Team Lead 4 Presentation

Nyah, Matias, Marissa

Overview

1. GRASP
2. Modules
3. Coupling
4. Cohesion
5. Visibility
6. Coding Standards
7. Gantt Chart

Note:
you will need to understand coupling and cohesion for the Post-mortem presentation

GRASP

- Once you make a class and add methods, there are consequences to where you place methods and the way objects interact.
- **GRASP** is a software tool of design patterns to solve organizational problems by outlining what object or class is **responsible** for what.
- GRASP stands for:
  - **G**eneral
  - **R**esponsibility
  - **A**ssignment
  - **S**oftware
  - **P**atterns

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 fix it
- Especially important for novel (unique and unexpected) situations

GRASP defines nine basic object oriented patterns

1. Creator
2. Information expert
3. Low coupling
4. Controller
5. High cohesion
6. Polymorphism
7. Pure fabrication
8. Indirection
9. Protected variations

Creator

- who is responsible for creating a new instance of some class?
  - Object creation is one of the most common OO activities
  - “Creator” principle is to achieve **low coupling** and increased clarity, encapsulation, and reusability.

Information Expert

- who is responsible for the general principle of assigning responsibilities to objects?
  - Assign a responsibility to the information expert, and this expert is the class that has the information needed to fulfill the responsibility
  - Determines where to delegate responsibilities such as methods, computed fields, and so on
    - Delegation helps with cohesion

Controller

- what object beyond the UI layer receives and coordinates (controls) a system operation?
  - Should delegate the work that needs to be done to other objects
    - it coordinates or controls the activity.
  - Controller should not do much work itself

Coupling vs Cohesion Definition

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

**we want high cohesion and low coupling in our programs
High Coupling and Low Cohesion are the underlying theme of all patterns.

Modules

- In order to really understand coupling and cohesion you need to understand modules
- What is a module?

Module - Lexically contiguous sequence of program statements bounded by boundary elements with an aggregate identifier.

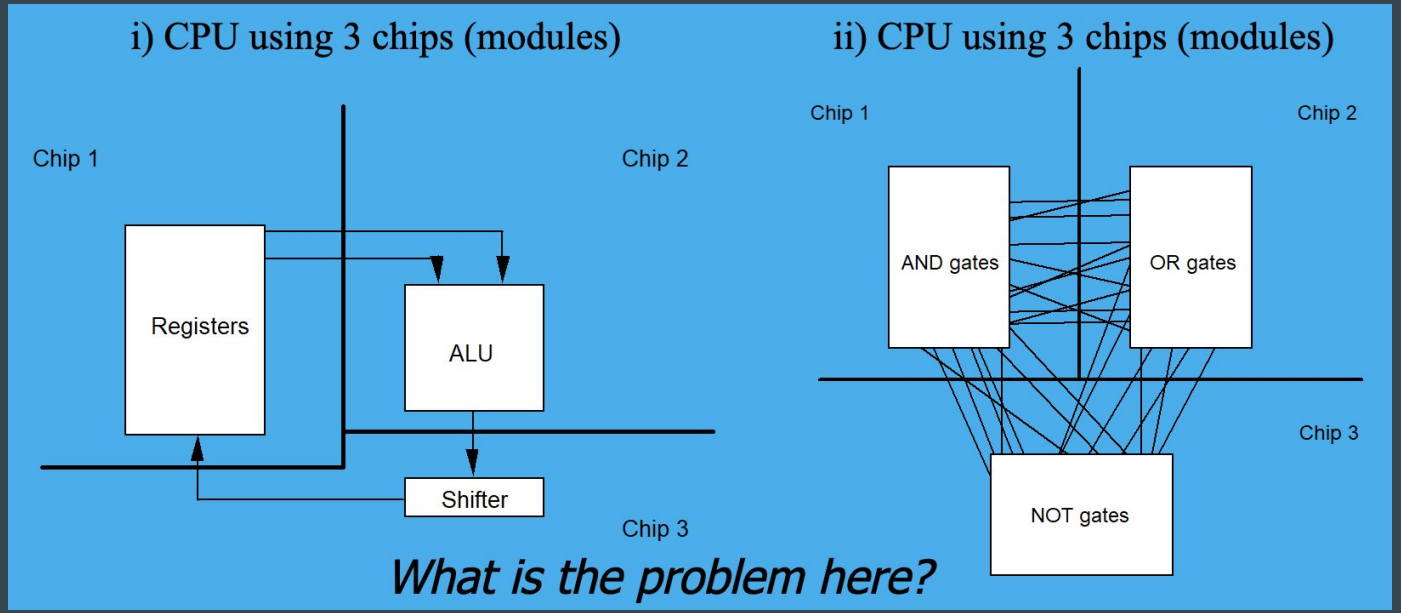
- “Lexically contiguous”
  - Connected in the code (not just related by control flow or data)
- “Boundary elements”
  - { ... }
  - Curly brackets
- “Aggregate identifier”
  - A name for the entire module

