Lecture 04 - CSCC01 Fall 2019

Week 4 - Agile Planning. Cohesion and Coupling metrics. Design Patterns.

Daniel Razavi

University of Toronto

Sept. 26, 2019

Scrum Team

- ScrumMaster:
  - Educates the team about scrum
  - Ensures practices are followed
  - Facilitates problem solving and runs interference for the team
- Product Owner:
  - Communicates the vision of the product
  - Maximizes the return on investment (ROI) by prioritizing desirable features
- The Team
  - Everyone from every discipline needed to complete the sprint goals

Scrum Master

- Not a traditional lead or management role
- Improves the use of Scrum through coaching, facilitation and rapid elimination of any distractions to the team
- Ensure principles behind scrum remain intact even as the company's own implementation evolves
- Sometimes scrum may be difficult to follow - the team may want to cut corners to deliver a sprint, the scrum master needs to remind them about what they must do
- Nurture sense of ownership

Product Owner

Responsibilities:

- Managing ROI (Return On Investment)
- Establishes a shared vision for the product among the customers / developers
- Knowing what to build and in what order
- Owns the product backlog
- Creating release plans and establishing delivery dates
- Supporting spring planning and reviews
- Representing the customers

The team

Members of the team are primarily developers and testers, however in many projects, specialists from other disciplines may also join the team.

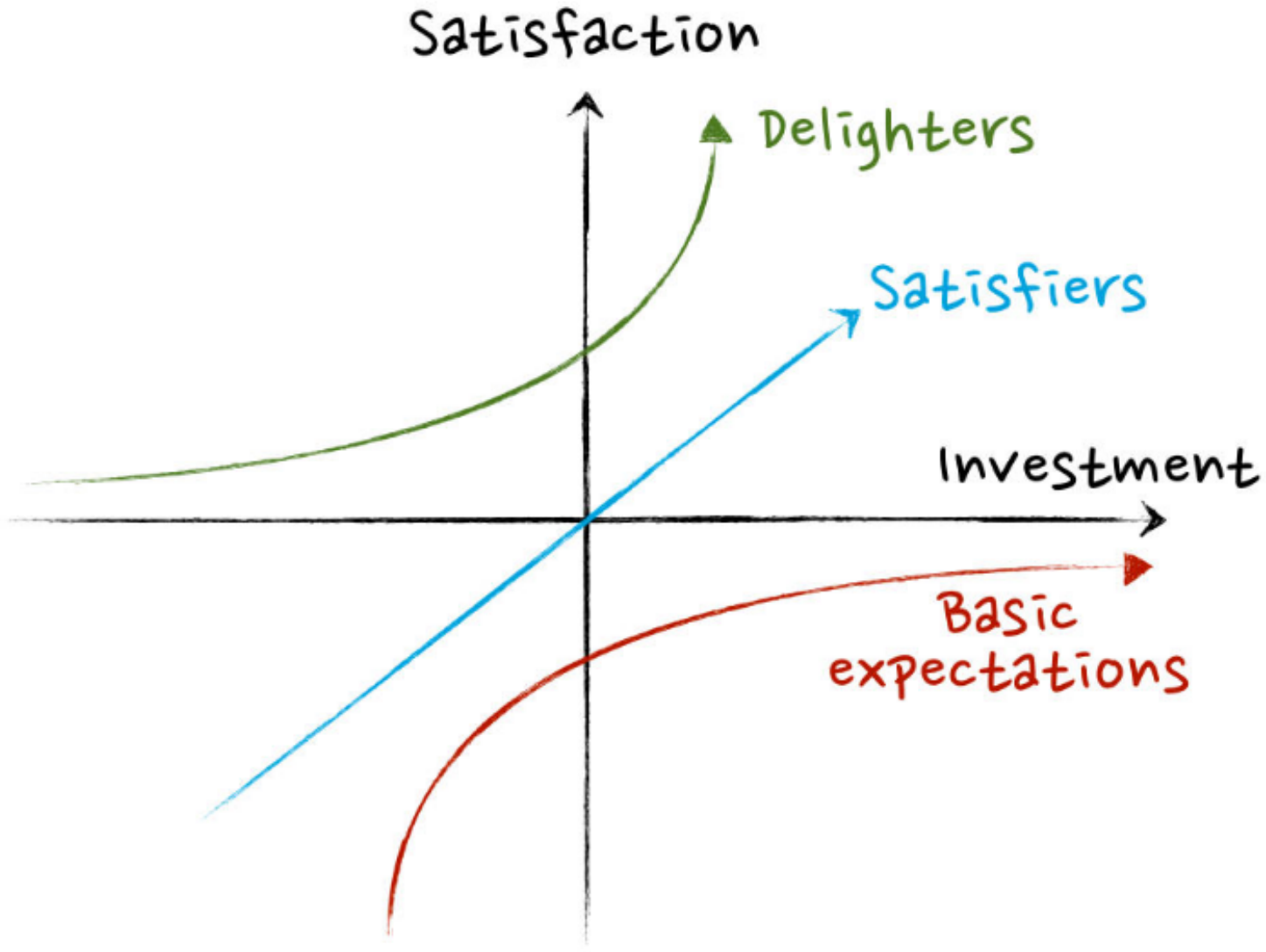
How to prioritize?

- Prior to each sprint, the product owner is allowed to change the priorities in the product backlog
- Decisions can be based on the following criteria:
  - Value: what value does the story add to the player buying the game, helps maximize ROI
  - Cost: some features may prove too costly to implement and affect ROI
  - Risk: uncertainty about value/cost
  - Knowledge: if the product owner doesn't have enough information about feature to do a proper estimate they can introduce a spike to explore it and get more info

VALUE/COST: A PROXY FOR ROI

- A working definition for ROI is Value/Cost.
- Need to have a relative measure of value and cost, w.r.t other features.
- Why?
  - Easy to understand and calculate
  - It makes apparent economical sense
- We have a proxy for effort - story points.
- What is a proxy for value?

