Designing Objects with Responsibilities

Several design principles to guide OO design decisions.

A Project Manager Presentation

Why You Should Listen

- ✧ Post-Mortem GRASP Analysis of Project
- ✧ Ethics Mid-Term will Include GRASP
- ✧ Good for the Business Side

Challenge

- ✧ Old-school advice on OOD
  - "After identifying your requirements and creating a domain model, then *add methods to the appropriate classes*, and *define the messages between the objects* to fulfill the requirements."
- ✧ New-school advice: there are consequences to:
  - where we place methods
  - they way objects interact in our design
  - specific principles can help us make these decisions

Recall: The big picture

- Object design is part of the larger modeling effort
- Some inputs to modeling:
  - What's been done?
  - How do things relate?
  - How much design modeling to do, and how?
  - What is the output?
  - Domain models, SDDs (Software Design Description)
- Some outputs from modeling:
  - object design (UML diagrams – interaction, class, package)
  - UI sketches and prototypes
  - database models
  - sketches of reports and prototypes

Design of objects

- Responsibility-driven design (RDD)
  - What are an object's *responsibilities*?
  - What are an object's *roles*?
  - What are an object's *collaborations*?
- The term is important
  - We are initially trained to think of objects in terms of *data structures* and *algorithms* (attributes and operations)
  - RDD shifts this by treating objects as having *roles* and *responsibilities*
- Objects then become
  - service providers
  - information holders
  - coordinators, controllers
  - interfaces, etc.

What is an object responsibility?

- Doing
  - doing something itself (creating an object; performing a calculation)
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate
- Responsibilities vary in "granularity"
  - big (hundreds of classes); example: "provide access to relational databases"
  - small (one method); example: "create an instance of *Sale*"

What is an object collaboration?

- ✧ Assumption:
  - responsibilities are implemented by means of methods
  - the larger the granularity of responsibility, the larger the number of methods
  - this may entail an object interacting with itself or with other objects
- ✧ Therefore:
  - fulfilling responsibilities requires collaborations amongst different methods and objects

What is an object collaboration?

- "Low representational gap":
  - RDD is a metaphor
  - "Objects" are modeled as similar to persons / systems with responsibilities
  - "Community of collaborating responsible objects"

"Notice that although this design class diagram is not the same as the domain model, some class names and content are similar. In this way, OO designs and languages can support a lower representational gap between the software components and our mental models of a domain. That improves comprehension."

–Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Craig Larman (2004) (pg 11)

A couple of parenthetical comments

- ✧ "Responsibility" here implies "non-interference"
  - This is idealized (i.e., in the real world this is not necessarily true in boundary cases)
  - However, it is suitable enough for object design
- ✧ Suitable for "programming-in-the-large"
  - As opposed to "programming-in-the-small"
  - This is a qualitative difference:
    - mass of details
    - amount of communication
    - this become overwhelming
- ✧ "Practical" as opposed to "theoretical"

GRASP

- ✧ Craig Larman's methodical approach to OO design
- ✧ GRASP stands for
  - General
  - Responsibility
  - Assignment
  - ◉ Software
  - ◉ Patterns

GRASP

- In essence:
  - a tool to help apply responsibilities to OOD design
  - designed (and meant to be used as) methodical, rational, explainable techniques
- They are “patterns of assigning responsibilities”
  - Note: the use of “pattern” here is slightly different from some that intended by the GoF book
- Recognizes that “responsibility assignment” is something we already do:
  - UML interaction diagrams are about assigning responsibilities

A bit more about “pattern” terminology

- “Patterns” as applied to architecture:
  - invented by Christopher Alexander
  - context that of a “pattern language” for designing spaces used by humans
- Classic design patterns for software
  - invented by the Gang of Four (GoF) (Gamma, Johnson, Helm, Vlissides)
- “Patterns” is now a rather “loose” term
  - Conferences exist to look at patterns
  - "Pattern Language of Programs" http://hillside.net/plop/2007/
- Larman has a special meaning for his GRASP patterns

What is a GRASP pattern?

- A named and well-known problem/solution pair
- General enough to be applied to new contexts
- Specific enough to give advice on how to apply it
- Especially important for novel situations
- Also comes with a discussion of:
  - trade-offs
  - Implementations
  - variations

Specifically GRASP

- Defines nine basic OO principles
  - basic building blocks for OO designs
- Meant to be “pragmatic”
- Meant to be a “learning aid”:
  - aids naming the group of ideas
  - aids in presenting insight of ideas
  - aids in remembering basic, classic design ideas
- Also intended for combination with:
  - a development process (UP)
  - an organizational metaphor (RDD)
  - a visual modeling notation (UML)

The Nine GRASP Patterns

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

Jargon alert: terms and definitions

- **coupling:** the degree to which each program module relies on other modules
- **cohesion:** a measure of how well the operations in a module work together to provide a specific piece of functionality
- **encapsulation:** synonym for “information hiding”
- **reusability:** likelihood a segment of structured code can be used again to add new functionality with slight or no modification

