Iteration 3

- Peer Response – please let me know ASAP if you still do not have something to review.

- **Assessment**
  - 30% Fully functional requirements of iteration 2*
  - 20% Refactoring (3 branches)
  - 05% No Warnings During compilation
  - 05% Submission, File Structure, File Names
  - 20% Peer Response Worksheet
  - 10% Doxygen
  - 05% cpplint Error Free
  - 05% Bug Report

*Iteration2 grade weighted heavily towards interfaces. Iteration3 grade weighted heavily towards implementation (still getting partial credit for dysfunctional code).

Warnings: Unused Variable

- Fix with ..

```
// in common.h
#define __unused __attribute__((unused))
```

Or delete it!

Why do you have it if it is unused? Can be legit, but know why it is there.

Warnings: X initialized after Y

- Fix by matching order of initialization with order of declaration ...

```
class ClassA {     // WARNING
private:
  int a_;
  int b_;
public:
  ClassA(int a, int b) : b_(b), a_(a) {}
}

class ClassB : public ClassA {       // WARNING
private:
  int c_;
public:
  ClassB(int a, int b, int c) : c_(c), ClassA(a,b) {}
}
```

Warnings: Copy and = not defined

- Fix by creating or deleting ...

- DEFINED

```
GraphicsArenaViewer& operator=(const GraphicsArenaViewer& other) {
  ... here is some code for assignment operator overload; }
GraphicsArenaViewer(const GraphicsArenaViewer& other) {
 ... here is some code for copy constructor; }
```

- DEFINED as empty

```
GraphicsArenaViewer& operator=(const GraphicsArenaViewer& other);
GraphicsArenaViewer(const GraphicsArenaViewer& other);
```

- DELETED

```
GraphicsArenaViewer&  operator=(const  GraphicsArenaViewer&  other) = delete;
GraphicsArenaViewer(const  GraphicsArenaViewer&  other) = delete;
```

Copy Constructor

```
class ClassA {
      private:
      Robot* r_;
public:
      ClassA(Robot* r) : r_(r) {}
}
class ClassB {
   Robot *robot = new Robot;
   ClassA objA(robot);
}
ClassB objB;       // robot = 0xA2
ClassB copy_B(objB)
// copy_B.r_ = 0xA2
// copy_B.r_ = new Robot whose members are a copy of robot at 0xA2
// copy_B.r+ = new Robot whose members are set to initial values

// What happens when objB is destroyed ??
```

Warnings: Casting

```
int helper_fun(int a) {
  return a; }

int main() {
  int a = 5.682;
  double b = 8.23;
  std::cout << a << " " << b << std::endl;
  a = helper_fun(a);
  b = helper_fun(b);
  std::cout << a << " " << b << std::endl;  return 1;}
```

warnings.cc:12:11: warning: implicit conversion from 'double' to 'int' changes value from 5.682 to 5

```
  int a = 5.682;
```

Iteration 3

- Checklist
  - Iteration3Checklist.md in class repo.
  - Print and check (you can pdf converter), or place ‘x’ in brackets (e.g. [x]) and it will display with checkmark.

FINAL EXAM

- C++ Syntax
  - Pointers and References
    - Declaring, defining, passing
  - Const
    - Const variables
    - Const pointers
    - Const functions
    - Const parameters
  - Class Constructors
  - Polymophism and Abstract Classes
    - Virtual
    - pointers