KANO MODEL


Satisfaction
Delighters
Satisfiers
Investment
Basic
expectations

Kano model

- Developed by Noriaka Kano in the 1970s and 1980s while studying quality control and customer satisfaction.
- Basic features that a user expects to be there and work will never score highly on satisfaction, but can take inordinate amounts of effort to build and maintain.
- At the opposite end of the spectrum are features that delight the user. These score very highly on satisfaction and in many cases may not take as much investment. Small incremental improvements here have an outsized impact on customer satisfaction.
- Satisfiers: requirements that customers are not expecting, but the more of them that are included, the happier the customer.

What else do we need to account for?

- First, we need to quantify value, and numbers can be read from the Kano graph.
- Next, need to consider the cost of delay. How much relative value do we lose if we deliver x-feature after y-feature?
- Last, but not least, need to consider how much do we reduce risk and uncertainty.
- If two user stories have same ROI, the one with highest CoD must be prioritized first.

Example

Feature	Effort	Value	ROI	CoD	Priority
1	1	3	3	3	1
2	3	6	2	4	2
3	10	20	2	3	3

Exercise

Suppose you have the following user stories on the table:

story ID	estimated cost	value
1	1	2
2	2.5	4
3	1	3
4	1.2	4
5	3	1

From the previous iterations, you estimate the project velocity to be 5. Which user stories, and in which order, should be implemented in the next iteration (ignoring risks)? Why?

Exercise

Things never go as planned in software development. It turns out that the actual costs of the user stories are:

story ID	actual cost
1	2
2	1.5
3	2.5
4	1.7
5	2

Produce a burndown chart of the iteration.

Determining Sprint Length

Typically two to four weeks

- Factors:
  - Frequency of customer feedback
    - How long can the stakeholders go without seeing progress / giving input
  - Team's level of experience
    - Sequential (new teams - using "small waterfall" approach) vs. Parallel (experienced teams)
  - Time overhead for planning and reviews
    - Review and planning require a good chunk of the day regardless of sprint length
  - Ability to plan the entire sprint
    - If there is uncertainty about how to achieve the sprint goal a shorter duration is advisable
  - Intensity of the team over the sprint
    - Avoid mini-crunch periods

Agile Planning

- Traditional software projects place the bulk of project planning at the start of the project
- Agile spreads project planning out over the entire duration of the project
  - Agile teams may actually spend more time overall in planning, but it is not as concentrated

Goal of the Product Backlog

- Prioritize stories so teams are always working on the most important features
- Supports continual planning as the game emerges so the plan matches reality (instead of a stale design document)
- Improves forecasts so stakeholders can make better decisions about the project

Prioritizing the Product Backlog

- Prior to each sprint, the product owner is allowed to change the priorities in the product backlog
- Decisions can be based on the following criteria:
  - Value: what value does the story add to the player buying the game, helps maximize ROI
  - Cost: some features may prove too costly to implement and affect ROI
  - Risk: uncertainty about value/cost
  - Knowledge: if the product owner doesn't have enough information about feature to do a proper estimate they can introduce a spike to explore it and get more info

Velocity

- Velocity is measured by calculating the size of the user stories completed each sprint
- Changes in velocity are an effective way to see how changes / improvements to the agile process affect the team

When do we create the estimates?

- During story workshops or sprint planning meetings
- Estimates should be a relatively quick process involving the following:
  - Expert opinion: domain experts can help inform the group about issues and effort required
  - Analogy: stories are compared to each other to help determine size. Triangulation can be used to help provide better results (compare it to a larger and a smaller story)
  - Disaggregation: larger stories may be difficult to estimate, so they can be broken down into smaller ones to help make the estimates more accurate
- Having to come up with a physical number usually removes any fuzziness about the requirements of the story

Converting Story points to hours

- Suppose for some reason you track how long every one-story-point story takes a given team to develop. If you did, you'd likely notice that while the elapsed time to complete each story varied, the amount of time spent on one-point stories takes on the shape of the familiar normal distribution.
- A tool used to convert story points to hours, is ideal days.
  - Another unit of measure - can be used as a transition for teams that are new to agile
  - Represents an ideal day of work - with no interruptions (phone calls, questions, broken builds, etc.)
  - Associated with something really so easier to grasp initially
  - Doesn't mean an actual day of work to finish, useful for relative measurements when compared to other stories
- That said, converting story points to hours, is something to be done carefully.

Breaking down user stories into tasks

- Tasks are estimated in hours
  - Estimation is an ideal time (without interruptions / problems)
  - Smaller tasks estimates are more accurate than large
- After all tasks have been estimated the hours are totaled up and compared against the remaining hours in the sprint backlog
  - If there is room, the PBI is added and the team commits to completing the PBI
  - If the new PBI overflows the spring backlog, the team does not commit
    - PBI can be returned to the product backlog and a smaller PBI chosen instead
    - Break original PBI into smaller chunks
    - Drop an item already in the backlog to make room
    - Product owner can help decide best course of action

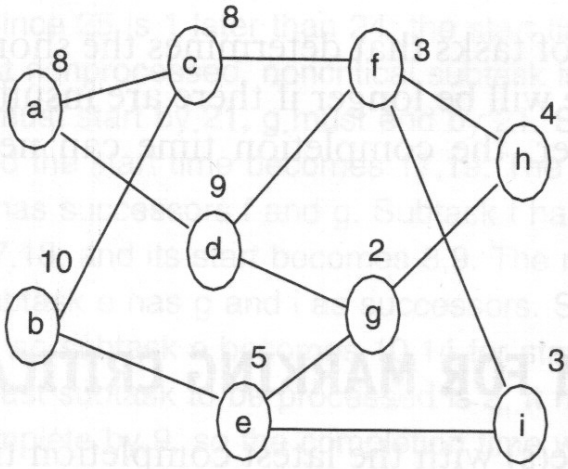
What is critical path?