Modules

- connected sequence of program statements, bounded together under a name

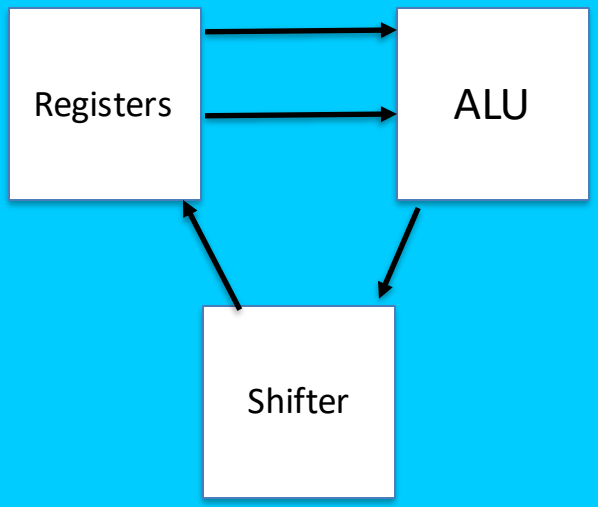
- Examples:
  - Functions
  - Objects

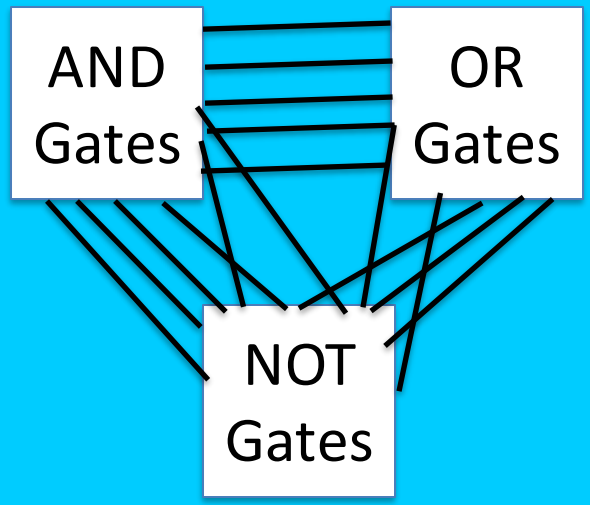
Modules

- ✧ Why?
  - Make code more readable
  - Make code more reusable

Modules

✧ Modularity is not inherently good





Modules

- ✧ High cohesion within modules
- ✧ Low coupling between modules

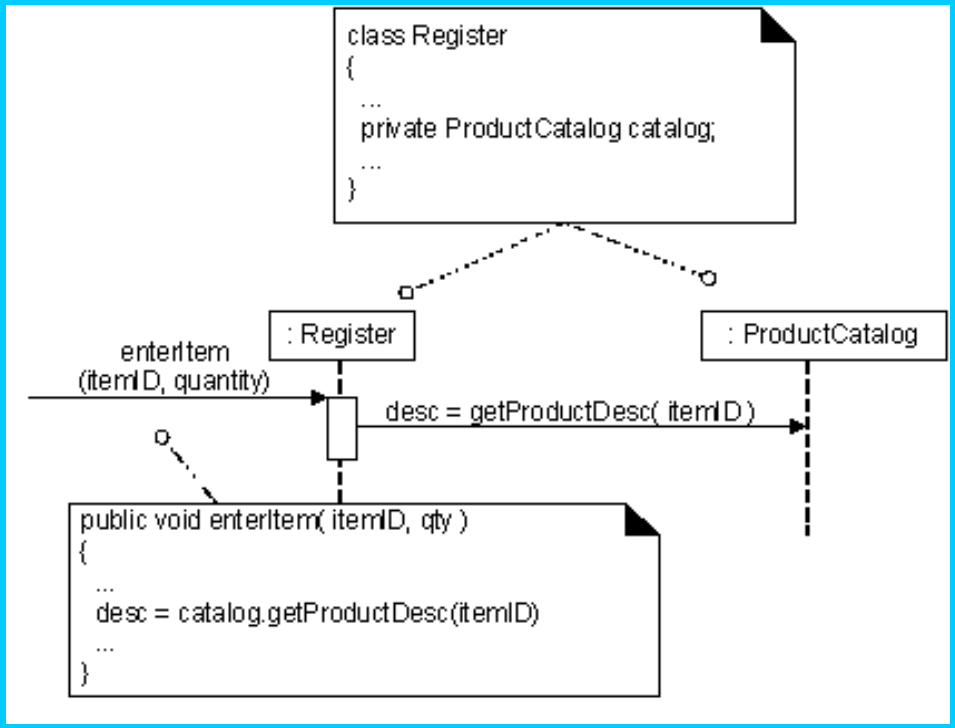
Modules

- ✧ Deliberate and informed
- ✧ Avoid one function modules
- ✧ Refactor with a reason

Visibility

✧ Definition:

- Ability of one object to "see" or to "have reference" to another



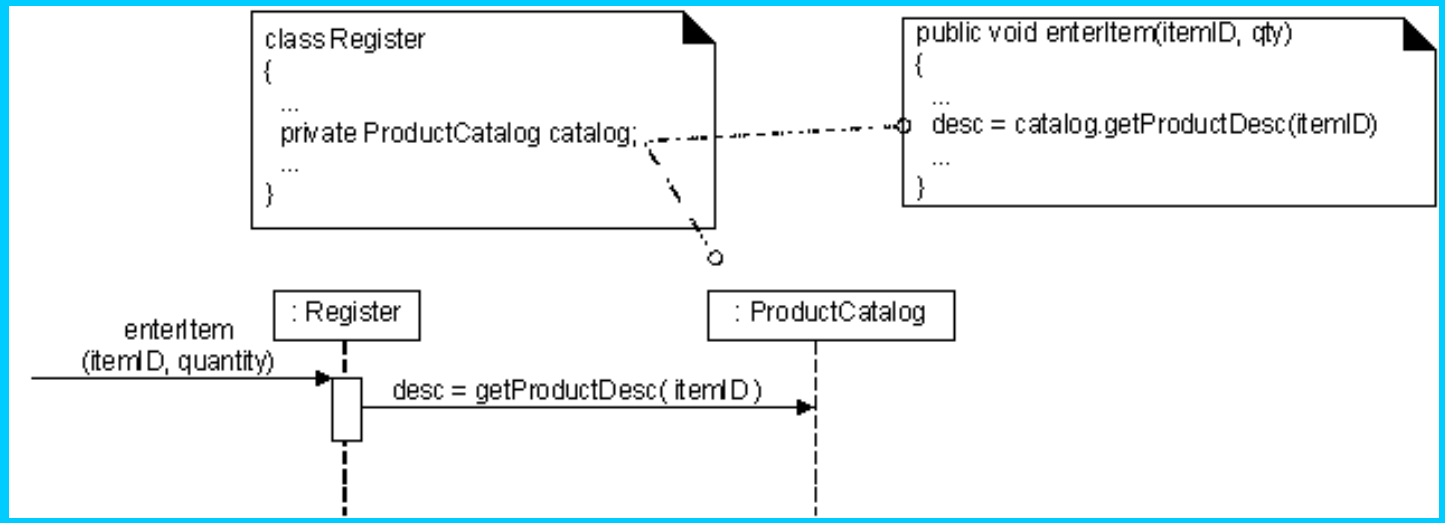
Kinds of Visibility

- ✧ Four kinds:
  - Attribute visibility
  - Parameter visibility
  - Local visibility
  - Global visibility

1. Attribute Visibility

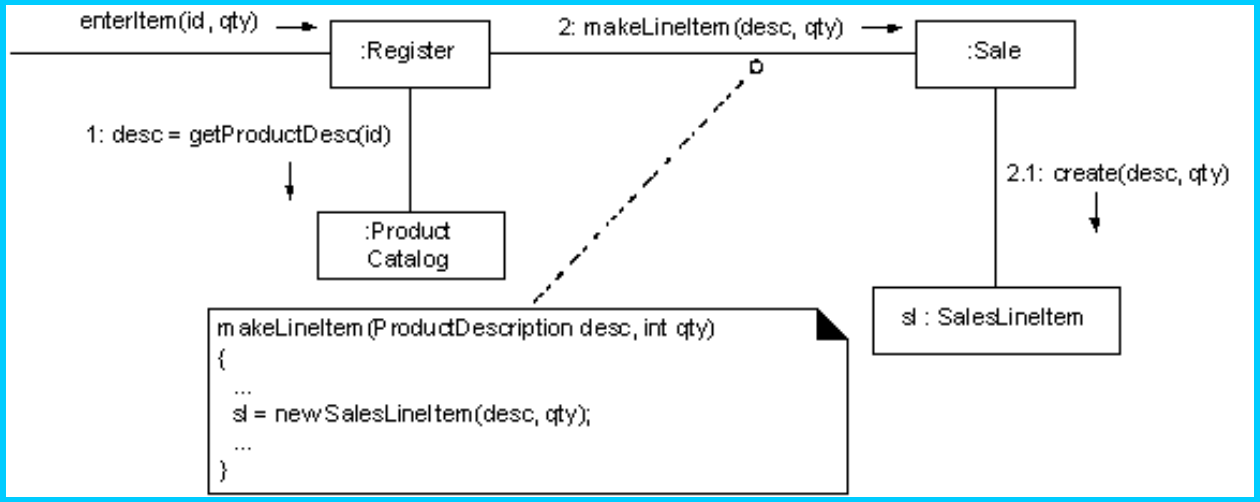
- ✧ Relatively permanent form
  - It exists if objects A and B exist
  - Declare a member variable (attribute) in the class

- ✧ Probably the most common form



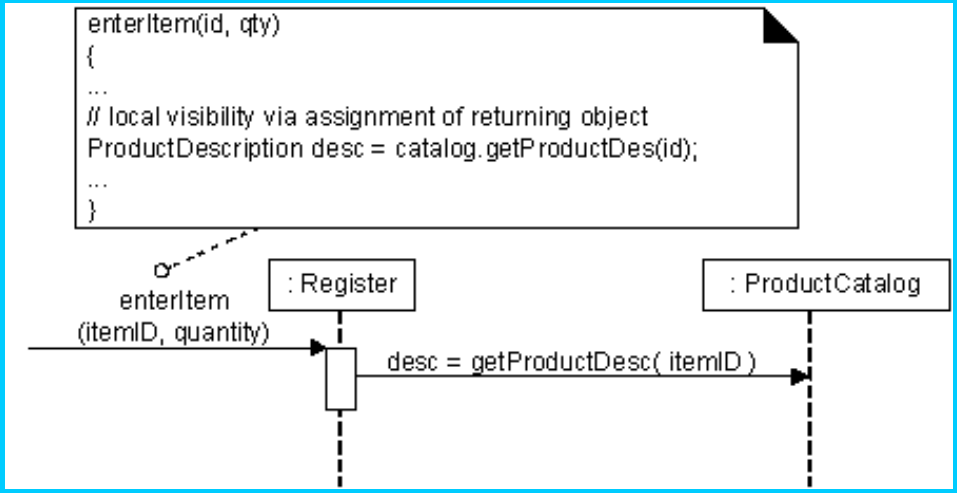
2. Parameter Visibility

- ✧ Exists when B is passed as a parameter to method of A
- ✧ Compared to Attribute visibility, this is relatively temporary
  - That is, persists only within scope of called method
- ✧ Can transform Parameter visibility to Attribute visibility