Examples of Modules

- Procedures and functions in classical program languages
- **Objects** and **methods** within objects in object oriented programming languages

Good vs Bad Modules

- Modules in themselves are not “good”
- You must design a module to have good properties



The first design is better

- The two designs function equivalently, but second design is
  - Hard to understand
  - Hard to locate faults
  - Difficult to extend or enhance
  - Cannot be reused
  - Expensive to perform maintenance
- Good modules must be like the first design:
  - Maximal relationships within the modules (**cohesion**)
  - Minimal relationships between the modules (**coupling**)
  - **this is the main contribution of structured design

Don’t Over Do It

- Don’t stifle creativity
  - Make informed, deliberate decisions, on when and where to apply modularization.
  - Ask yourself: how will this impact the development environment?
- Modules that are too small
  - One function modules lead to unnecessary overhead
- Refactoring for no reason
  - Only refactor and modularize the code that is likely to change and needs to be understandable
  - Some code almost never changes, so it is not worth refactoring

Cohesion - degree of action **within** module

Seven levels of cohesion

1. Coincidental (worst)
2. Logical
3. Temporal
4. Procedural
5. Communicational

7. Functional | Informational (best)

1. Coincidental Cohesion (worst)

- Def: module performs multiple, completely **unrelated actions**
- Example:
  - a module that prints next line, reverses the characters of the 2nd argument, and adds 7 to 3rd argument
- How could this happen?
  - Hard organizational rules about module size
- Why is this bad?
  - Degrades maintainability and modules aren't reusable
- Easy to fix. How?
  - Break into separate modules each performing one task

2. Logical Cohesion

- Def: module performs series of related actions, **one of which is selected by calling module**
- Example:
  - function code = 7;
  - new operation(op code, dummy 1, dummy 2, dummy 3);
  - //dummy 1, dummy 2, and dummy 3 are dummy variables
  - //not used if function code is equal to 7
- Why is this bad?
  - Interface difficult to understand
  - Code for more than one action may be intertwined
  - Difficult to reuse

3. Temporal Cohesion

- Def: module performs series of actions **related in time**
- Initialization example:
  - open old db, new db, transaction db, print db, initialize sales district table, read first transaction record, read first old db record
- Why is this bad?
  - Actions weakly related to one another, but strongly related to actions in other modules
  - Code spread out -> not maintainable or reusable
- Initialization example fix
  - Define these initializers in the proper modules and then have an initialization module call each

4. Procedural Cohesion

- Def: module performs series of actions **related by procedure to be followed by product**
- Example:
  - update part number and update repair record in master db
- Why is this bad?
  - Actions are still weakly related to one another
  - Not reusable
- Solution?
  - Break it up!

