Brushwork (Iteration #1) Retrospective

CSCI-3081: Program Design and Development

Retrospective

- looking back on or dealing with past events or situations.

Fowler (the UML guru) Iteration Retrospective

- At the end of each iteration, conduct an iteration retrospective, whereby the team assembles to consider how things went and how they can be improved.
- A good way to do this is to make a list with 3 categories:
  - Keep: things that worked well that you want to ensure you continue to do.
  - Problems: areas that aren't working well.
  - Try: changes to your process to improve it.
- www.retrospectives.com adds one more category/question:
  - What still puzzles us?

Keep

- communication
  - via facebook multiperson chat
  - github issues tracker
  - slack
  - face-to-face meetings in common areas
  - photo text message
- setting intermediate deadlines / milestones
- in person meetings more effective than Skype
- sitting at the same console, programming together (pair / team programming)
- talk / discuss before coding

Problems

- how to divide workflow evenly
  - progression / dependencies
    - amount of work
- balance between design / implementation, maybe more design up front

Try

- intermediate milestones / assigning tasks, watching out for dependencies, tackle dependencies first
- new forms of team communication
- pair programming
- standards, e.g., how to comment code
- one person working on a file at a time and/or learn more about git to facilitate merging

What Still Puzzles Us?

- differences between installations / environments with graphics programming.
- memory leaks, identifying, using valgrind, this is a difference w/ Java
- github: merging and branching
- more on design, helper functions where do they go

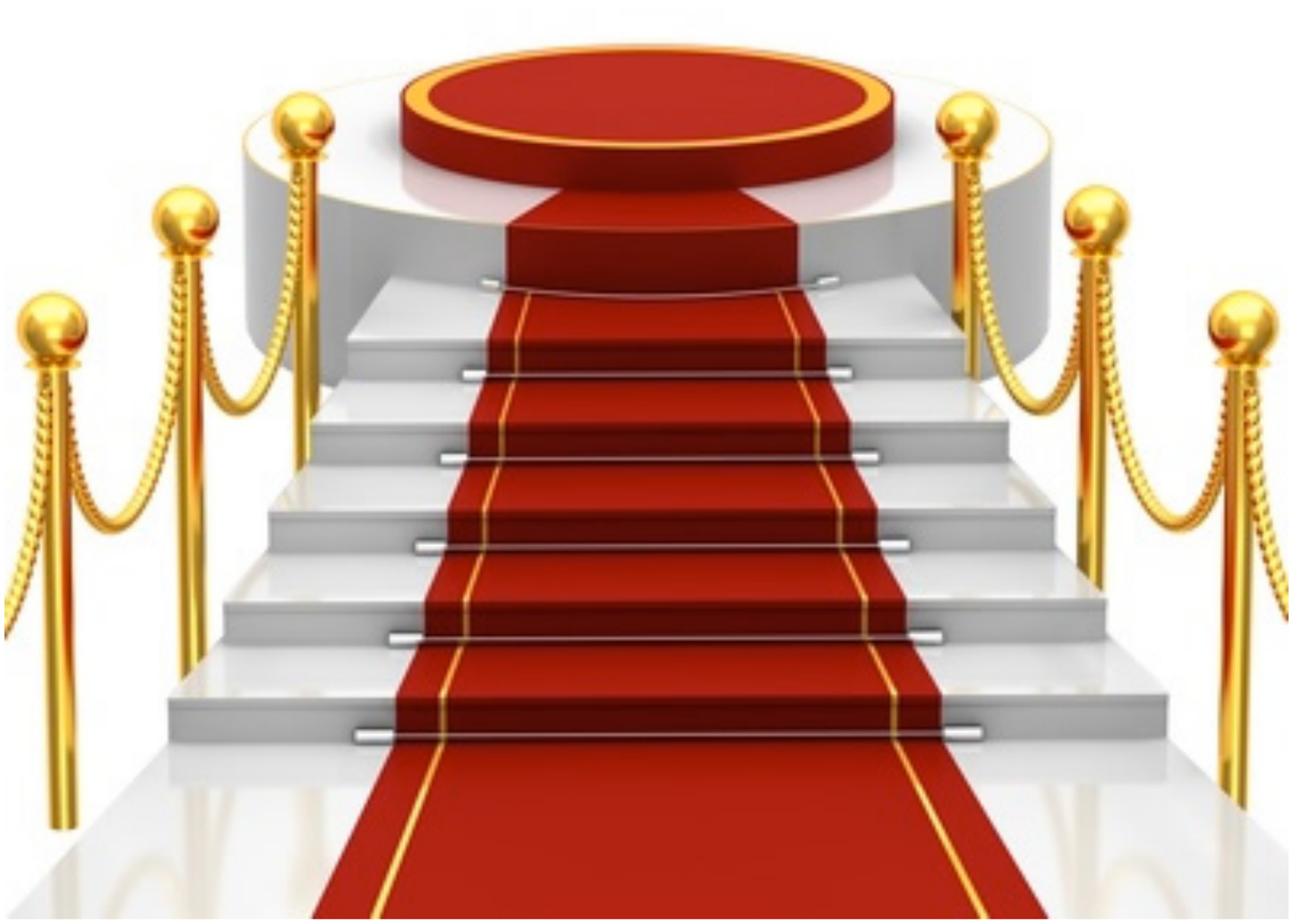
How did you handle key design decisions in your group?