3. Local Visibility

- ✧ Exists when B is declared as a local object within a method of A
- ✧ Two ways to do this:
  - Creating a new local instance and assigning it to a local variable
  - Assigning the returning object from a method invocation to a local variable



4. Global Visibility

- ✧ Exists when B is global to A
- ✧ Has relatively permanent visibility, but also the least common
- ✧ Two ways to do this are by:
  - Ordinary global variables
  - Singleton Pattern

1. GRASP Creator

- Problem
  - Who should be responsible for creating a new instance of some class?
- Solution
  - Assign class B the responsibility to create an instance of class A if
    - B “contains” or compositely aggregates A
    - B records A
    - B closely uses A
    - B has initializing data for A (i.e., B is an *Expert* with respect to creating A)

1. GRASP Creator

- ✧ Benefits
  - low coupling
  - increased clarity
  - increased encapsulation
  - increased reusability
- ✧ Examples
  - Factory Method and Abstract Factory
  - Level Manager (creating objects)

2. GRASP Information Expert

- Problem
  - What is a general principle of assigning responsibilities to objects?
  - Recall: design model may end up very large (hundreds of classes and methods)
  - If assignment of responsibilities is well chosen, result is system more easily understood than one with poorly chosen responsibility assignment.
- Solution
  - Assign a responsibility to the information expert
  - This expert is the class that has the information needed to fulfill the responsibility

Key idea:

("Clear Statement" Rule): Start assigning responsibilities by clearly stating the responsibility.

2. GRASP Information Expert

- ✧ Benefits
  - Responsibility is placed on the class with the most information required to fulfill it
  - Low coupling
  - High cohesion
- ✧ Examples
  - Level manager (highest level)

3. GRASP Low Coupling

- ✧ Problem
  - How can our design support
    - low dependency?
    - low change impact?
    - increased reuse?
- ✧ Solution
  - Assign a responsibility so that coupling remains low.
  - Use this principle to evaluate alternative assignment of responsibilities.

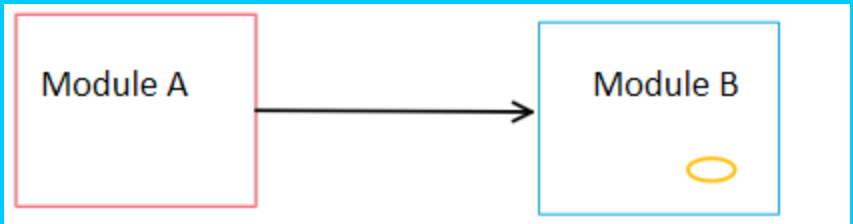
Types of Coupling

- 5 Levels
  - Content Coupling (Worst)
  - Common Coupling
  - Control Coupling
  - Stamp Coupling
  - Data Coupling (Best)

Content Coupling

✧ One module directly references contents of the other

Content Coupling Examples



Module A directly modifies Module B's Data

How to avoid Content Coupling?

1. Encapsulating data
2. Information hiding (Declaring them as private) and provide access using get and set methods

- Why is it bad?
  - Any changes to module B will require changes to module A first

Common Coupling

✧ Two module have written access to the same global data

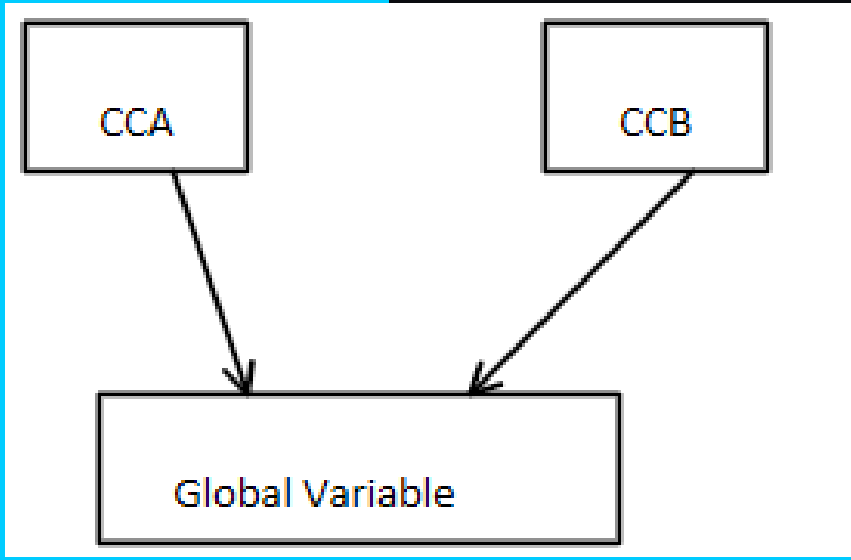
Common Coupling Examples

Enemy Module

```
public void takeDamage(int dmg)
{
    if(dmg > health)
    {
        health = 0;
        return;
    }
    health -= dmg;
}
```

NPC Module

```
public void receiveAttack(AttackCommand attack)
{
    health -= attack.damage;
    if (health < 0)
    {
        health = 0;
    }
}
```



Common Coupling

✧ Why is it bad?