- Critical path is the sequential activities from start to the end of a user story. Although many user stories have only one critical path, some may have more than one critical paths depending on the flow logic used in the user story implementation.
- If there is a delay in any of the activities under the critical path, there will be a delay of the user story delivery.

Key steps in Critical Path Method

- Step 1: Activity specification
  - Break down a user story in a list of activities.
- Step 2: Activity sequence establishment
  - Need to ask three questions for each task of your list.
    - Which tasks should take place before this task happens.
    - Which tasks should be completed at the same time as this task.
    - Which tasks should happen immediately after this task.
- Step 3: Network diagram
  - Once the activity sequence is correctly identified, the network diagram can be drawn.
- Step 4: Identify critical path
  - The critical path is the longest path of the network diagram. If an activity of this path is delayed, the user story will be delayed.

EXAMPLE



TaskID	Time	Dep.	CP
a	8		
b	10		*
c	8	a,b	*
d	9	a	
e	5	b	
f	3	c	*
g	2	d	
h	4	f,g	*
i	3	e,f	

Slack Time

Tasks on the critical path have to started as early as possible or else the whole project will be delayed. However tasks not on the critical path have some flexibility on when they are started. This flexibility is called the slack time.

TaskID	Start Time	Comp.Time	CP
a	0,1	8,9	
b	0	10	*
c	10	18	*
d	8,9	17,18	
e	10,14	15,19	
f	18	21	*
g	17,19	19,21	
h	21	25	*
i	21,22	24,25	

What is PERT?

- PERT (Program Evaluation and Review Technique) is one of the successful and proven methods among the many other techniques, such as, CPM, Function Point Counting, Top-Down Estimating, WAVE, etc.
- PERT was initially created by the US Navy in the late 1950s. The pilot project was for developing Ballistic Missiles and there have been thousands of contractors involved.
- After PERT methodology was employed for this project, it actually ended two years ahead of its initial schedule.

The PERT Basics

- At the core, PERT is all about management probabilities. Therefore, PERT involves in many simple statistical methods as well.
- Sometimes, people categorize and put PERT and CPM together. Although CPM (Critical Path Method) shares some characteristics with PERT, PERT has a different focus.
- Same as most of other estimation techniques, PERT also breaks down the tasks into detailed activities.

Measuring Cohesion

Program Slices

```
# x, y: positive integers
# z is the product of x and y
z = 0
while x > 0:
    z = z + y
    x = x - 1
return z
```

- *Output Slice:* statements that affect the value of the output
- *Input Slice:* statements that are affected by the value of the input
- Either can be used to define cohesion.
- We will use *output slices* following Bieman and Ott.
- The code above contains one output slice, computing z.

Measuring Cohesion

Tokens

```
# x, y: positive integers
# q, x: quotient and remainder of
# x divided by y
q = 0
while x >= y:
    q = q + 1
    x = x - y
```

- How many output slices are in this piece of code?
- *Token:* A variable or a constant (`x, y, q, 0`)
- *Glue token:* token present in more than one slice (`x,y`)
- *Superglue token:* token present in all slices (none)

Functional Cohesion Measures

- *Weak functional cohesion:* The ratio of glue tokens versus total tokens
- *Strong functional cohesion:* The ratio of superglue tokens versus total tokens
- *Adhesiveness of a token:* Percentage of output tokens containing that token
- *Code adhesiveness:* The average adhesiveness of all tokens

Measuring Cohesion Example

Tokens

```
# x, y: positive integers
# q, x: quotient and remainder of
# x divided by y
q = 0
while x >= y:
    q = q + 1
    x = x - y
```

- Output slices: 2
- Adhesiveness of tokens:
  - A(x)=A(y)=A(q)=A(0)=1/2=50%
  - A=(2*50%+2*50%)/4=50%

Measuring Coupling

Defining coupling

- Coupling may be defined as the degree of interdependence that exists between software modules and how closely they are connected to each other.
- There are several dimensions of coupling:
  - Content coupling: this is a type of coupling in which a particular module can access or modify the content of any other module.
  - Common coupling: this is a type of coupling in which you have multiple modules having access to a shared global data
  - Control coupling: this is a type of coupling in which one module can change the flow of execution of another module
  - Data coupling: in this type of coupling, two modules interact by exchanging or passing data as a parameter

