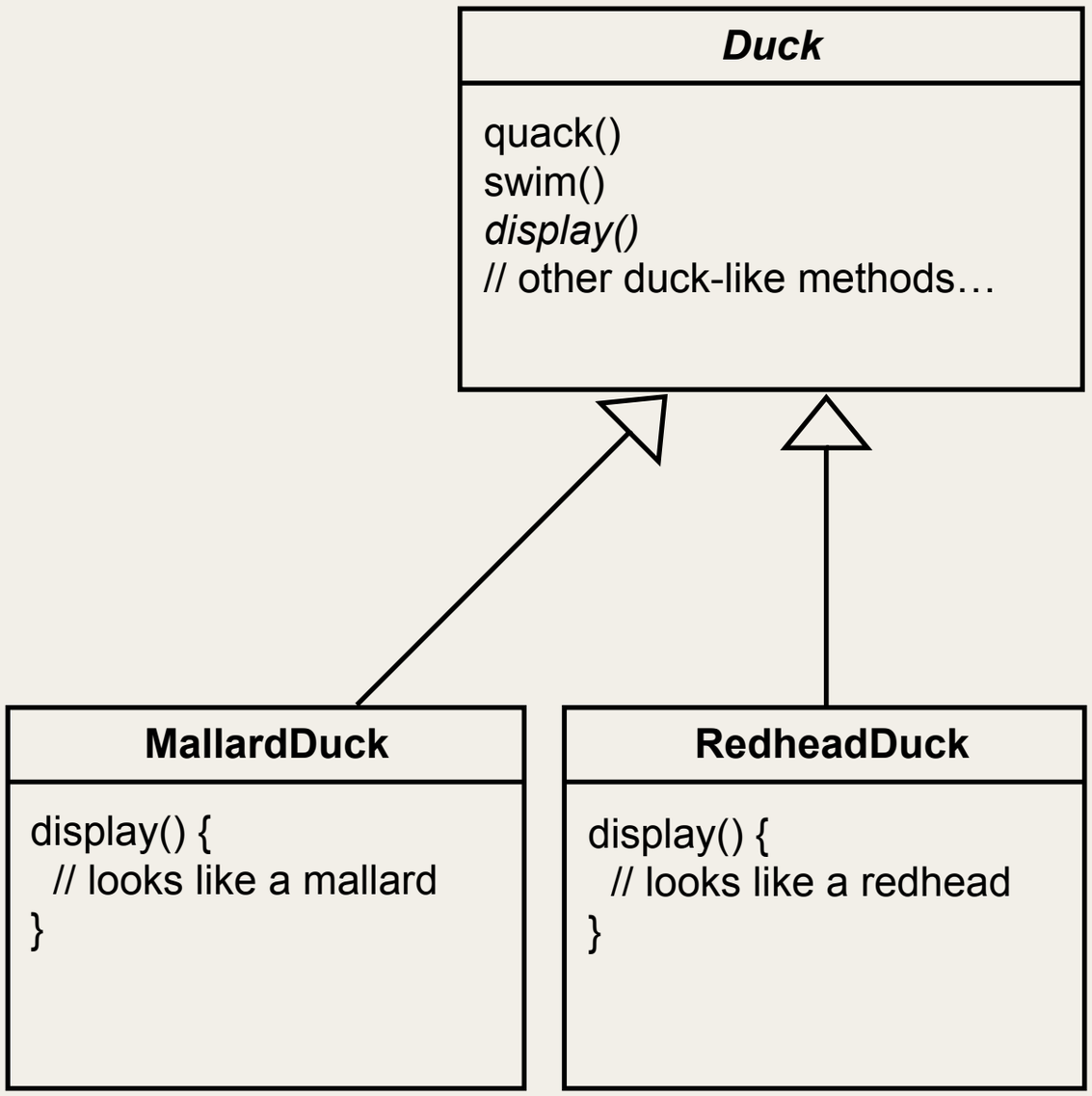
Duck Behaviors Continued / Intro to Makefiles

