Duck Example

Consider the text example (up to page 6).

- Each type of duck is a subclass of Duck
- Most subclasses implement their own fly() and quack() operations
- **Maintenance appears difficult** as new duck types are required and as behaviour changes.
- **As flying is seen to be subject to change** there is motivation to extract that behaviour into its own class

Page 7: inheritance not working out so well…

- As quacking is seen to be subject to change there is motivation to extract that behaviour into its own class

•The above leads to applying the strategy pattern – once for flying and once for quacking.

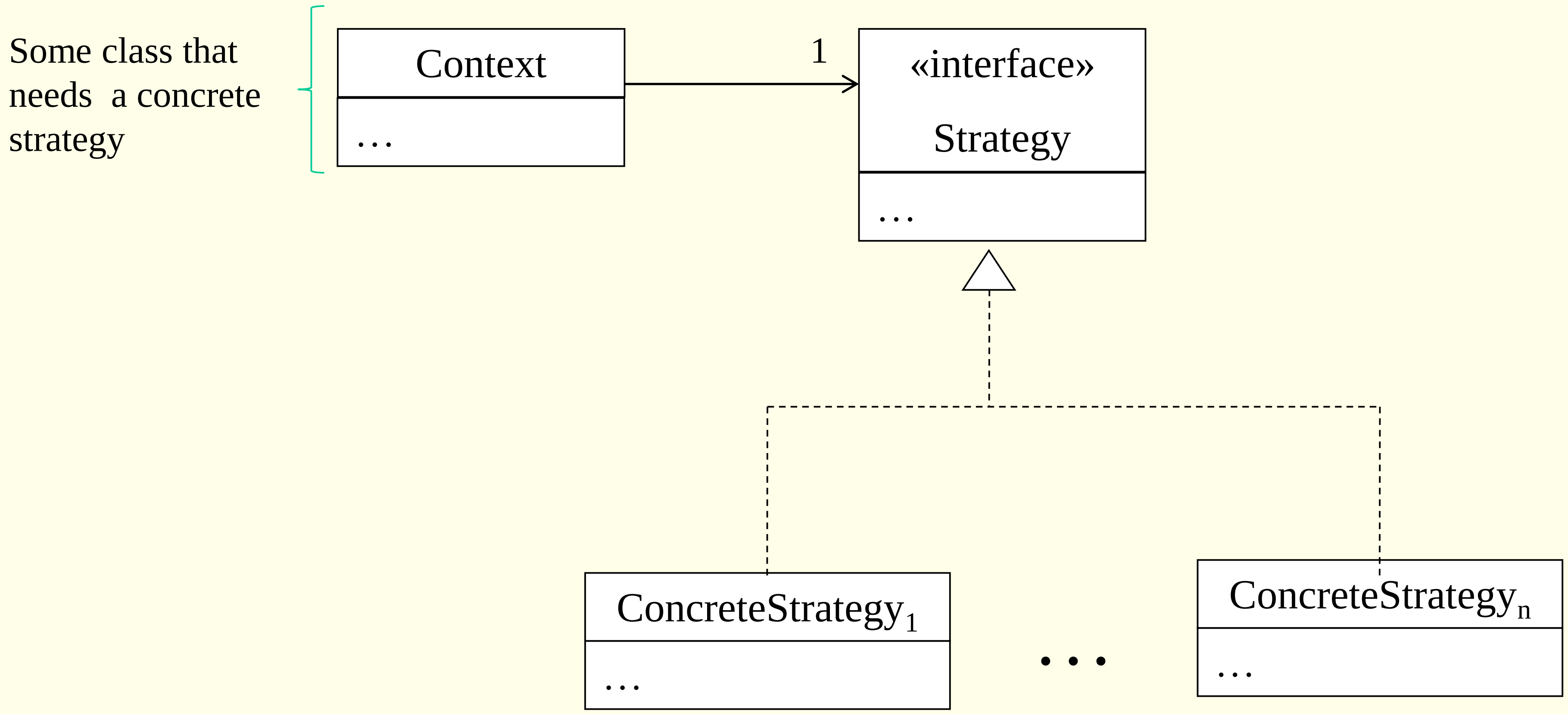
Strategy Pattern

Consider that we have a family or collection of algorithms, one of which is to utilized at runtime by a client.

When we utilize the strategy pattern we make the algorithms interchangeable and implement each in its own subclass of an interface or abstract class.

