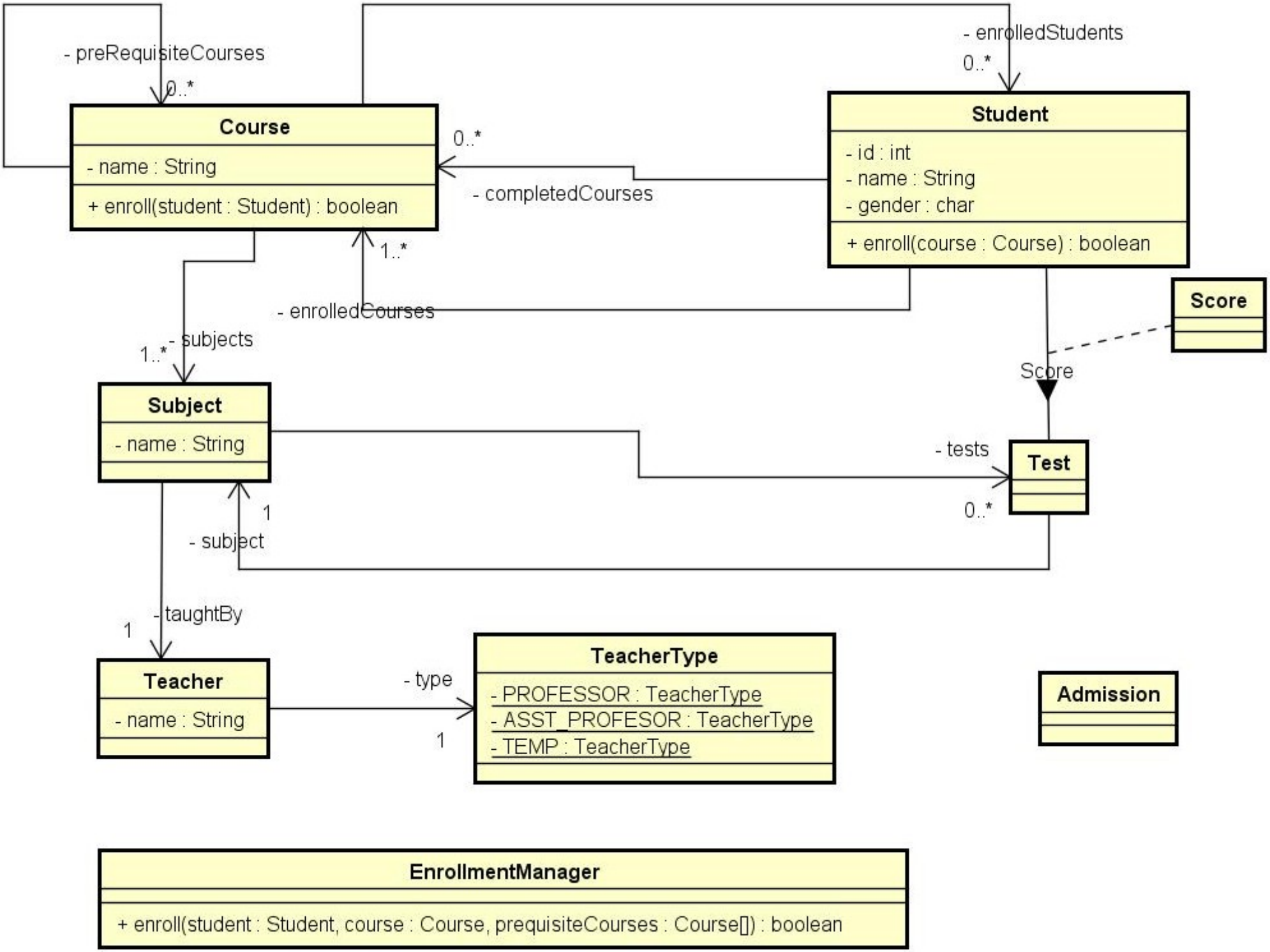
Performs most of work leaving minor details to a collection of trivial classes

Symptoms:

- Classes with names ending with Manager, System Subsystem etc.
- Classes with Names derived from Verbs
- Classes with too many fine grained accessor methods implies some GOD class is asking for this information

Solution: Always use CRC

- Class – Responsibility – Collaboration
- Identify Core Responsibilities.
- Identify what should be delegated and to whom
- Draw Draw Draw
- Identify CRC Violation
  - Too much calls to other objects
  - Classes with Names derived from Verbs. Dont make operation into class
  - Place agent classes during analysis and remove if not needed


- preRequisiteCourses
0..*
- enrolledStudents
0..*
Course
- name : String
+ enroll(student : Student) : boolean
0..*
- completedCourses
Student
- id : int
- name : String
- gender : char
+ enroll(course : Course) : boolean
1..*
- enrolledCourses
Score
1..*
- subjects
Score
Subject
- name : String
- tests
Test
0..*
1
- subject
1
- taughtBy
Teacher
- name : String
- type
1
TeacherType
- PROFESSOR : TeacherType
- ASST_PROFESOR : TeacherType
- TEMP : TeacherType
Admission
EnrollmentManager
+ enroll(student : Student, course : Course, prequisiteCourses : Course[]) : boolean