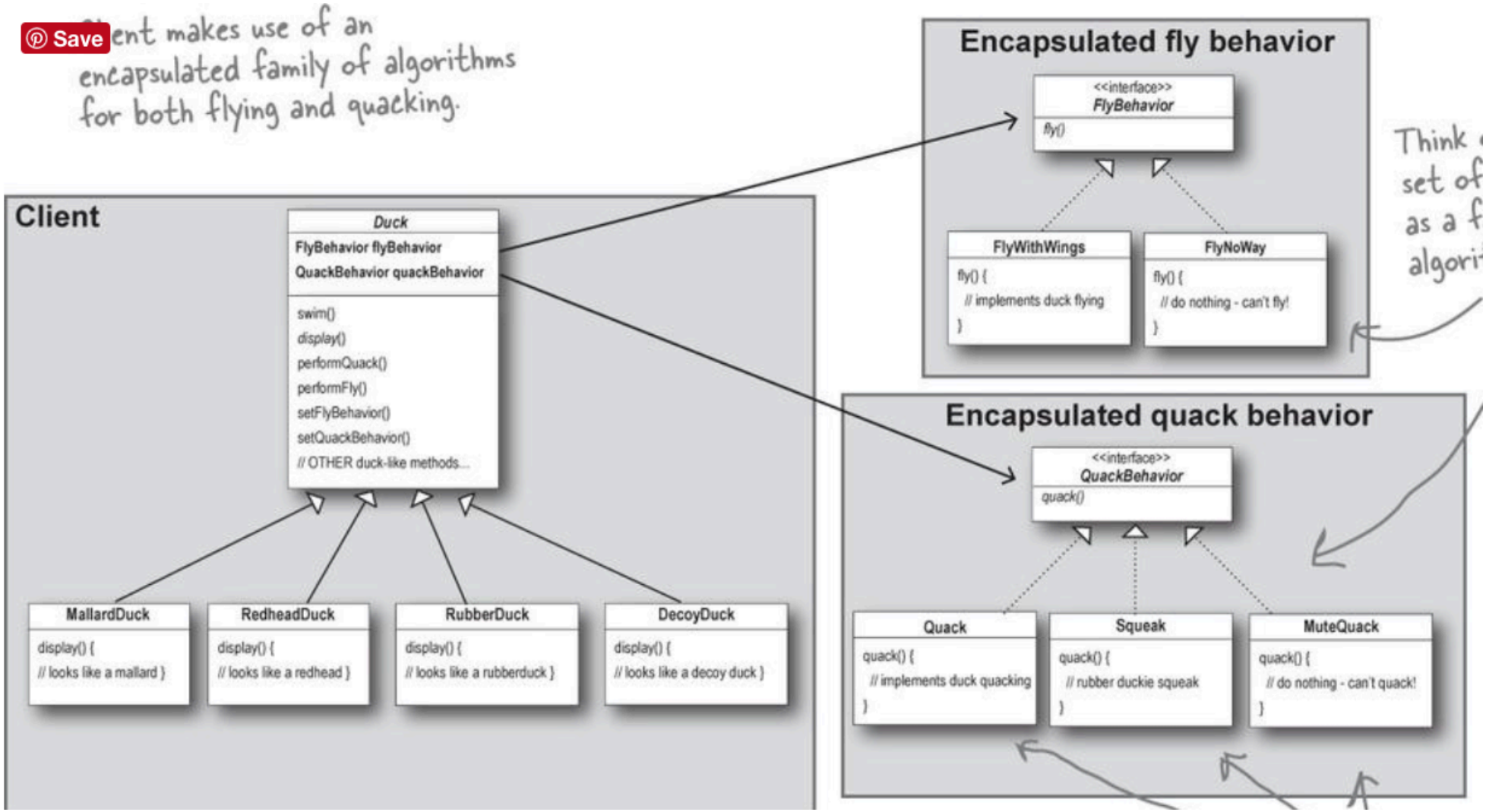
FINAL EXAM

- C++ Syntax cont ...
  - Operator Overload
    - Unary/Binary
    - Class member, friend
  - Copy Constructor
    - Define
    - Uses
  - Templates
    - Define
    - Use