CSCI-3081: Program Design and Development

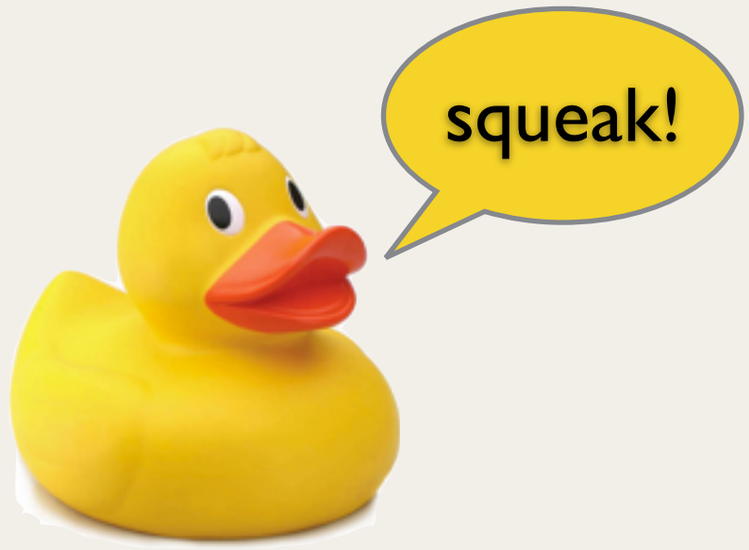
Duck example adapted from Head First Design Patterns by Freeman and Robson

Your First Challenge (this one is easy)