CRC Example – How to

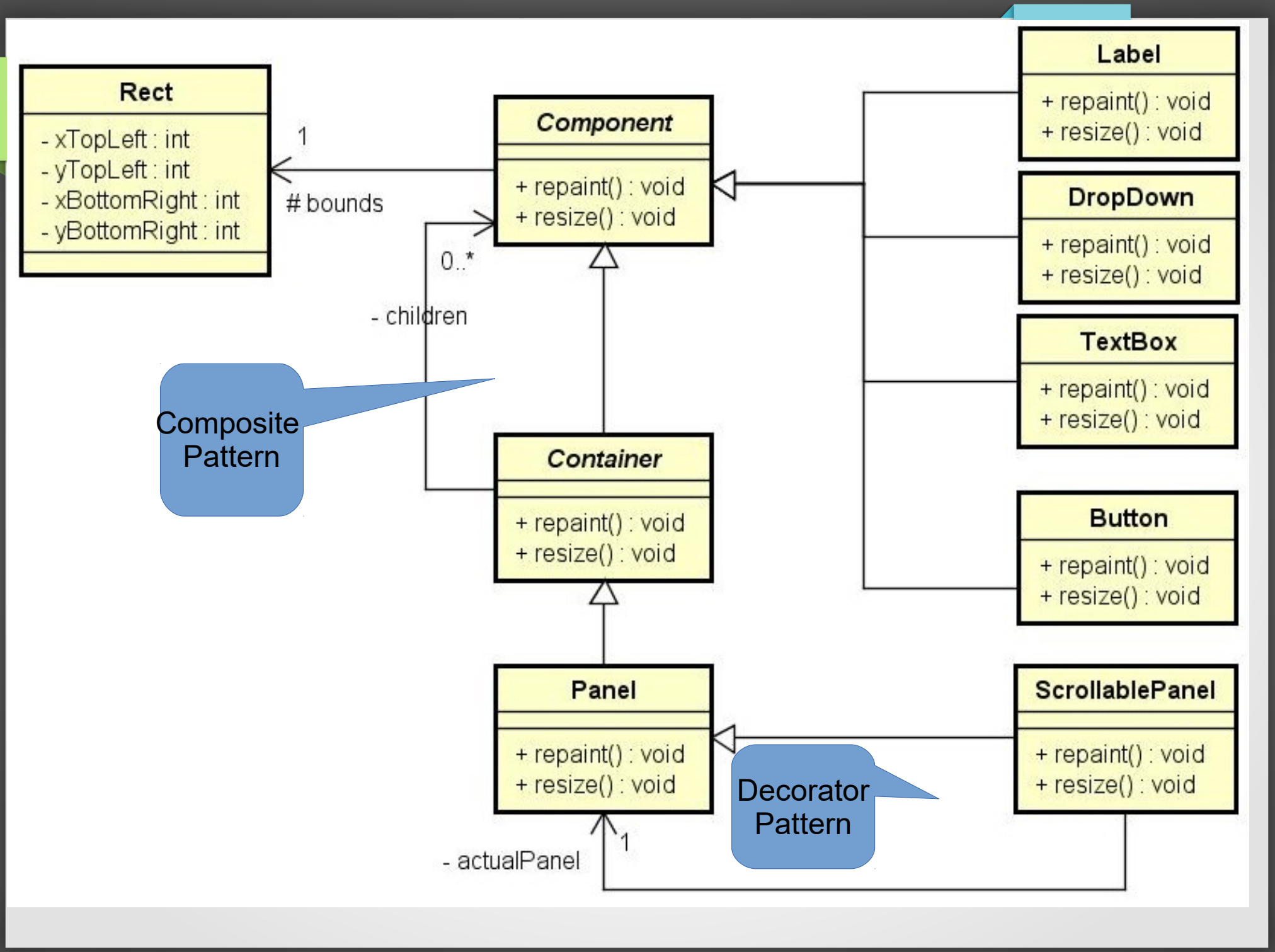
- Notice the Enroll function in 3 places. Which is right?
- Can EnrollmentManager class be eliminated?
- Use CRC Technique to identify whose responsibility it is to provide enrollment functionality?
- And Which classes should it collaborate to provide that functionality? (Eliminate Agent Classes during Design)
- Develop clusters of collaborating classes with DEFINED RESPONSIBILITIES
- Merge (Union) the clustered data+ behavior for final UML

NEVER use Inheritance for code reuse

- If you want reusable code, use
  - Composition
  - Helper Classes
- Use Inheritance for polymorphic behavior
- Roles are candidates for composition
-
- If/Else are symptoms of missing inheritance (already covered)

Composite Design Pattern

- Super Class – Sub Class relationship
-
- Containment with multiplicity
-
-
-


Rect
- xTopLeft : int
- yTopLeft : int
- xBottomRight : int
- yBottomRight : int
1
bounds
Component
+ repaint() : void
+ resize() : void
0..*
- children
Label
+ repaint() : void
+ resize() : void
DropDown
+ repaint() : void
+ resize() : void
TextBox
+ repaint() : void
+ resize() : void
Button
+ repaint() : void
+ resize() : void
Composite Pattern
Container
+ repaint() : void
+ resize() : void
Panel
+ repaint() : void
+ resize() : void
ScrollablePanel
+ repaint() : void
+ resize() : void
Decorator Pattern
1
- actualPanel

Next Session

- Simple Design Patterns
  - Singleton
  - Adapter
  - Proxy
  - Decorator
-

Next Session

- Interfaces
- Mock Objects
- Object Oriented Programming in Agent/Controller/Service Classes – Dependency Injection
- Creational Patterns -
  - Builder
  - Template Method
  - Factory Method
- Behavioral Patterns (Visitor,Observer,Command)