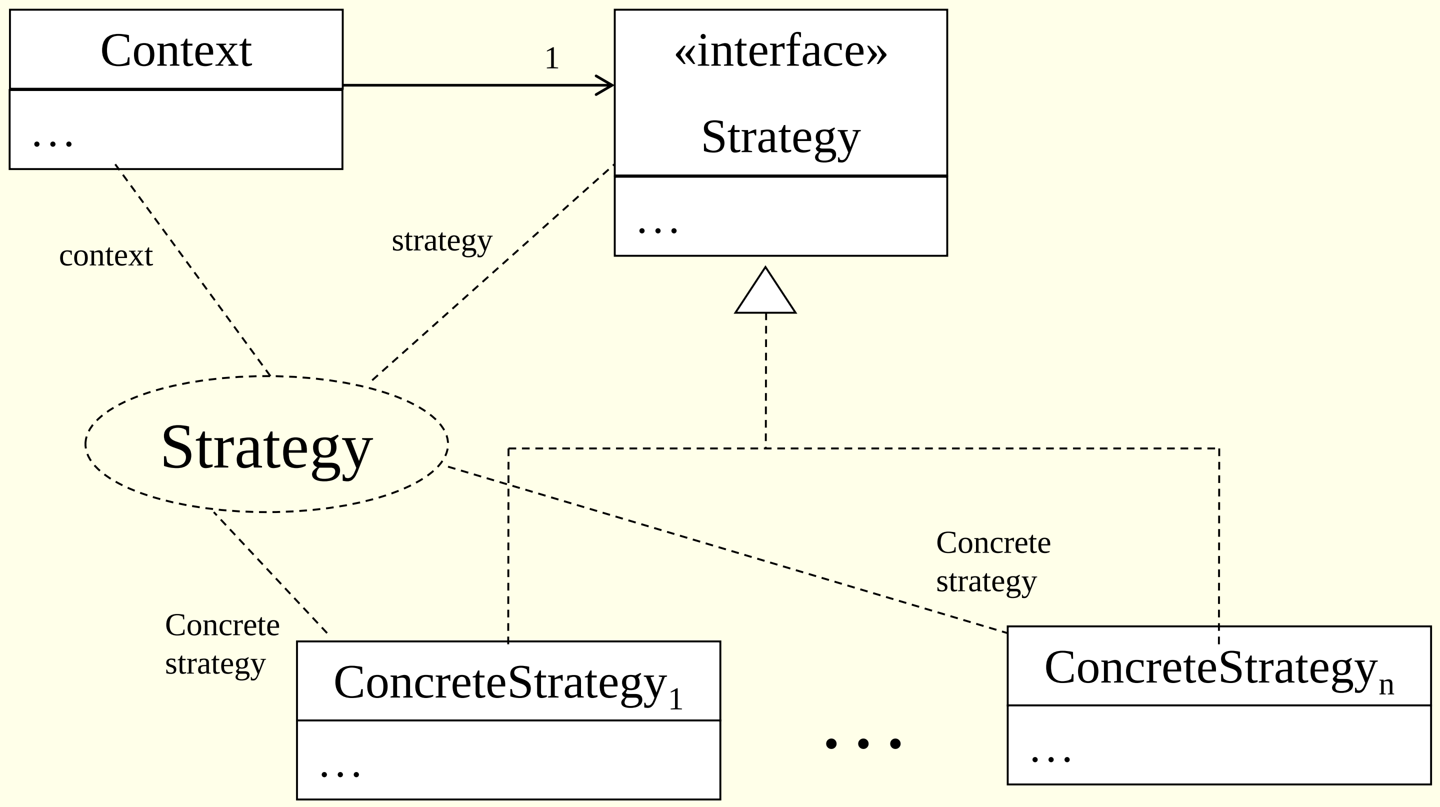
At runtime the appropriate algorithm is associated with the client.