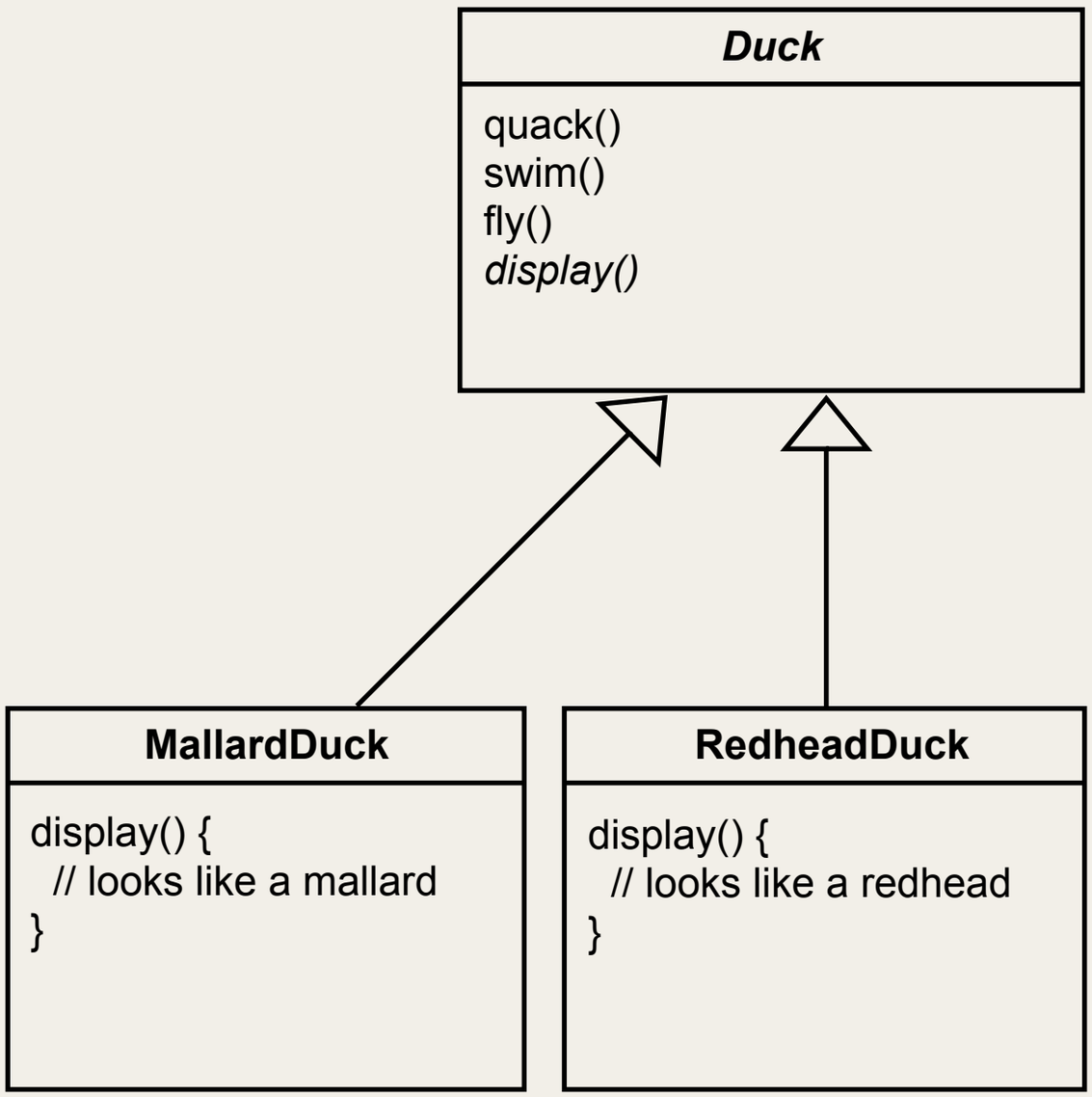
- Extend the program to make Ducks fly



Your Second Challenge

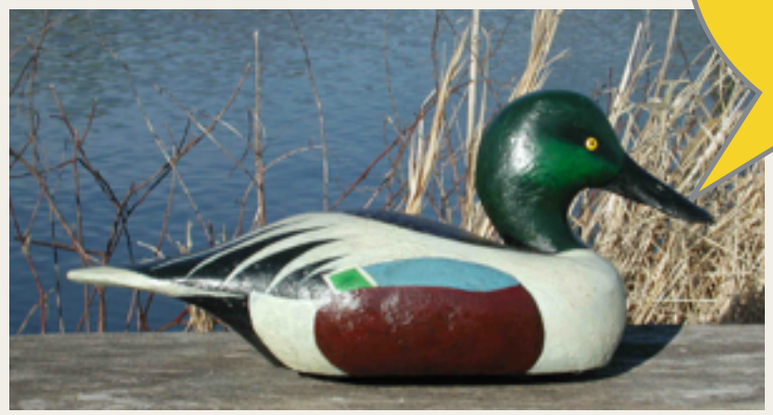


- Add a new type of duck - RubberDuck



Your Third Challenge

- Add a new type of duck - DecoyDuck



What is the one constant in software development?

CHANGE

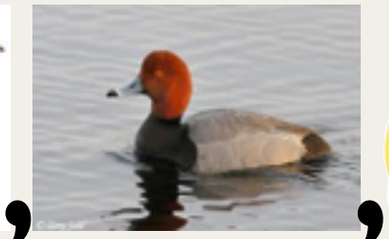
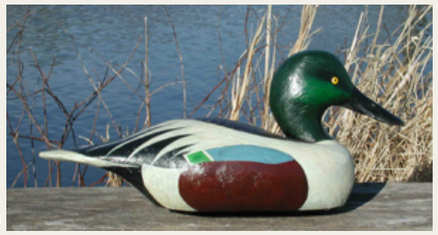
UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

So, what do we do about change?

- Identify the aspects of the application that change and separate them from what stays the same.
- In other words, take what varies and “encapsulate” it so it won’t affect the rest of your code.

How does this work in our example?

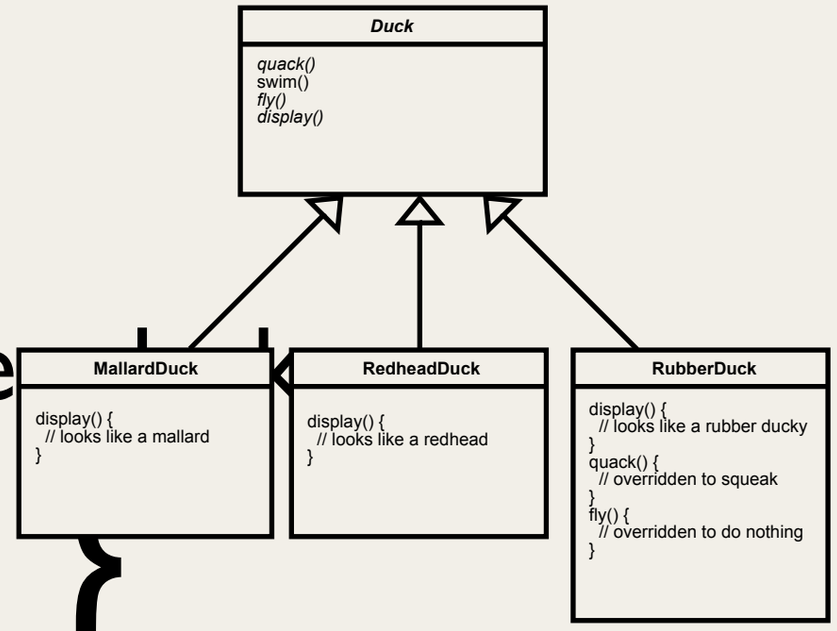
First, let's identify what changes between ducks:

{  ,  ,  ,  , . . . }

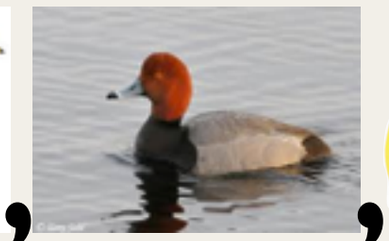
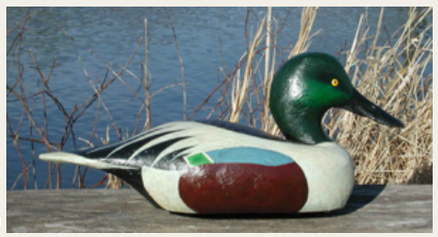
quack() - could be a quack, a squeak, or nothing

fly() - could fly or not

How does this work in our example?



First, let's identify what changes betwee

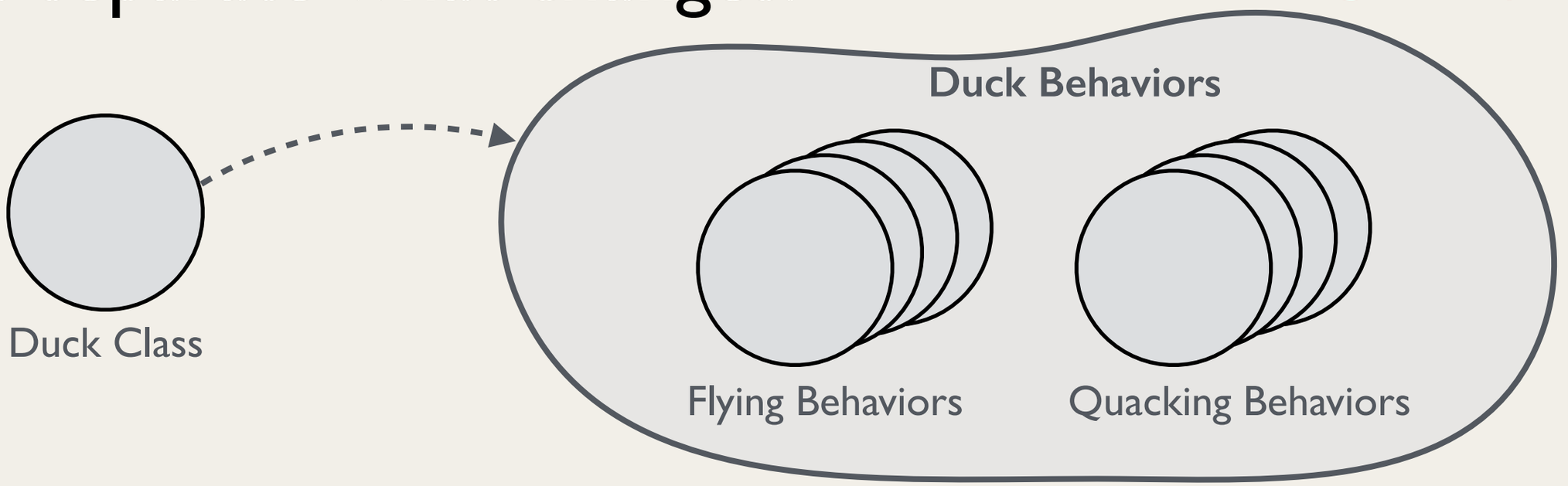
{  ,  ,  ,  , . . . }

quack() - could be a quack, a squeak, or nothing

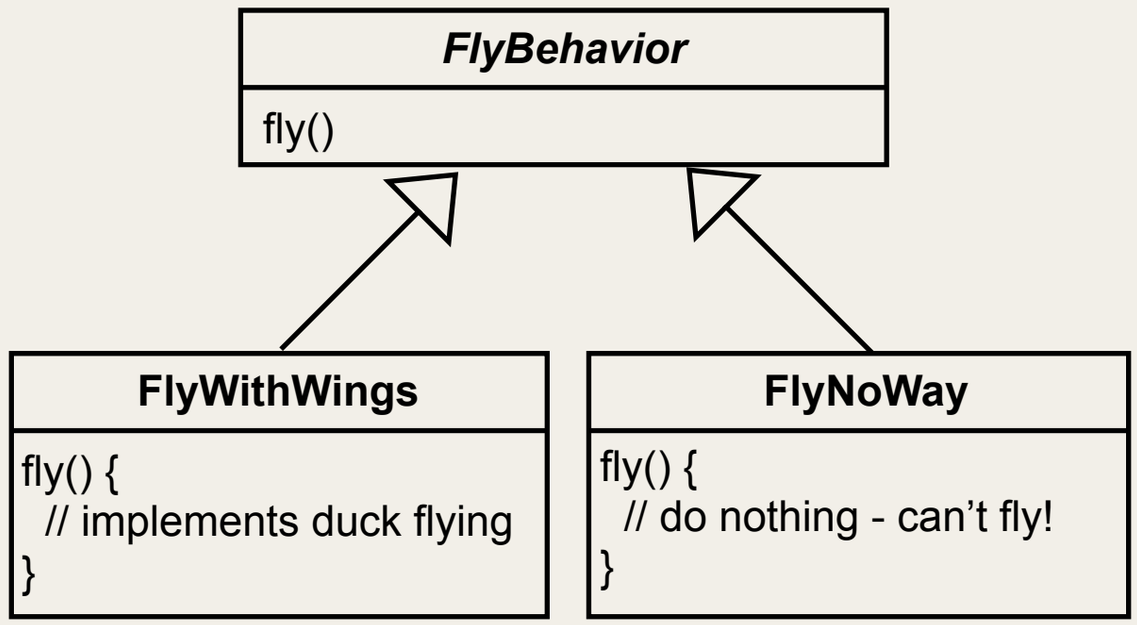
fly() - could fly or not

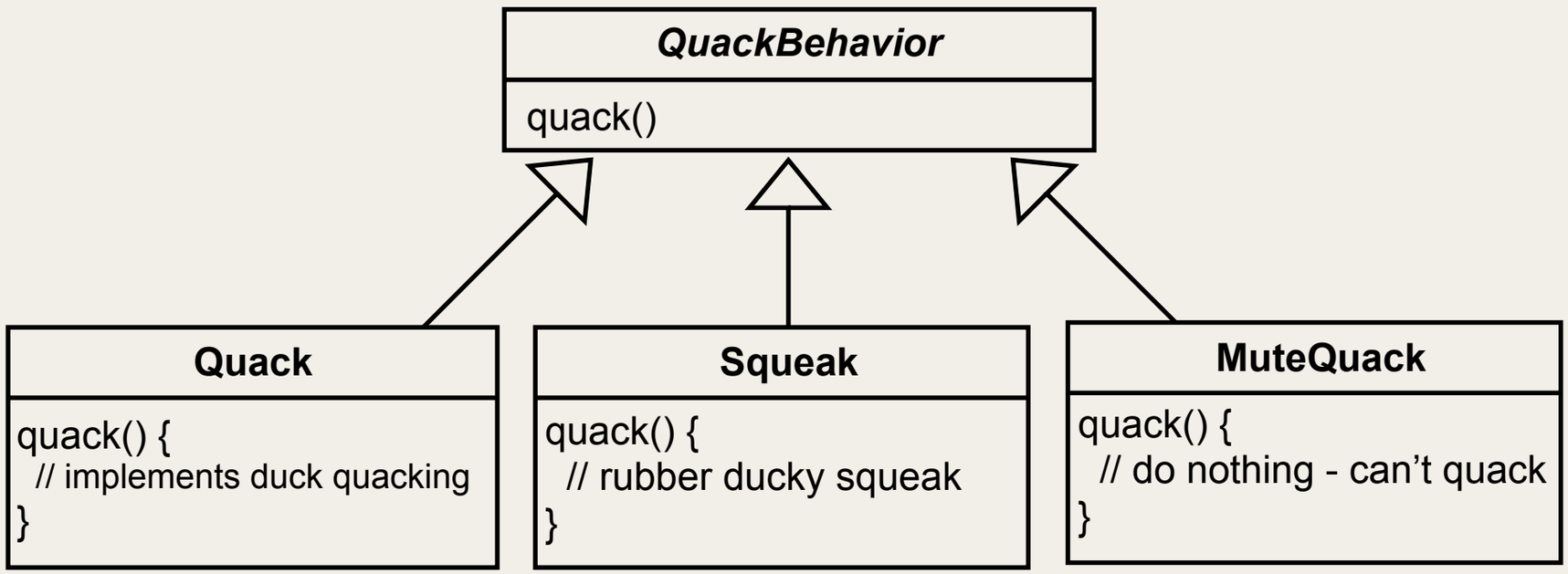
How else could you design this program?