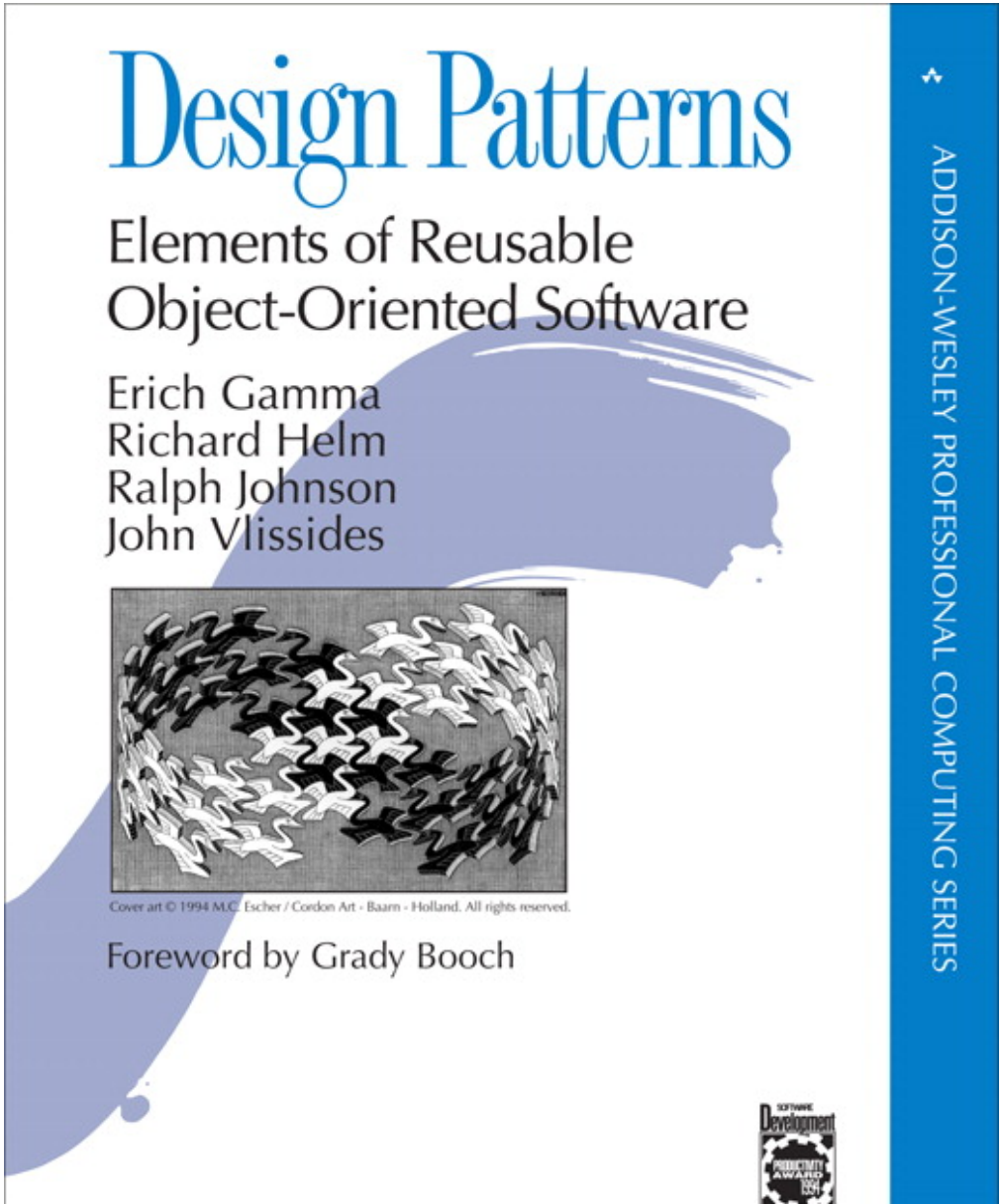
Measuring Coupling

Module coupling

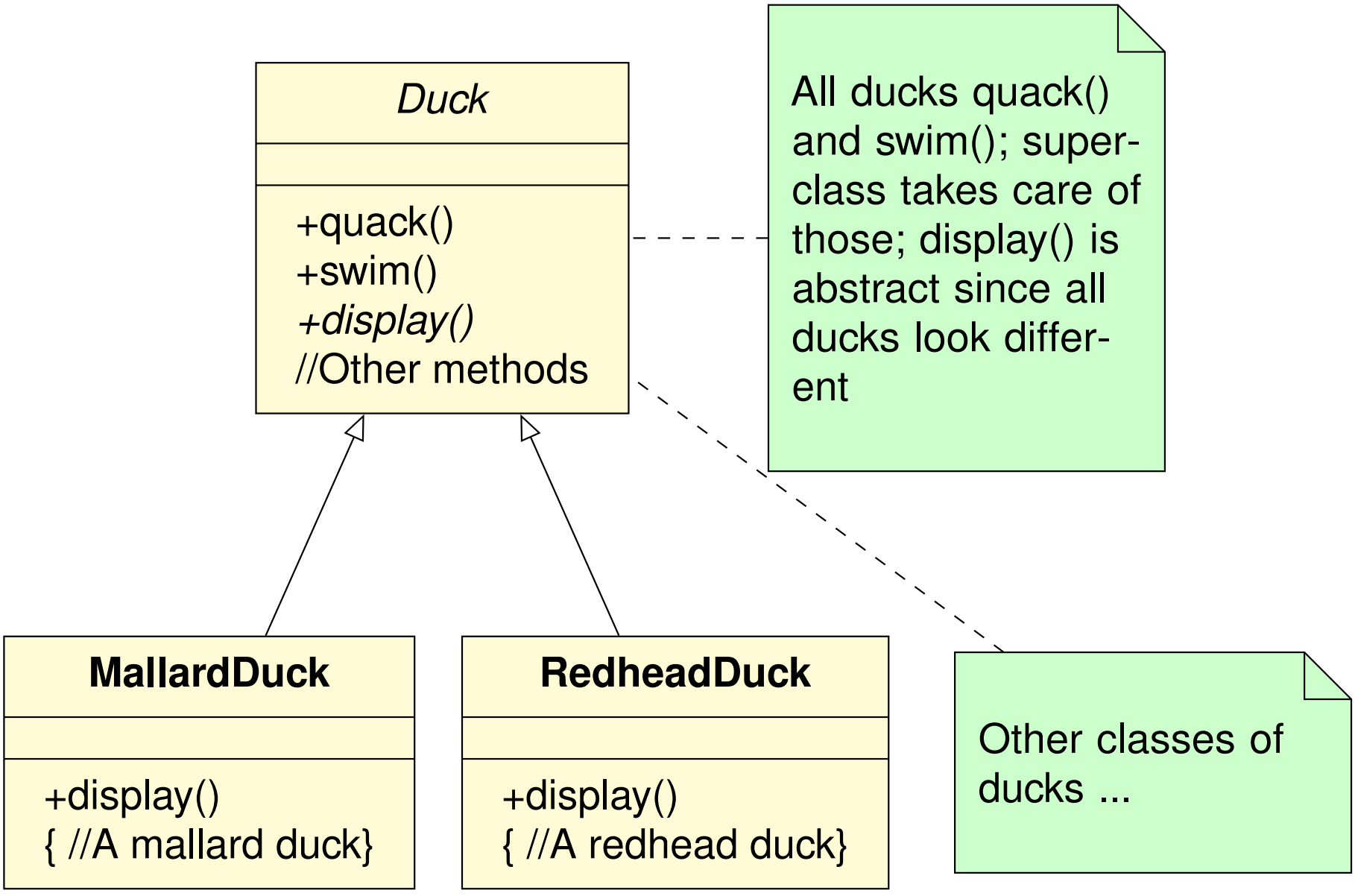
- Data and control flow coupling
  - Let **di** (resp. **do**) be the number of data input (resp. output) parameters
  - Let **ci** (resp. **co**) be the number of control input (resp. output) parameters
- Global coupling
  - Let **gd** (resp. **gc**) be the number of data global (resp. control) variables
- Environmental coupling
  - Let **w** number of modules called (fan-out)
  - Let **r** number of modules calling the module under consideration (fan-in)