Generic pattern



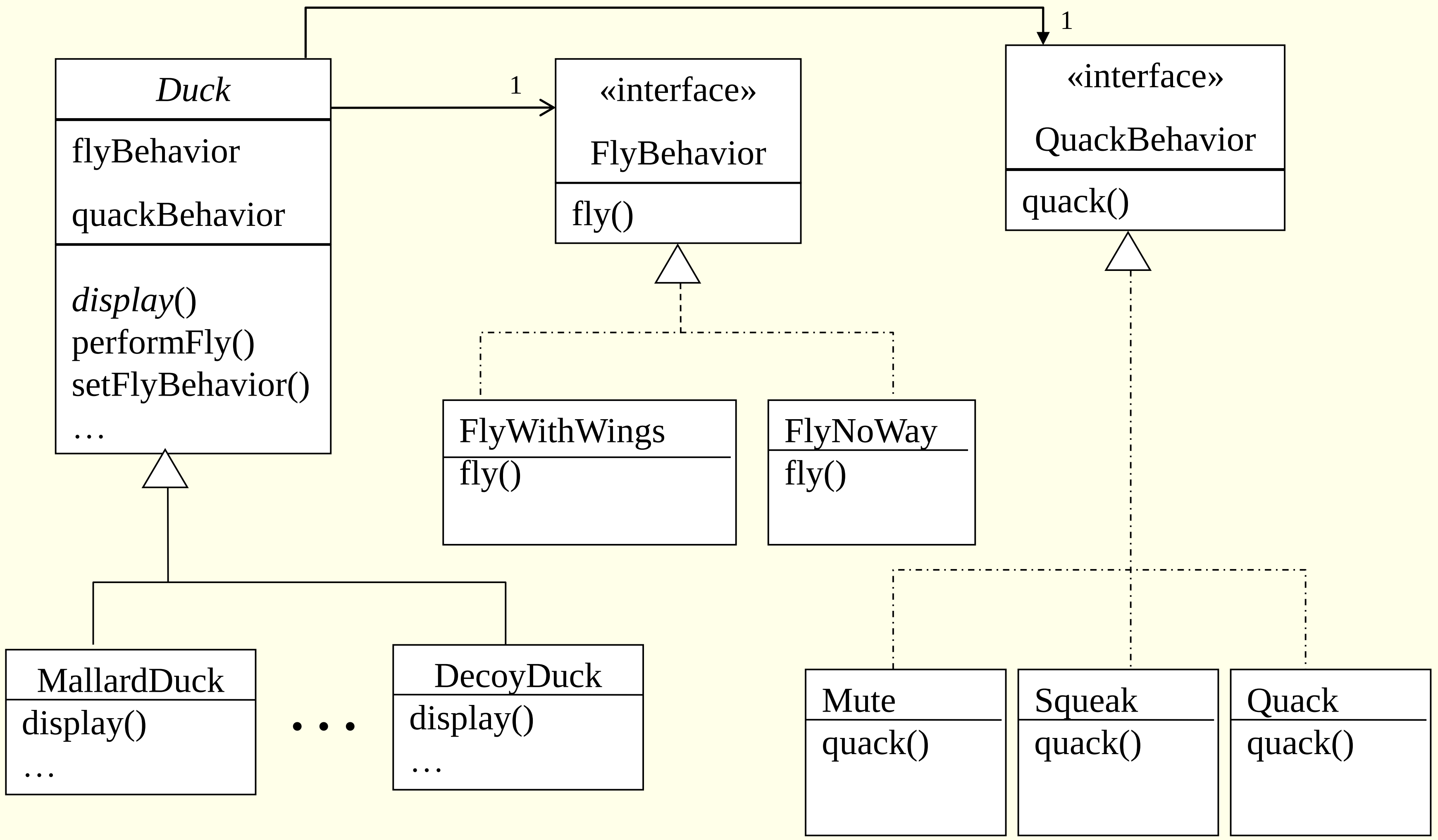
The class diagram indicates the context knows only the strategy interface and not the concrete strategy types.
This is all the context needs to know and this allows us to add/remove strategies without affecting the context.

Generic pattern <u>with collaboration</u>



Strategy is a common interface for all supported algorithms.
The context uses this interface … invokes an algorithm defined by some ConcreteStrategy.
Each concrete strategy implements an algorithm.
The context participant is <u>composed</u> with a ConcreteStrategy object.

Duck Example



The algorithms that implement flying are each encapsulated in a subclass of FlyBehavior, and the algorithms that implement quacking are each encapsulated in a subclass of QuackBehavior. A duck is composed with one flying behaviour and one quacking behaviour.

Ch1 & Duck Example

Consider the text example.

- Examine the code to ensure you understand how the strategy pattern is implemented. Run the example.

Three design principles are discussed.

- Identify the aspects of your application that vary and separate them from what stays the same
- Program to an interface and not an implementation
- Favour composition over inheritance

Duck Example

Draw an object diagram for the case immediately after the statement (page 21):

```
Duck mallard = new MallardDuck();
```

Draw sequence diagrams to illustrate behaviour. What messages are sent

- When main() on page 21 executes?

```
Duck mallard = new MallardDuck();
mallard.performQuack();
mallard.performFly();
. . .
```

Duck Example

Suppose you are a developer and there is a requirement for a new quacking behaviour.

- What must you change in the existing code?
- What classes must you write?

Part of assignment 1:

Create a new algorithm for quacking behavior … talking. Name this class Italk.

Remember … identify yourself via @author …

Create a sequence diagram that shows the messages sent for the code below.

Create an object diagram to show the objects that exist immediately after the last statement below.

```
Duck m = new DecoyDuck();
m.performQuack();
m.setQuackingBehavior(new Italk());
m.performQuack();
```