- What is the relationship between Tools and Masks? Do you have a Mask class? Do you have subclasses of Tool and/or Mask?
- How do you create new Tools? How many files and how many places in the code would need to be edited in order to create a new tool in your design?
- How do you handle the “special case” of the eraser? It needs access to the canvas background color whereas all other tools just use the canvas current color. How do you pass the background color to the eraser? How does the eraser apply itself to the canvas - does it reuse code used for the other tools without duplicating the code?
- Were there other important design decisions in your implementation?

Midterm Exam Mini Review

UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

CSci-3081W
Slide Hall of Fame



UNIVERSITY OF MINNESOTA
PROFESSOR DANIEL F. KEEFE

Class Implementation

Specify the interface in the Date.h file:

```
class Date {
public:
  Date(int y, int m, int d);
  virtual ~Date();
  string print();

private:
  int year;
  int month;
  int day;
};
```

Define the details of what happens when each method is called within the Date.cpp file:

```
Date::Date(int y, int m, int d) {
  year = y;
  month = m;
  day = d;
}

Date::~Date() {
}

void Date::print() {
  cout << year << " " << month
       << " " << day << endl;
}
```

C++ Example of a Class Interface with Mixed Levels of Abstraction

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

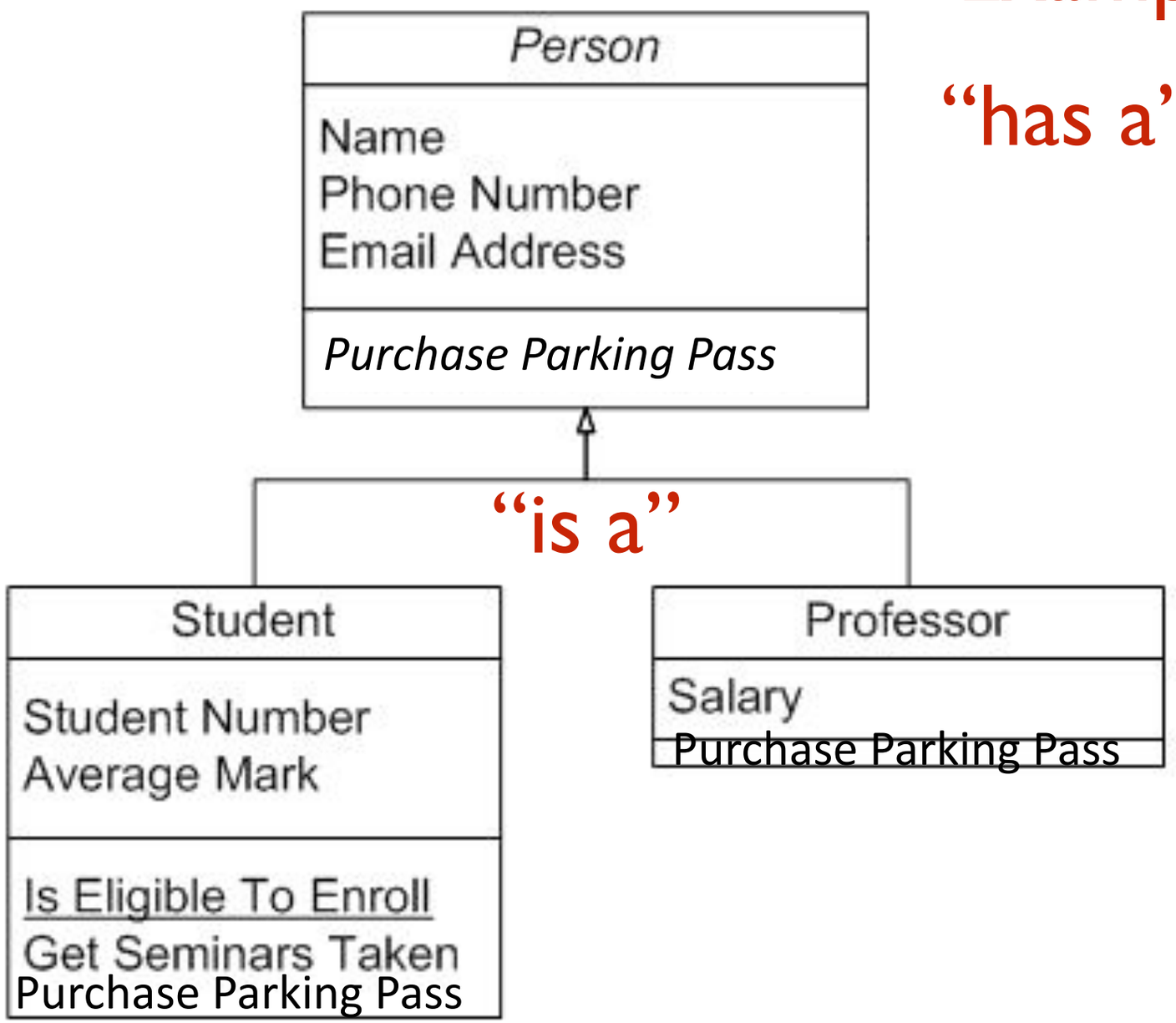
The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