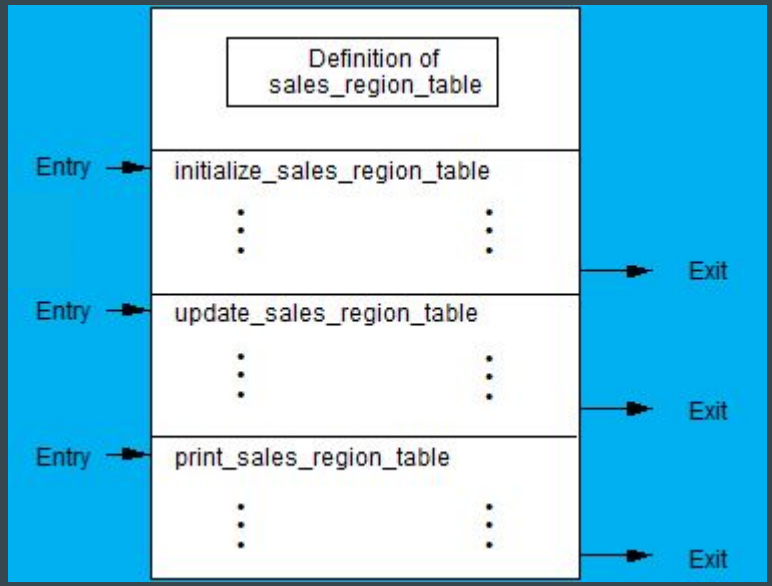
5. Communicational Cohesion

- Def: module performs series of actions related by procedure to be followed by product, but **in addition all the actions operate on same data**
- Example 1:
  - update record in db and write *it* to audit trail
- Example 2:
  - calculate new coordinates and send *them* to window
- Why is this bad?
  - Still leads to less reusability -> break it up

7. Informational Cohesion (best)

- Def: module performs a number of actions, **each with its own entry point**, with **independent code** for each action, all performed on the **same data structure**

This in an ADT



7. Functional Cohesion (best)

- Def: module performs **exactly one action**
- Examples:
  - get temperature of furnace
  - compute orbital of electron
  - calculate sales commission
- Why is this good?
  - More reusable
  - Corrective maintenance easier
    - Fault isolation
    - Reduced regression faults
  - Easier to extend product

Coupling - degree of interaction **between** two modules

- Kinds of Coupling
  - Class X has an attribute referring to class Y instance
  - Class X objects call on the services of a class Y object
  - Class X has methods referencing instances (parameters, local variables) of class Y
  - Class X is a direct or indirect subclass of class Y
  - Y is an interface and class X implements it
- note: all of these are needed at some point, the point is not to eliminate coupling, but to make careful choices

Five levels of coupling

1. Data (best)
2. Stamp
3. Control
4. Common
5. Content (worst)

5. Content Coupling (worst)

- One module **directly** references contents of the other
- Example:
  - module a modifies statements of module b
  - Module a refers to local data of module b in terms of numerical displacement within b
  - Module a branches into local label of module b
- Why is this bad?
  - Almost any change to b requires changes to a

4. Common Coupling

- Two modules have *write* **access to the same global data**
- Example:
  - two modules have access to the same database, and can both read and write the same record
- Why is this bad?
  - Resulting code is unreadable
    - modules can have side-effects
    - must read entire module to understand
  - difficult to reuse
  - module exposed to more data than necessary

3. Control Coupling

- One module passes an element that is intended to control the internal logic of the other modules
- Example:
  - control-switch passed as an argument
- Why is this bad?
  - Modules are not independent
    - module b must know the internal structure of module a
    - effects reusability