- Difficult to reuse
- Difficult to determine all the modules that affect a data element
- Module is exposed to more data than necessary

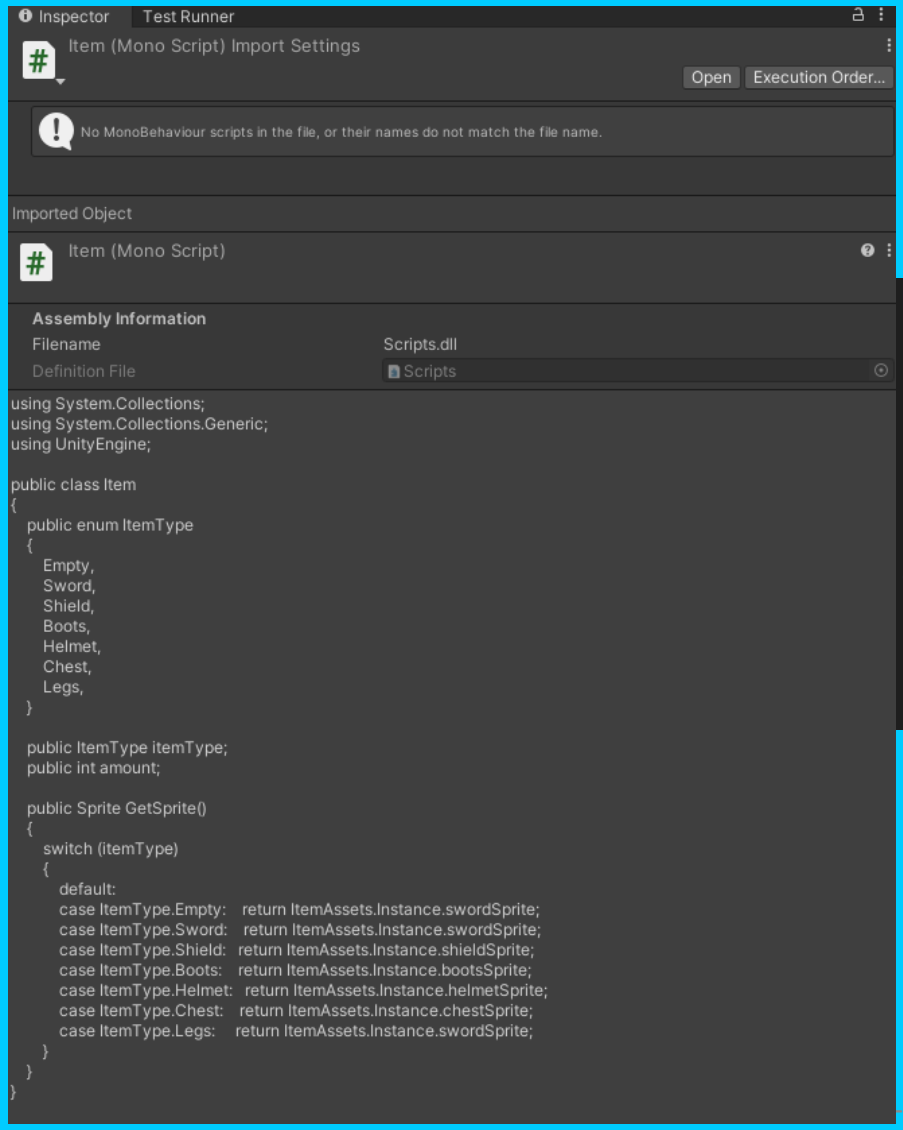
✧ Solution

- If there is a change in the declared variables in the singleton, only the singleton is affected.

Control Coupling

✧ One module passes an element of control to the other

Control Coupling Examples




```
bool moduleA(int x){
    if (x == 0)
        return false
    else
        return true
}

void moduleB(){
    //call moduleA by passing a value which controls its flow
    moduleA(1)
}
```

Control Coupling

- ✧ Why is it bad?
  - Module B must know the internal structure of Module A
  - Will affect reusability
- ✧ Solution:
  - Variable made nonpublic

Stamp Coupling

- ✧ Data structure is passed as parameter, but called module operates on only some of individual components

Stamp Coupling Examples

```
int health_scalar;
//multiplier for health amount
int health_multiplier;
//
int damage_scalar;
//multiplier for damage amount
int damage_multiplier;
//length of effect time
float effectTime;
```

```
public void health_effect(int health_scalar, int health_multiplier)
{
    //Will be added in the fututre
}

public void damage_effect()
{
    //Will be added in the fututre
}
```

Stamp Coupling

- ✧ Why is it bad?
  - Passes more data than necessary
  - Affects understanding
- ✧ Solution
  - Only send the information that is needed

Data Coupling (Best)

- ✧ Every argument is either a simple argument or a data structure in which all elements are used by the called module