Second, let's separate what changes:

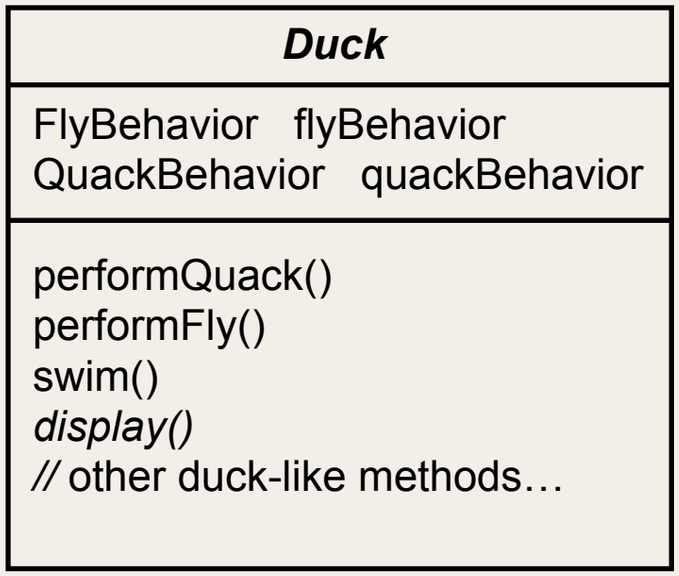


Implementing Duck Behaviors (1/2)