Control Coupling Example

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

void bar(){
    // Calling foo() by passing a value which controls its flow:
    foo(1);
}
```

2. Stamp Coupling

- Data structure is passed as parameter, but called **module operates on only some of individual components** (of the structure)
- Example
  - Calculate withholding (employee record)
- Why is this bad?
  - Affects understanding
    - **Not clear without reading the entire module which fields of record are accessed or changed**
  - Unlikely to be reused
    - Other products have to use the same higher level data structures
  - **Passes more data than necessary**
    - Ex. uncontrolled data access can lead to computer crime

Stamp Coupling Example

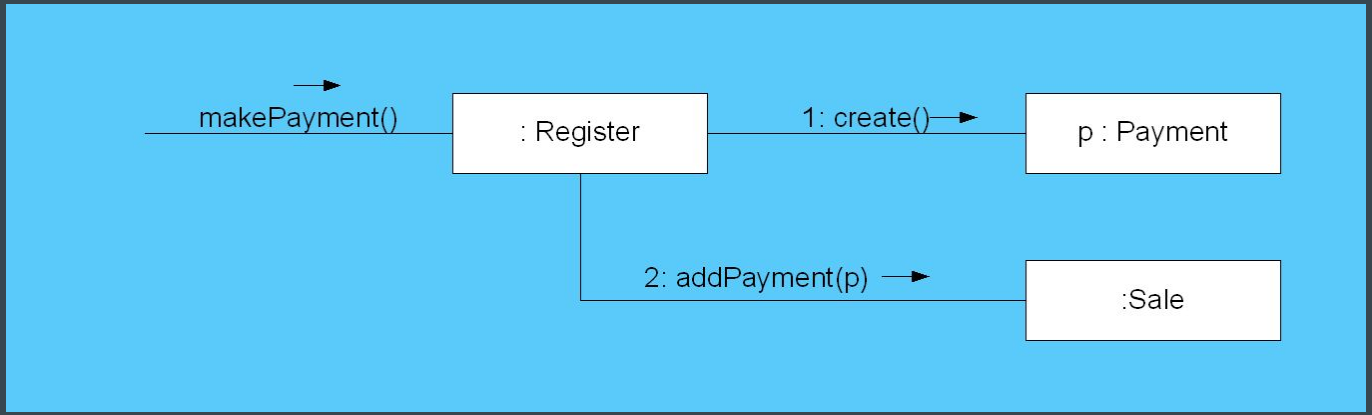
```
class A{
   // Code for class A.
};

class B{
    // Data member of class A type: Type-use coupling
    A var;