$$\text{Coupling(C)} = 1 - \frac{1}{di+2ci+do+2co+gd+2gc+w+r}$$

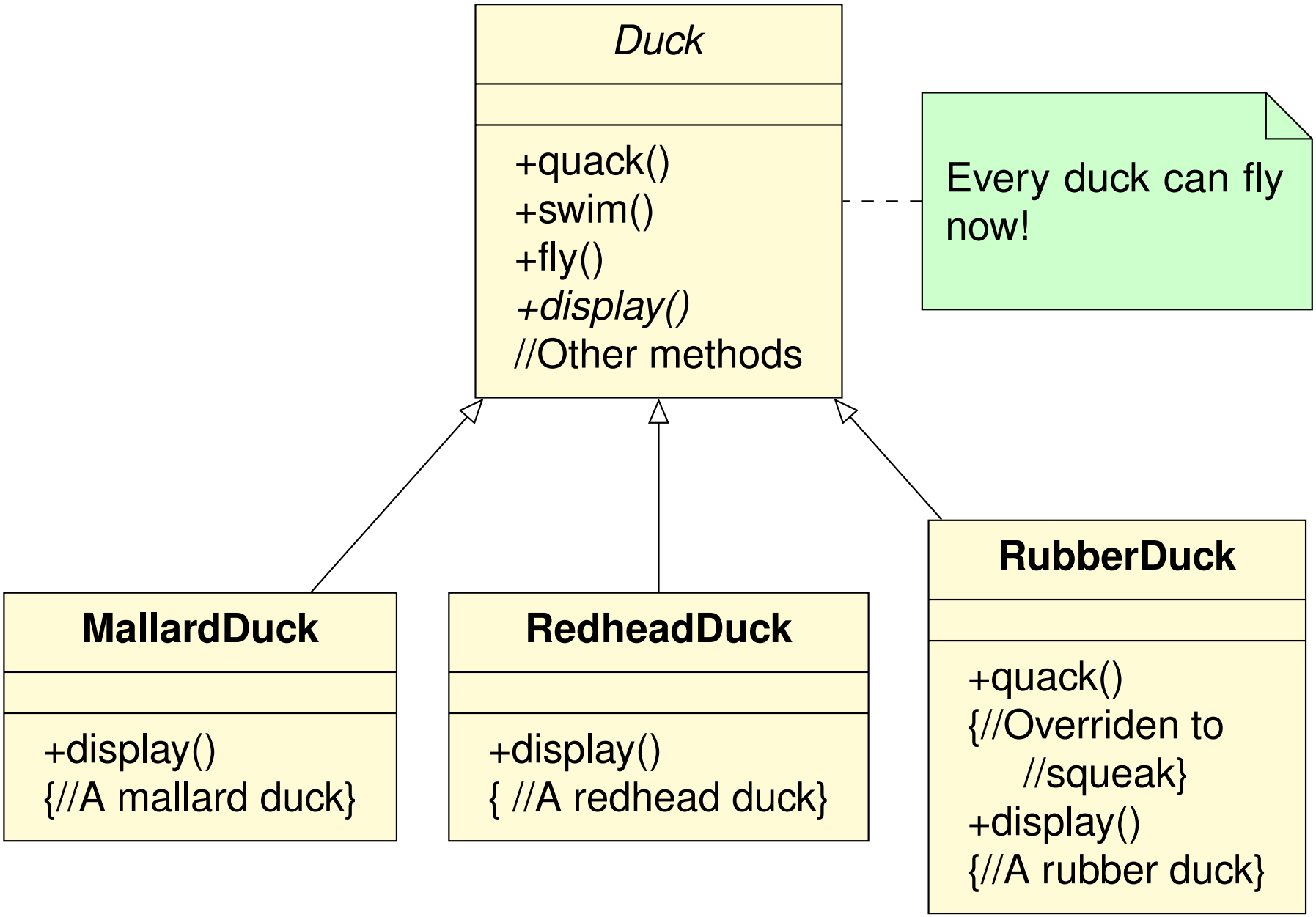
Design Patterns



Why do we need patterns?