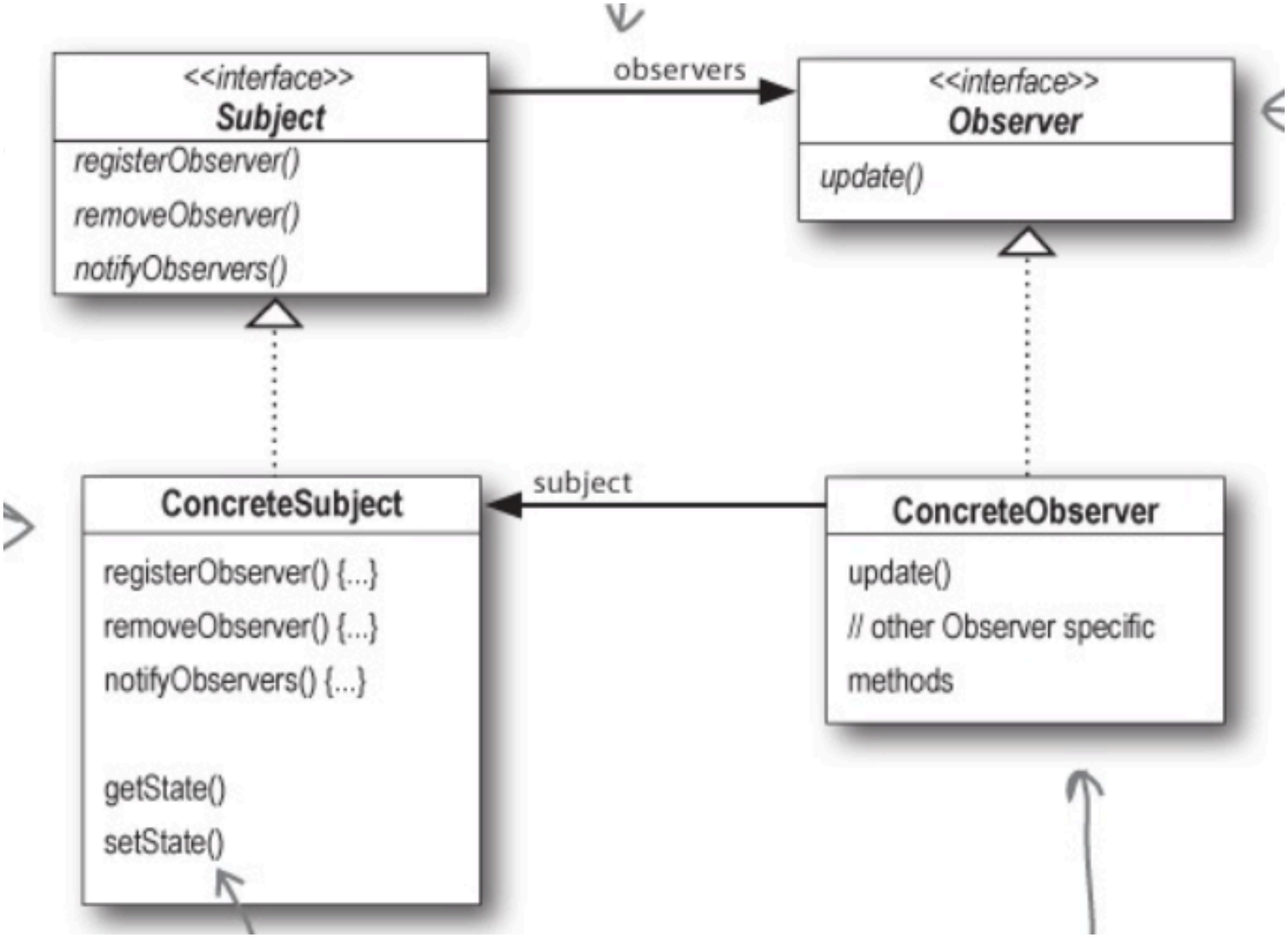
FINAL EXAM

- Design Patterns
  - HAS-A and IS-A : favor composition over inheritance
  - Strategy
    - Use composition and polymorphism to distinguish behavior.
  - Visitor
    - Allow other classes to modify private data without knowing the details.
  - Observer
    - Registered observers of a subject
  - Factory
    - Contain initialization in a separate class

Strategy


Save
ent makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think
set of
as a f
algori
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

Observer Pattern



Factory Design Pattern

```
class Entity; class Robot; class Player;

class EntityFactory {
public:
    Entity* Create(    ...

class RobotFactory : public EntityFactory {  ...

class PlayerFactory : public EntityFactory { ...

class Arena {
    void AddEntities( EntityFactory factory, ...
```