    // Argument of type A: Stamp coupling
    void calculate(A data){
        // Do something.
    }
};
```

1. Data Coupling (best)

- Every argument is either a simple argument or a data structure in which **all elements are used** by the called module
- Example:
  - display time of arrival (flight number)
  - Get job with highest priority (job queue)
- Why is this good?
  - Maintenance is easier
  - Good design has high cohesion and weak coupling

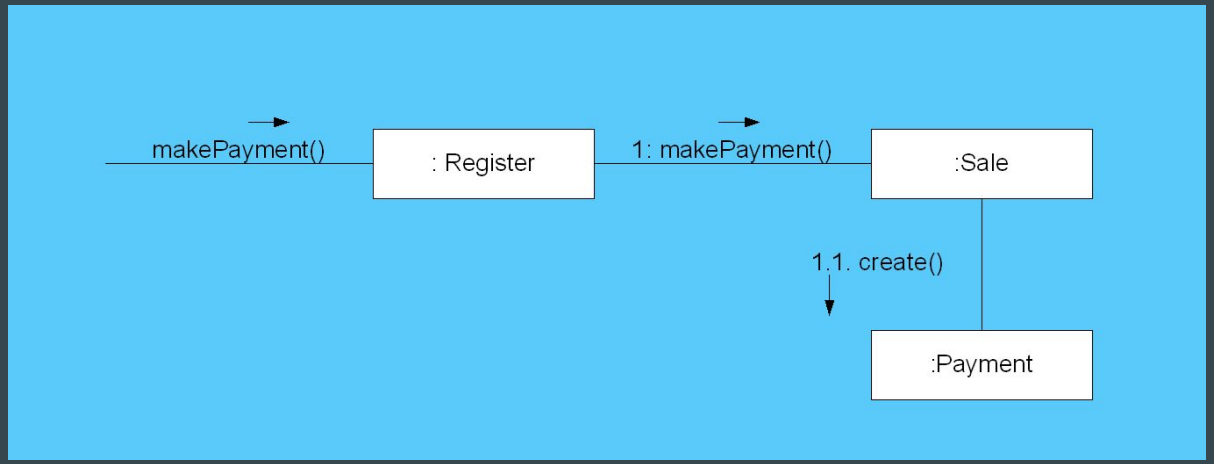
Coupling Example

- Suppose we apply our two previous GRASP principles like so:
  - Creator - suggests Register as a candidate for creating Payments
  - Information Expert - Register knows about Payments and therefore associates these with Sales
- This assignment of responsibilities couples the Register class to knowledge of the Payment class and send Payment as a parameter to the Sale class.

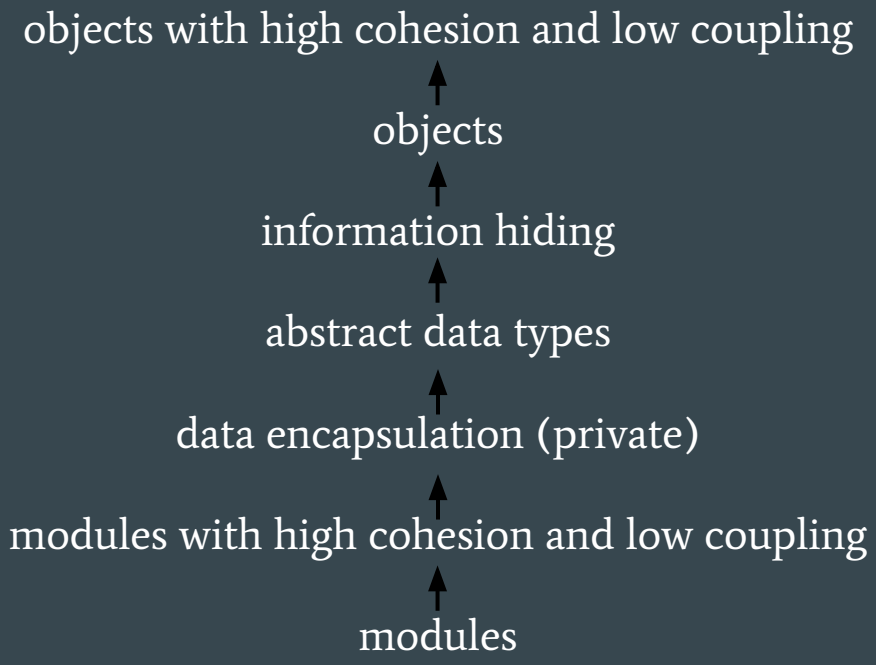


Coupling Solution

- Another approach: Creator - Suggests Sales creates Payments
- We will assume the Sale must eventually be coupled to knowledge of a Payment
- Instead of sending Payment as a parameter to Sale, Sale does the creation of Payment
- Lower coupling is maintained



Modules to Objects



Information Hiding

- Technically “details hiding”
  - We are hiding implementation details, not information
- Information will create loose coupling
- Example
  - Design modules so that items likely to change are hidden (declare them as **private**)
    - Future change is localized
    - Change cannot affect other modules
  - Data abstraction
    - Provide only essential details, hide background details

Visibility

- Visibility is the ability for objects to “see” each other
- An object can only interact with another object if it is has reference to it
- Visibility is usually a function of scope, where one object is within the scope of another
- There are four types of Visibility:
  - Attribute Visibility
  - Parameter Visibility
  - Local Visibility
  - Global Visibility

Attribute Visibility

- Declare one object as a member variable in the other
- Relatively permanent, as it exists as long as both objects exist
- Most common

Parameter Visibility