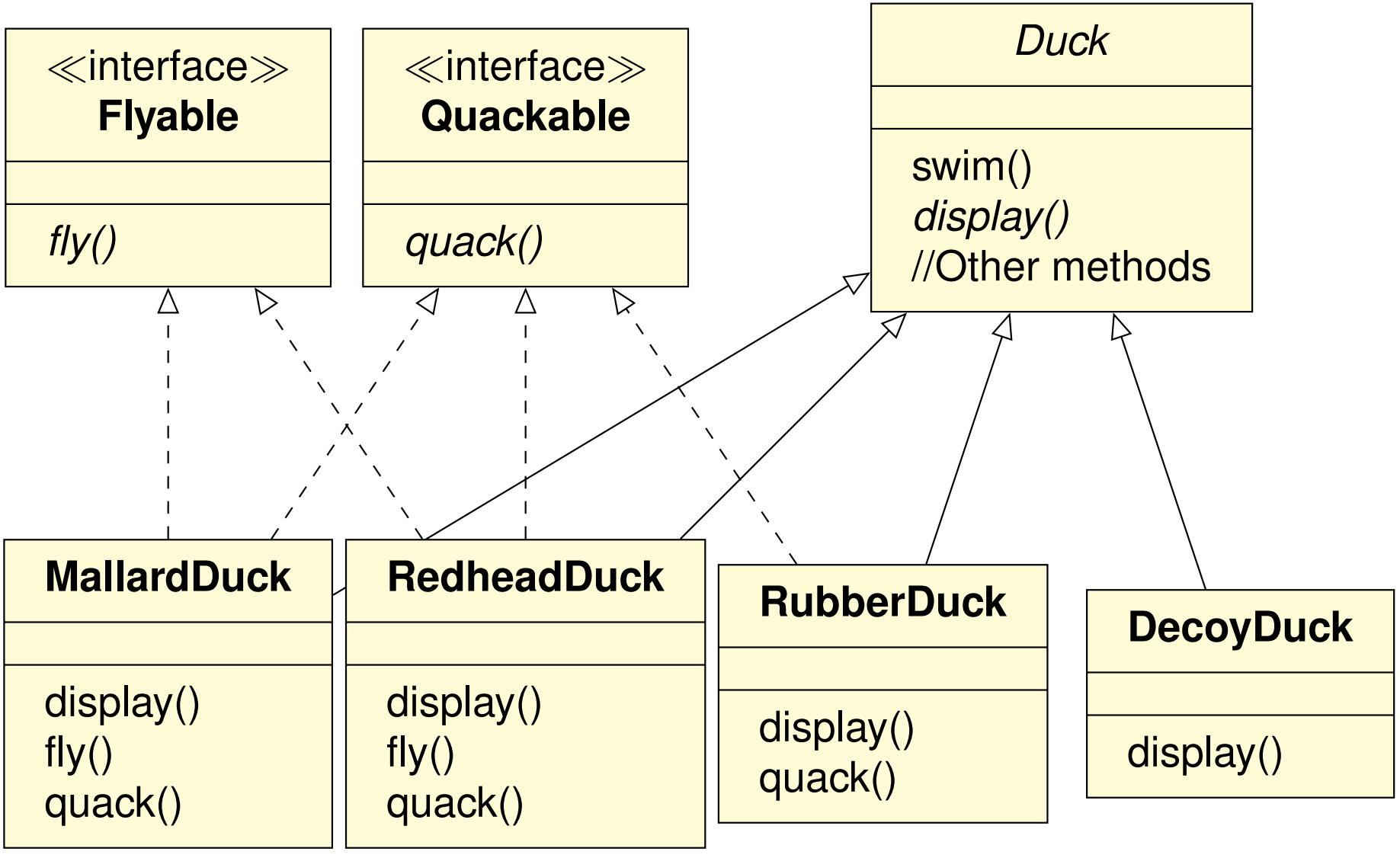
A QUICK CHANGE IS NEEDED - DUCKS NEED TO FLY!



An attempt to solve the problem

- Override the `fly()` method in `RubberDuck` same way we have already done with `quack()`.
- But what happens if we add a `DecoyDuck` made of wood? They are not supposed to fly or quack!
- Which of the folowing are disadvantages of using inheritance to provide specific Duck behavior? (Chose all that apply).
  A. Code is duplicated across subclasses.
  B. Runtime behavior changes are difficult.
  C. We can't make a duck dance.
  D. Hard to gain knowledge of all duck behaviors.
  E. Ducks can't fly and quack at the same time.
  F. Changes can unintentionally affect other ducks.

How about interfaces?