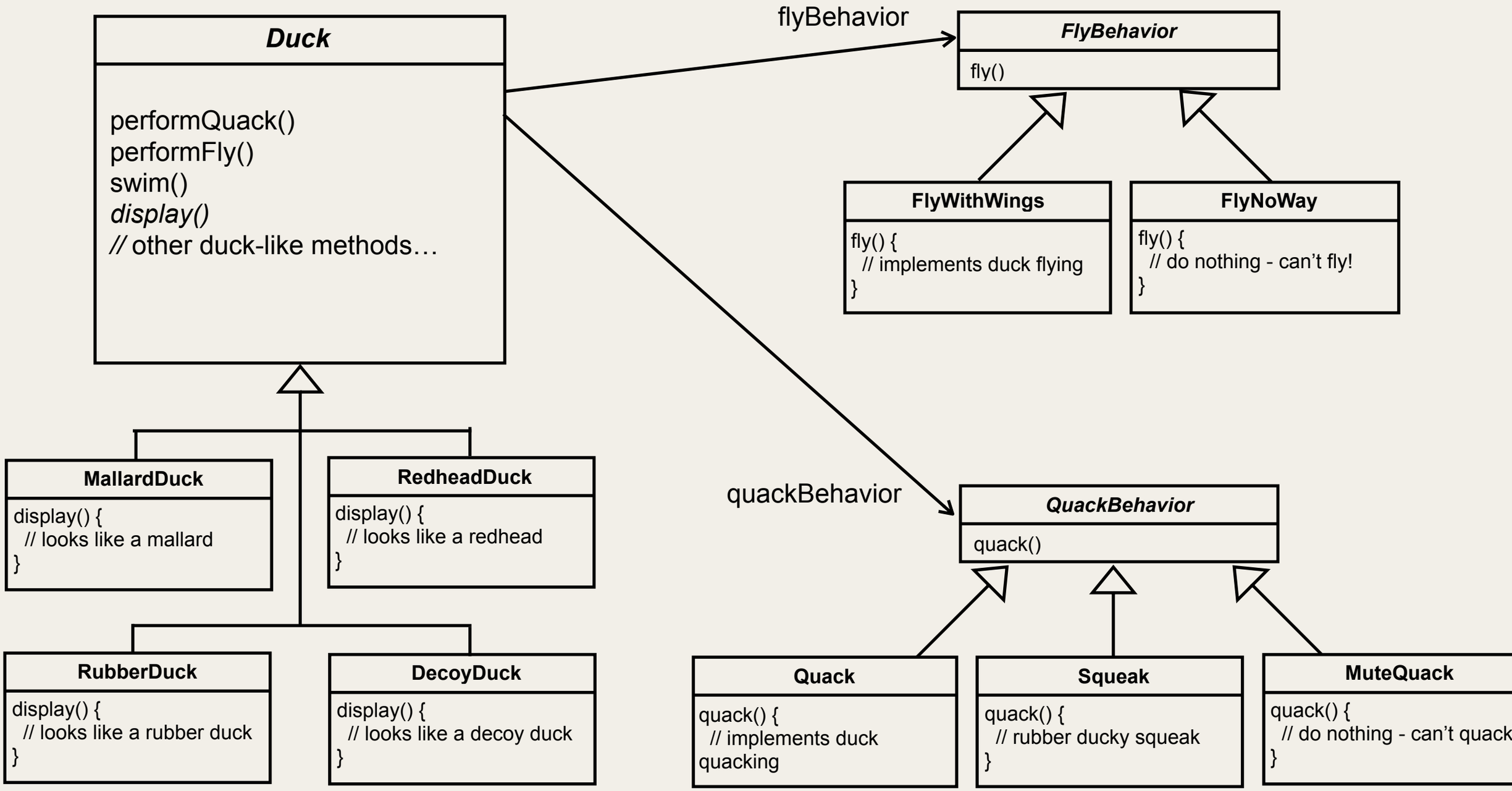


Implementing Duck Behaviors (2/2)




```
class Duck {
public:
  void performQuack() {
    m_quackBehavior->quack();
  }

  void performFly() {
    m_flyBehavior->fly();
  }

  // more…

protected:
  QuackBehavior *m_quackBehavior;
  FlyBehavior *m_flyBehavior;
};


class MallardDuck {
public:
  MallardDuck() {
    m_quackBehavior = new Quack();
    m_flyBehavior = new FlyWithWings();
  }
};
```

All Together:



What just happened?

- We identified some parts of the program that change (and are likely to continue to change).
- We isolated this change using the concept of "behaviors" to create *interfaces* for fly and quack.
- We preferred a "has a" relationship to an "is a" relationship:
  - We started with a MallardDuck "is a" Duck, *and every Duck can fly()*, which is a problem for some Ducks.
  - Now we have, every Duck "has a" FlyBehavior, which is valid for all Ducks.

Why is this so cool / a better design?

- Already in our program with just 4 ducks:
  - We can reuse flying (or quacking) code for ducks that fly (or quack) the same way — this was our original motivation for using inheritance in the first place.
  - Every duck that flies or quacks the same way uses the exact same code, which is only written in one place.
- Moving forward, anticipating change when new ducks are added:
  - We can add as many new flying and quacking behaviors as we want without changing any of our existing classes.
  - We can easily mix and match flying and quacking behaviors to create new types of ducks.
  - If we develop a better flying algorithm for any duck, we can just substitute it.

Ducks in C++, note the use of pointers…

See example code on Moodle.

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Makefiles

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

With our recent upgrades to the Duck program, we now have a lot of files to compile!