Data Coupling Examples

```
int i;

int example(int i)
{
    i = i + 10;
    return i;
}

int example2(int i, int j)
{
    j = j + i;
    return j;
}

int main()
{
    int j = 50;
    int k = 10;
    j = j + k;

    example(i);
    example2(j,i);

    return 0;
}
```

```
public AttackCommand(NPC issuer, NPC target, int power)
{
    sender = issuer;
    receiver = target;
    damage = power;
}
```

Data Coupling

- ✧ Why is it good?
  - Maintenance is easier
  - Loosest form of coupling

4. GRASP Controller

- Problem
  - What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- Solution
  - Assign responsibility to a class based on one of the following
    - Class is "root object" for overall system (or major subsystem)
    - A new class based on use case name

4. GRASP Controller

- ✧ Benefits
  - Acts as a delegator, allowing specificity
  - Keeps operation logic away from the GUI
  - Easy to change hardware inputs
- ✧ Examples
  - A single controller classes for all events (facade)
  - Controller performs many of the events itself rather than delegate work
  - Controller maintains a lot of state information about the system (e.g., variables and attributes)

5. GRASP High Cohesion

- ✧ Problem:
  - How can we keep objects in our design
    - focused?
    - understandable?
    - manageable?
- ✧ Solution:
  - Assign a responsibility so that cohesion remains high.
  - As with low coupling, use this evaluate alternatives to placing responsibilities in objects

Cohesion

- ✧ 7 Levels
  - Coincidental
  - Logical
  - Temporal
  - Procedural
  - Communicational
  - Functional
  - Informational

Coincidental Cohesion (Worst)

✧ Modules performs multiple, unrelated actions

Coincidental Cohesion (Worst)

..	
SoundEffects	add volume tests
.gitkeep	setup up file structure with empty folders
AudioPlayer.cs	make volume a property and make audio source private
AudioPlayer.cs.meta	create audio player for mvp
AudioStress.cs	add volume tests
AudioStress.cs.meta	add volume tests
CameraFollow.cs	make a basic player camera
CameraFollow.cs.meta	make a basic player camera
EW_Feature.txt	add feature readme
EW_Feature.txt.meta	Desktop Install
EricSrc.asmdef	make a basic player camera
EricSrc.asmdef.meta	Updated player and added assemblies
SoundEffects.meta	Alpine-Chronicles

–Joseph

Coincidental Cohesion (Worst)

- ✧ Degrades maintainability
- ✧ Harms reusability

- ✧ Fix
  - Break into separate modules

Logical Cohesion

- Module can perform a series of related actions that are chosen by the calling module

Logical Cohesion

```
3  void logicalCohesion(int var, type v1, type v2, type v3)
4  {
5     switch var
6     {
7        case 0:
8           //Some code
9           break;
10       case 1:
11          //Some code
12          break;
13       case 4:
14          //Some code
15          break;
16       case 27:
17          //Some code
18          break;
19       /*
20       * More cases
21       */
22       default:
23          //Some code
24    }
25 }
```

Logical Cohesion

- ✧ Interface can be difficult to understand
- ✧ Code for more than one action may be intertwined
- ✧ Difficult to reuse

Temporal Cohesion

✧ Module performs series of actions related in time

Temporal Cohesion

```
3   void temporalCohesion(int d1,int d2)
4   {
5       //Create a database
6       Database newDatabase = new Database(d1);
7
8       //Create a list
9       List exampleList = new List();
10
11      //Create a new weather table
12      Weather exampleWeather = new Weather(d2);
13  }
```

Temporal Cohesion

- ✧ Actions weakly related to one another
- ✧ Code can become spread out
  - Degrades maintainability and reusability

Procedural Cohesion

- ✧ Module performs series of actions related by procedures to be followed by the product

Procedural Cohesion

```
 5   public class FogDoor : Door
 6   {
 7       public GameObject fog;
 8
 9       override public void open()
10       {
11           //Remove associated fog
12           Debug.Log("Remove Fog");
13           fog.SetActive(false);
14           //Open the Door
15           this.gameObject.SetActive(false);
16       }
17   }
```

Procedural Cohesion

- ✧ Actions are weakly related
- ✧ Not reusable

Communicational Cohesion

- Module performs series of actions related by procedure to be followed by the product, but in addition all the actions operate on the same data

Communicational Cohesion

```
16      void OnTriggerStay2D(Collider2D other)
17      {
18          triggered=1;
19          Vector2 position = new Vector2(0f,0f);
20          PlayerClass player = PlayerClass.Instance;
21          if(other.name != "Player")
22          {
23              return;
24          }
25
26          //Move Player to the start of the level
27          if(scene == 1)
28          {
29              position.x = -9f;
30              position.y = 3.75f;
31              player.setPlayerPos(position);
32          }
33        else if(scene == 2)
34          {
35              position.x = 9.125f;
36           position.y = -32.5f;
37           player.setPlayerPos(position);
38          }
39          //Damage the player
40          player.updateHealth(damage);
41      }
```

Communicational Cohesion

✧ Harms reusability

Informational Cohesion (Best)

- ✧ Module performs a number of actions, each with its own entry point, with independent code for each actions, all performed on the same data structure