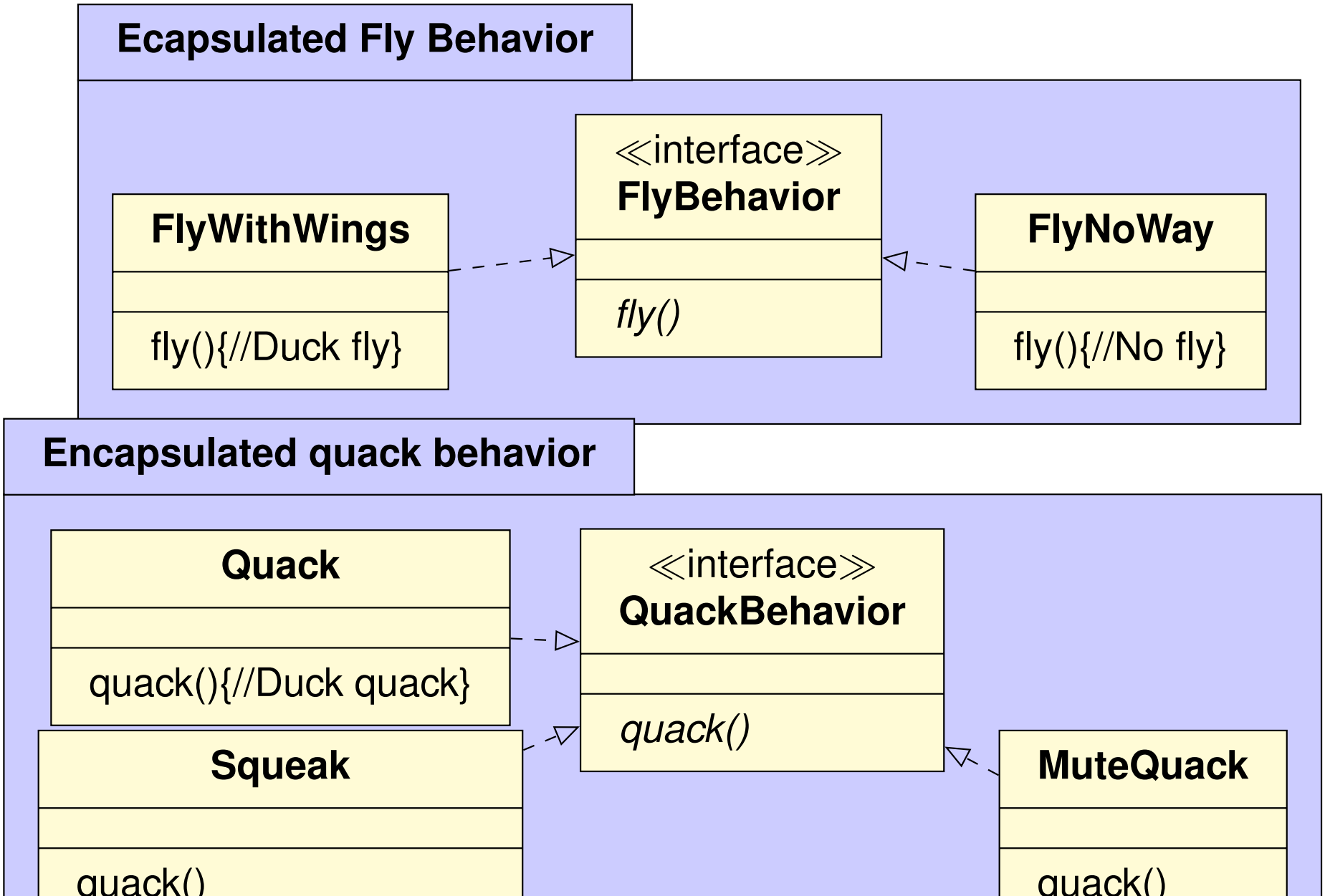
The solution

- The use of interfaces solves part of the problem (no flying rubber ducks), it compleletely destroys reuse of code, creating a huge maintenace problem.
- This can be addressed by using a **design principle**:
  - Identify the aspects of your application that vary and spearate them from what stays the same.
  - Or, take the parts that vary and encapsulate them so later you can modifyor extend them without affecting those that don't.
- Let's do it:
  - The parts of `Duck` that vary are `fly()` and `quack()`.
  - Pull them out of `Duck` class and create a new set of classes to represent each behavior.

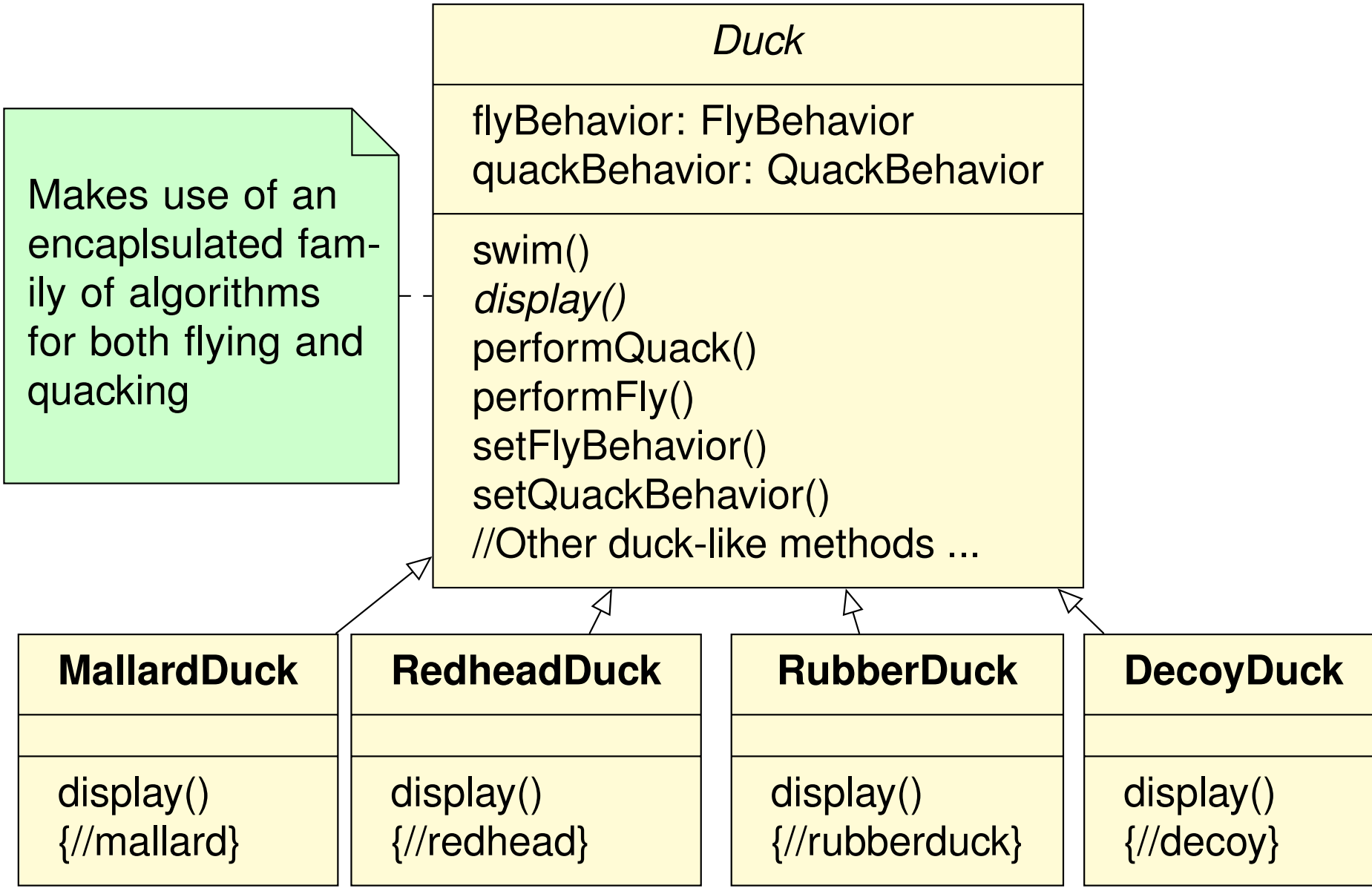
More design principles

- Meanwhile we need to keep things flexible, so we might want to assign a fly behavior to a `MallardDuck` at instantiation time.
- To successfully do so, we need to **program to an interface, not to an implementation**.
- We also need to **favour the composition over inheritance**.