- Duck.h/cpp
- DecoyDuck.h/cpp
- FlyBehavior.h/cpp
- FlyNoWay.h/cpp
- FlyWithWings.h/cpp
- MallardDuck.h/cpp
- MuteQuack.h/cpp
- Quack.h/cpp
- QuackBehavior.h/cpp
- RedheadDuck.h/cpp
- RubberDuck.h/cpp
- Squeak.h/cpp
- main.cpp

Intro to Makefiles and the Make Program

- If you have used Visual Studio or Eclipse or some other integrated development environment, you might be familiar with the idea of a Project File or Project Settings or something like this.
- Makefiles are sort of similar. They are like a cross between Visual Studio Project Files and a shell script.
- The most common use for Makefiles is to:
  - Maintain a list of all of the files in your project.
  - Define the appropriate commands to run in order to (re)compile each file and link all the files together to create your program.
- Makefiles are read by a command-line program called *make*.

A Bit More High-Level Background

- Make can be used for lots of things, not just compiling. Makefiles are sort of like really fancy shell scripts.
- However, one of the key distinctions between Makefiles and scripts is that Makefiles know about *dependencies*:
  - If MyProgram.exe depends upon MyClass.cpp, and you edited MyClass.cpp more recently than the last time that I built MyProgram.exe, then I know that MyProgram.exe is out of date and needs to be rebuilt.
  - You have to manually tell make (by writing a Makefile) what these dependencies are in your program, but once you do, make can look at the last modification date for each of your files in order to automatically handle the dependencies.

How does it work?

- At the command line type:
  - make all
  - make some-other-target
- When invoked, the *make* program looks for a file you have written with all your targets and commands in it, typically named *Makefile.*
- Make searches the Makefile for the target you specified on the command line and then runs the commands listed under that target.

Inside a Makefile: Rules

```
target: [dependencies]
[TAB][command1]
[TAB][command2]
…
```

Using a TAB is critical. This has caused me much pain over the years!
The most common mistake in Makefiles is to use spaces rather than a tab.

Use Case

Suppose you want to compile helloworld.
You typed **`make all`** into your shell.
What happens?

Makefile

```
all: helloworld # default target

helloworld: main.o # linking rule
    g++ -o helloworld main.o

main.o: main.cpp # compile rule
    g++ -c main.cpp

clean:
    rm helloworld main.o
```

How make works

- Execution is recursive
- If a target in the Makefile is not needed (it is not a dependency of the target you specified on the command line), its rule is not processed
- Some useful common targets
  - *all*
  - *test*
  - *clean*

Variables

- Some items get repeated
  - compiler, flags, object files...
- We can define them as variables

Variables

```
CXX=g++
CXXFLAGS=-O3
all: helloworld

helloworld: main.o
    $(CXX) $(CXXFLAGS) -o helloworld main.o

main.o: main.cpp
    $(CXX) $(CXXFLAGS) -c main.cpp

clean:
    rm helloworld main.o
```

Static Pattern Rules

- Rules that apply to multiple targets
- Use patterns to define the names of the dependencies from the name of the target.
- Very useful... e.g. the rule to compile .cpp files into .o files is the same for all .cpp files so it would be nice to only specify it once.

Static Pattern Rules

```
targets: target-pattern: dependency-pattern
[TAB]commands
```

A specific example:

```
# This is a space-separated list of object files that need to be
# compiled to create our program
objects = apple.o banana.o

# This is a pattern rule that applies to everything in the list of
# object files specified above.
$(objects): %.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $<
```

Automatic Variables

- The example on the last slide contained an “automatic variable”
- These are variables that are updated each time the rule is executed. You can use them in your pattern rules:

```
$@ - filename of target
$< - first dependency
$^ - all dependencies
$(@D) - directory of the target file (no trailing slash)
```

More Advanced Use of Make: Conditionals

- Make supports *if*, *if-else*, *if-else if-else* statements

```
ifeq ($(CC),gcc)
 libs=$(libs_for_gcc)
else
 libs=$(normal_libs)
endif
```

More Advanced Use of Variables

- If a variable is set to another variable, expansion occurs
- “=” operator: recursive expansion
- “:=” operator: simple expansion

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Summary

- make: flexible, powerful, *not limited to recompiling*
- Used by everybody on every architecture.
- Extensively documented (170 pages)
- http://www.gnu.org/software/make/manual/make.html
- Try it out in your lab on Friday and try to really analyze what is happening on each line of the Makefile you create.
- We'll add more use cases as we continue in the course.