Informational Cohesion (Best)

```
6   public class PlayerClass : MonoBehaviour
7   {
8       public static PlayerClass Instance { get; private set; }
9
10      [SerializeField] float moveSpeed;
11      protected int health;
12      bool BCMode;
13      Rigidbody2D rgdb;
14      Vector2 newPos;
15      bool interacting;
16      bool frozen;
17      bool gameOver;
18      PlayerInventory inventory;
19      bool compSet;
20      int updateNum;
21
22      private void Awake()
23      {
24          // Ensure that only one instance of the player can exist
25          if (Instance != null && Instance != this)
26          {
27              Destroy(this.gameObject);
28          }
29          else
30          {
31              // Create player and Keep player between scenes
32              Instance = this;
33              DontDestroyOnLoad(this);
34          }
35      }
```

Informational Cohesion (Best)

```
37      // Start is called before the first frame update
38      void Start()
39      {
40          // Initialize player
41          this.health = 100;
42          this.updateNum = 0;
43          this.BCMode = false;
44          this.gameOver = false;
45          this.frozen = false;
46          this.interacting = false;
47          this.compSet = false;
48          setComponents();
49      }
50
51      public void setComponents()
52      {
53          if (!compSet)
54          {
55              this.rgdb = this.GetComponent<Rigidbody2D>();
56              this.inventory = this.GetComponent<PlayerInventory>();
57              compSet = true;
58          }
59      }
```

```
67      void OnValidate()
68      {
69          if (moveSpeed > 15)
70          {
71              moveSpeed = 15;
72          }
73          else if (moveSpeed < 5)
74          {
75              moveSpeed = 5;
76          }
77      }
78
79      private void FixedUpdate()
80      {
81          // If the player is not currently interacting with something
82          if(!this.frozen)
83          {
84              // Get the players current position
85              this.newPos = new Vector2(this.transform.position.x, this.transform.position.y);
86
87              // If user is moving left or right, adjust the x position of the player
88              if (Input.GetAxisRaw("Horizontal") != 0)
89              {
90                  this.newPos.x += (this.moveSpeed * Input.GetAxisRaw("Horizontal") * Time.deltaTime);
91              }
92
93              // If user is moving up or down, adjust the y position of the player
94              if (Input.GetAxisRaw("Vertical") != 0)
95              {
96                  this.newPos.y += (this.moveSpeed * Input.GetAxisRaw("Vertical") * Time.deltaTime);
97              }
98
99              // Apply any position adjustments made to the player
100             this.rgdb.MovePosition(newPos);
101         }
102     }
```

Informational Cohesion

```
104     private void OnTriggerEnter2D(Collider2D other)
105     {
106         // Check if user is interacting with something interactable or an item
107         if (other.gameObject.tag == "interactable")
108         {
109             this.interacting = false;
110             return;
111         }
112         else if (other.gameObject.tag != "item")
113         {
114             return;
115         }
116 
117         pickupItem(other.gameObject.GetComponent<ItemClass>());
118 
119     }
120 
121     private void OnTriggerStay2D(Collider2D other)
122     {
123         if (other.gameObject.tag != "interactable")
124         {
125             return;
126         }
127 
128         // Interact with the object if the user it touchinig it an presses E
129         if (Input.GetKey(KeyCode.E) && !this.interacting)
130         {
131             // Create a temp object to utilize the Interactable interface, then interact with it
132             IInteractable interactedObj = other.gameObject.GetComponent<IInteractable>();
133             this.interacting = true;
134             interactedObj.interact();
135         }
136     }
```

```
138     private void OnTriggerExit2D(Collider2D other)
139     {
140         // Reset the player interaction
141         this.interacting = false;
142         this.updateNum = 0;
143     }
144 
145     private void pickupItem(ItemClass item)
146     {
147         addInvItem(item);
148         Debug.Log("Player has picked up a " + item.name + " item.");
149     }
150 
151     public bool isInteracting()
152     {
153         // Show whether use is interacting with something
154         return this.interacting;
155     }
156 
157     public void isInteracting(bool isInteracting)
158     {
159         // Freeze the player and update interacting variables
160         this.interacting = isInteracting;
161         this.frozen = isInteracting;
162     }
```

Informational Cohesion

```
164     public void updateHealth(int change)
165     {
166         if (++this.updateNum > 1)
167         {
168             return;
169         }
170         // Check if BC mode is active
171         if (BCMode)
172         {
173             // Create a temporary long variable to hold the health's value (to prevent underflow)
174             long rangeExp = (long)this.health + (long)change;
175             // If user's health will be over 100, set it to the max of 100
176             if ((rangeExp) > 100)
177             {
178                 this.health = 100;
179             }
180             // If the new health is a lower value than the variable can hold, set it to the min value
181             else if (rangeExp < Int32.MinValue)
182             {
183                 this.health = Int32.MinValue;
184             }
185             // If the health is within range, then adjust as normal
186             else
187             {
188                 this.health += change;
189             }
190             // Finish updating health
191             Debug.Log("Player health is now " + this.health);
192             return;
193         }
194
195         // If user's health will be over 100, set it to the max of 100
196         if ((health + change) > 100)
197         {
198             health = 100;
199         }
```

```
195         // If user's health will be over 100, set it to the max of 100
196         if ((health + change) > 100)
197         {
198             health = 100;
199         }
200         // If user's health will be less than 0, set it to 0 and trigger game over
201         else if ((this.health + change) <= 0)
202         {
203             this.health = 0;
204             this.gameOverAct();
205         }
206         // Adjust health as normal
207         else
208         {
209             this.health += change;
210         }
211         Debug.Log("Player health is now " + this.health);
212     }
213
214     public int getHealth()
215     {
216         return health;
217     }
218
219     public void setSpd(float newSpd)
220     {
221         moveSpeed = newSpd;
222     }
223
224     public float getSpd()
225     {
226         return moveSpeed;
227     }
228
```