- One object is passed to a method of another object
- Relatively temporary, as the link only exists within the scope of the method
- You can transform Parameter visibility into Attribute visibility

Local Visibility

- Declare an object as a local variable within the method of another object
- Two ways to achieve this:
  - Create a new local instance assigned to a local variable
  - Assign the returning object from a method invocation to a local variable

Global Visibility

- Make one object global to another object
- Relatively permanent, but also the least common of the four
- Two technique to achieve this:
  - Ordinary global variables
  - Singleton Pattern

On Tags

- In Unity, Tags are another way to achieve visibility
- Tags can be applied to any object in Unity
- To apply a tag, click on the dropdown menu near the top of the inspector, and you can either choose one of the pre-made tags, or create your own!
- Using the below function, you can make your object visible to any of your classes, and in any of the previously discussed ways
    - GameObject.FindWithTag(“*your tag here*”)

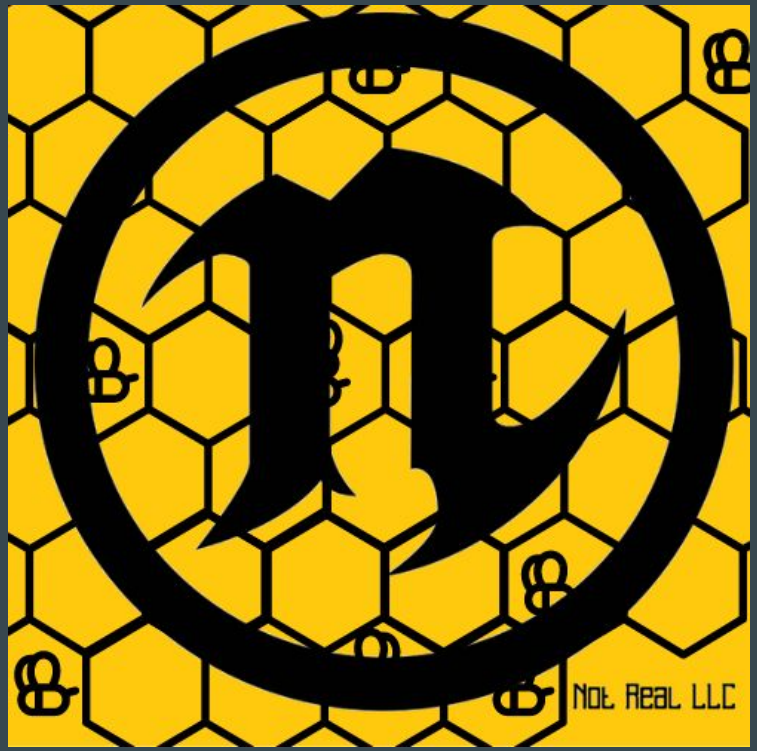
Coding Standards - guidelines for code style and documentation

- Document every time you violate a standard
- No standard is perfect for every application, but failure to comply with your standards requires a comment

Coding Standards

Gantt Chart and Coupling/Cohesion Roast!

notReal



Gantt Chart

Jacob (team lead 2)			
Setting up unity	2		complete
Setting up additional project it	2		complete
Building initial player framewo	4		complete
Smoothing out input	1		complete
Publicize needed variables and	1		complete
Hard code physics	8		complete
Set up animation controller	3		this week
Get sprites	10		this week
Set up state manager	8		planned
Graphics pass	6		planned
Advanced controls	4		planned
Apply level shaders	8		planned
Work on boss framework	6		planned
Add boss mechanics	5		planned
Finalize implementation	4		planned
totals	72	0	

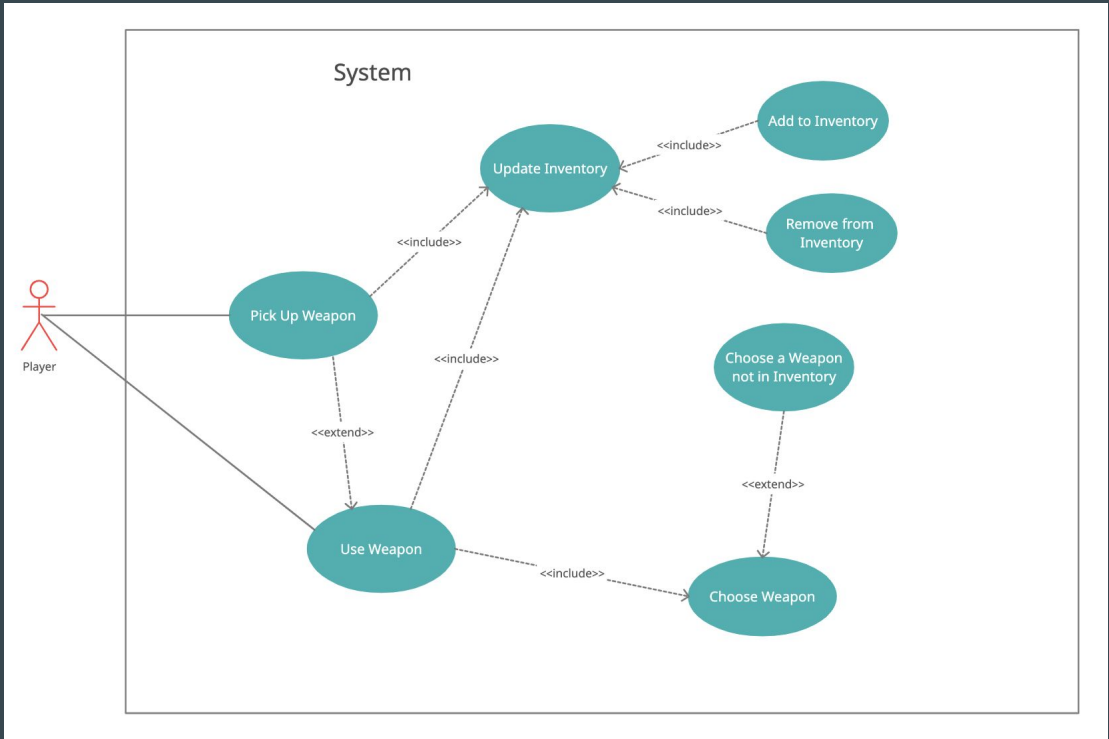
Spencer (team lead 2)		
0. Setting up unity	2	2
1. Basic attack framework	10	
2. Weapon registry datastore	3	
3. Basic weapons	1	
4. Test attack framework	2	
5. Item provider	4	