SEPARATING AND IMPLEMENTING DUCK BEHAVIORS



DUCK HAS A FLYBEHAVIOR AND A QUACKBEHAVIOR



The strategy pattern

- **The strategy Pattern** defines a family of algorithms, encaplsulates each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- For an example, see the solution of the duck problem.
- Exercise:
  - A duck call is a device that hunters use to mimic the calls (quacks) of ducks. How would you implement your own duck call that does not inherit from the Duck class?

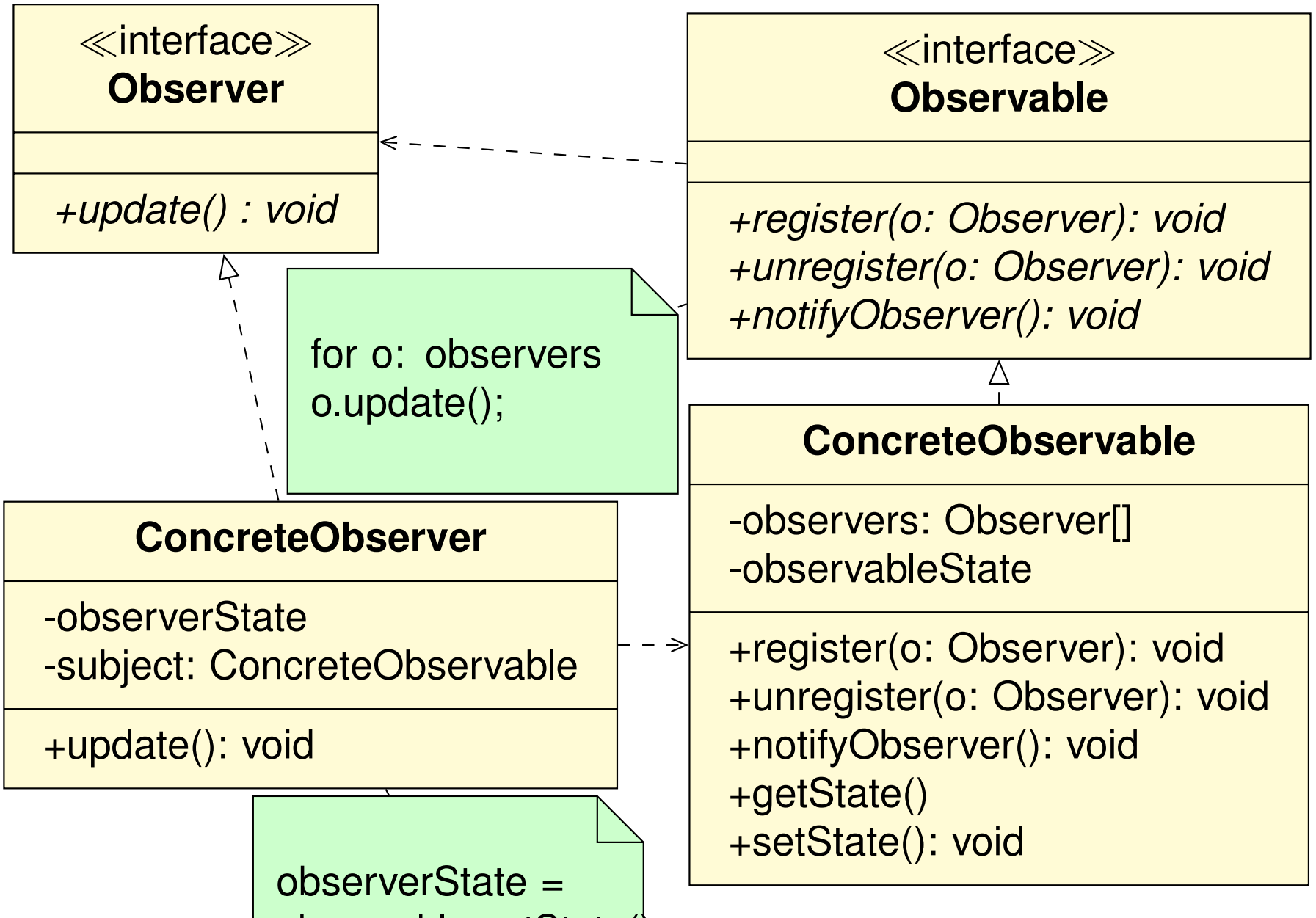
Where are we so far?

- OO Basics
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
- OO Design Principles
  - Encapsulate what varies
  - Favor composition over inheritance
  - Program to interfaces, no implementations.
- OO Design patterns
  - We did see strategy pattern
  - We will see a few more in the near future.

Observer Design Pattern

- Problem:
  - Need to maintain consistency between related objects.
  - Two aspects, one dependent on the other.
  - An object should be able to notify other objects without making assumptions about who these objects are.
- Example - Magazine subscription:
  - A magazine publisher goes into business and stars publishing magazines
  - You subscribe to the publisher - every time there is a new issue, it gets delievred to you, fors as long as you remain a subscriber.
  - You unsubsribe when you do not want the magazine anymore.
  - Meanwhile the publisher remains in business, other people keep subscribing and ubsubscribing to its magazine.

Observer Pattern UML



Observer: Java implementation