Informational Cohesion

```
234     public bool startBCMode(string password)
235     {
236         // Check whether password is correct
237         if (password.Equals("GoBig", StringComparison.Ordinal))
238         {
239             // If correct passowrd, activate BC mode
240             this.BCMode = true;
241         }
242 
243         return this.BCMode;
244     }
245 
246     public void setPlayerPos(Vector2 pos)
247     {
248         // Set the player's position
249         this.transform.position = new Vector3(pos.x, pos.y, 0);
250     }
251 
252     public bool addInvItem(ItemClass addedItem)
253     {
254         return this.inventory.addItem(addedItem);
255     }
256 
```

```
256 
257     public bool removeInvItem(int invIndex)
258     {
259         return this.inventory.removeItem(invIndex);
260     }
261 
262     public int getNumInvItems()
263     {
264         return this.inventory.getNumItems();
265     }
266 
267     public int getMaxItems()
268     {
269         return this.inventory.getMaxItems();
270     }
271 
272     public ItemClass getInvItem(int index)
273     {
274         return inventory.getItem(index);
275     }
276 
277     public void switchInvItems(int InvitemOne, int InvitemTwo)
278     {
279         inventory.switchItems(InvitemOne, InvitemTwo);
280     }
281 }
```

Informational Cohesion (Best)

✧ Data structure access and modification all in one place

Functional Cohesion

✧ Module performs exactly one action

Functional Cohesion

```
 7        override public void open()
 8        {
 9            Debug.Log("Open the door");
10            //Destroy(this);
11            this.gameObject.SetActive(false);
12        }
```

Functional Cohesion (Best)

- ✧ Reusable
- ✧ Easier maintenance
  - Fault isolation

6. GRASP Polymorphism

✧ Problem

- How to handle alternatives based on type?
- How to create pluggable software components?

✧ Solution

- When related objects vary by type, assign responsibility for the behavior to the types for which the behavior varies.

6. GRASP Polymorphism

- ✧ Benefits
  - Higher utility of functions
  - Easier understanding
- ✧ Examples
  - Override/Virtual Functions
  - Function Overloading
  - Variable Coercion

7. GRASP Pure Fabrication

- ✧ Problem
  - How to achieve low coupling, high cohesion easily in a system when other concepts fail?
- ✧ Solution
  - Create a class that does not represent a specific problem to achieve low coupling, high cohesion and future reuse.
    - Sometimes called a service class

7. GRASP Pure Fabrication

- ✧ Benefits
  - Enables low coupling/high cohesion when information expert fails
  - Reusability
- ✧ Examples
  - Mediator pattern

8. GRASP Indirection

✧ Problem

- Where do we assign responsibility to avoid direct coupling between two or more modules?
- How to de-couple objects so that low coupling is supported?

✧ Solution

- Create an object to mediate between two or more objects to avoid direct coupling.
  - Instead of direct connection, two modules share a mediating module
  - Helps maintain reusability via low coupling

8. GRASP Indirection

- ✧ Benefits
  - Reusability (via low coupling)
  - Flexibility
  - Security
- ✧ Examples
  - Messenger Module
  - Password Checker

9. GRASP Protected Versions

- ✧ Problem
  - How to design objects and subsystems so that their failure or instability does not influence the rest of the system?
- ✧ Solution
  - Identify points of instability and assign responsibilities to create a stable interface around them.
    - Test plans!

9. GRASP Protected Versions

- ✧ Benefits
  - Hide the instability from the rest of the system
  - Avoid cascading errors
  - Avoid unwanted outputs
- ✧ Examples
  - A state machine pattern with an error state
  - Default case statement

Summary

- ✧ High Cohesion and Low Coupling are the underlying theme of all patterns.
  - The Creator pattern supports low coupling by choosing a creator that already needs to be connected to created object.
  - Information Expert assigns responsibilities to the objects that have the information needed to complete the task.
    - Cohesion is maintained by delegating.
  - Sometimes patterns will give conflicting answers.
    - This is not an exact science but be able to justify your choices.

Summary

- ✧ No coupling is also undesirable
  - A central metaphor of object technology is a system of connected objects that communicate via messages.
  - Low Coupling is taken to excess leads to a few incohesive, bloated, and complex active objects that do all the work, with many very passive zero-coupled objects that act as simple data repositories.

Summary

- ✧ Libraries are Highly Coupled. This is permitted because:
  - these are stable
  - they are widespread/proven
- ✧ A subclass is Highly coupled to its superclass.
  - Have a good reason for sub classing between modules/team members.
  - If you have a superclass for another team member, ensure that it takes on the properties of a library.
    - Well documented
    - Infrequent changes
    - You will be held to a higher standard!

Summary

- ✧ Have a single controller that accepts messages (or break it into several based-on message type if it becomes too bloated.)
- ✧ Your GUI is NOT your controller.
  - Do not tie your business logic with your user interface because this limits future changes.