6. Test item provider	2	
7. Pickup handler	6	
8. Test pickups	1	
9. Advanced weapons	12	
10. Acquire graphics	6	
11. Integrate graphics	6	
12. Cohesive Playtesting	4	
totals	59	2

Jacob has done a lot of work, but hasn't added any hours. This means he won't get paid.

Spencer only has 2 hours, but we know he has done more work than that.

Nyah and Spencer: weapon pickup and update inventory

Too much coupling?



NaughtyCat

Gantt Chart

Ambrea		
Research Inventory	3	1
Create UIs	4	0
Make user interact w/inventory	6	0
Make item interact w/inventory	6	0
Test UI's	4	0
Finalize	6	0
totals	29	1

Tosin		
1. Acquire Sprites and Assets	2	0
2. Animation	20	0
3. Scripting	15	0
4. Testing	3	0
	0	0
	0	
	0	0
totals	40	0

Has been working on things, but haven't been updating the gantt chart regularly, important for getting paid

It's important that everyone works on their features regularly as to not get left behind or delay the project

Coupling or Cohesion example

So far there is low/no coupling that I can find. It is good to have coupling somewhat; we're trying to limit it, not eliminate it. But, there doesn't seem to be any coupling issues so far :)

Unity

Gantt Chart

Matias Crespo - Walls and Secrets			
ID	Feature Name	Predicted Time (hrs)	Time Spent (hrs)
0	Unity Set-Up	1	1
1	Research and ideation	5	1
2	Room Data Structure	5	
3	Randomization Procedure	10	
4	Wall Destruction	8	
5	Secret Room Design	12	
6	Room Effects	5	
7	Room Cutscenes	5	
8	Documentation	1	
9	Testing	3	
10	Integration	5	
11	Artwork	4	
Matias' Totals		64	2

Cole Halvorson - UI			
ID	Feature Name	Predicted Time (hrs)	Time Spent (hrs)
0	Unity Set-Up	1	2
1	Main Menu	3	
2	HUD	6	
3	Pause Menu	6	
4	Shop Menu	9	
5	Death Screen	3	
6	Win Screen	3	
7	Help Screen	3	
8	Settings Menu	6	
9	Refine Menus	12	
10	Implement Feature	6	
Cole's Totals		58	2

Looks like some people are falling behind! (on updating their work hours)

Coupling or Cohesion example

Triston, Cole, and Matias are all working on the Player Shop?

Sounds like too much coupling!

Summary

- GRASP assigns responsibility to objects
- High cohesion, low coupling
- If you can’t see an object, you can’t talk to it
- Update Gantt Charts

Thank you!

Questions?