- Poor abstraction, mixed levels of abstraction.
- Each class should implement one and only one abstract data type.

Another Example (by the way this is a UML diagram)

Example of Containment



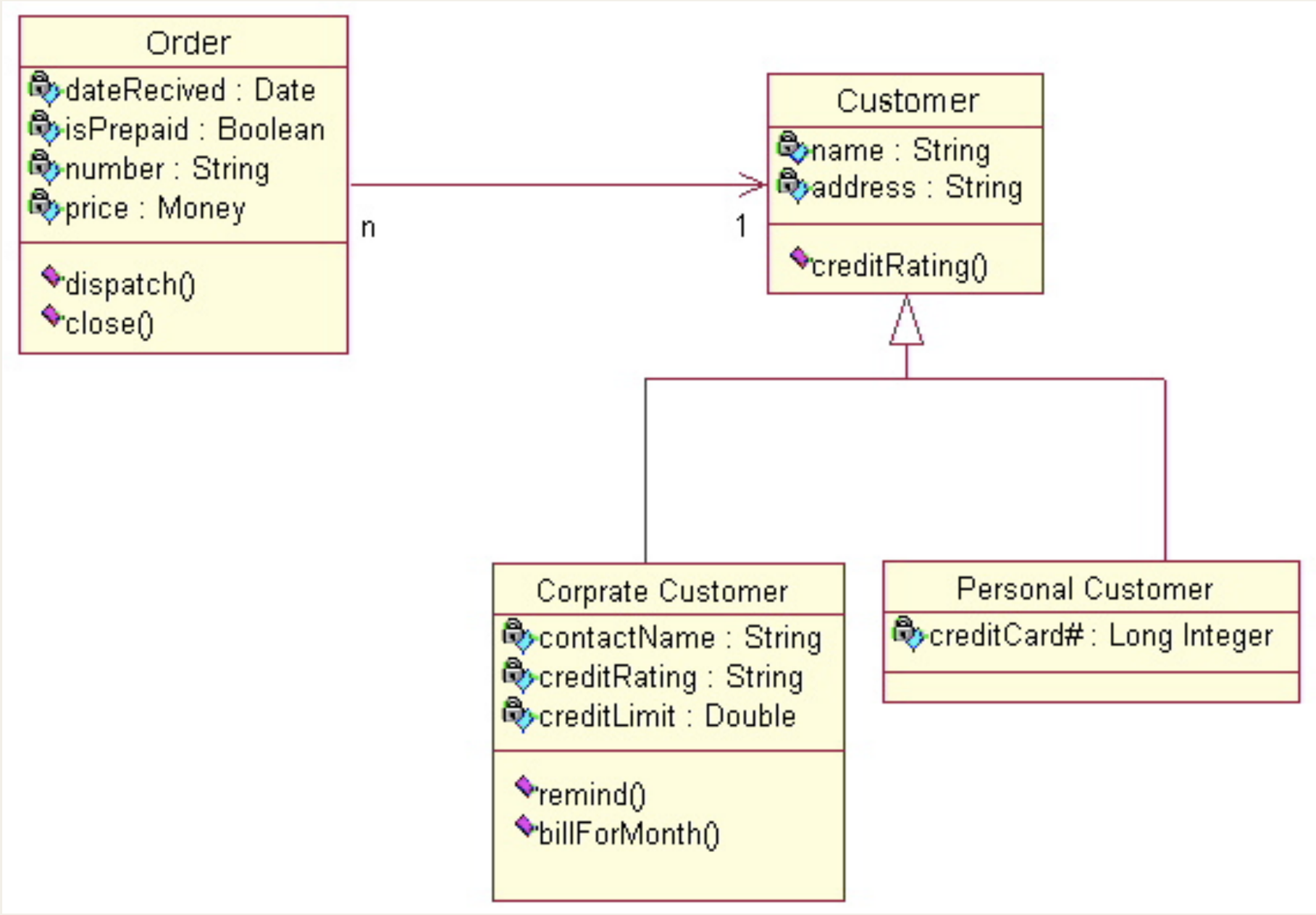
Example of Inheritance

What you need to learn about building C++ programs

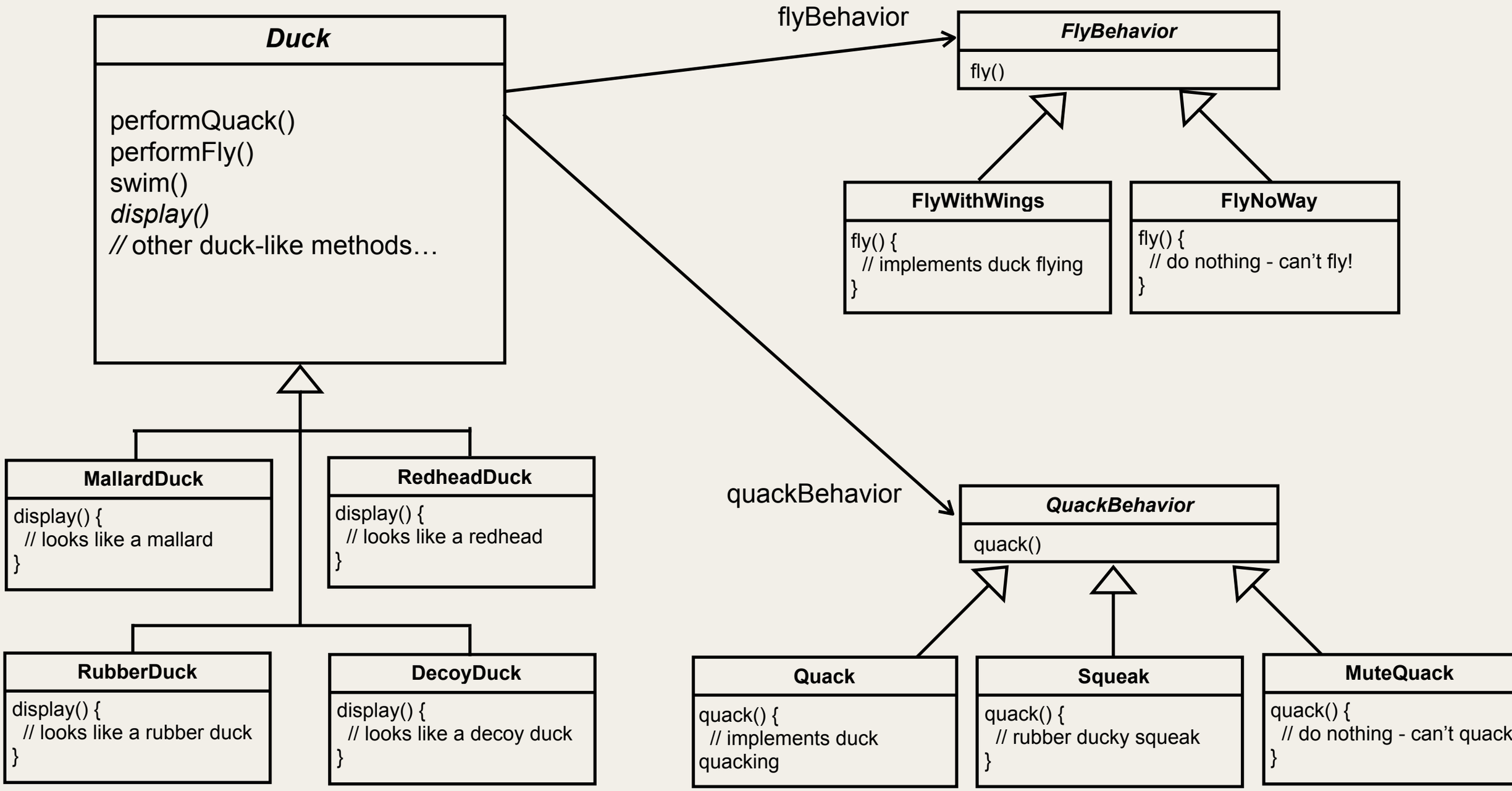
- There are two steps to building a C++ program:
  1. compiling
  2. linking
- Compiling: Every .cpp file that is part of the program needs to be compiled separately into an object file.
  - File1.cpp gets compiled into File1.o
  - File2.cpp gets compiled into File2.o
  - …
- Linking: All of the resulting .o files are then linked together to create an executable program.
  - File1.o, File2.o, and … all get linked together to create MyProgram.exe (or just MyProgram on linux systems)

Class Diagrams (4)

- All together:



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.

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
```

Last time, I asked you to review some example code that uses pointers to C++ classes… why important? questions?

```
// EXAMPLE 6:
// This strategy of working with pointers to a base class is especially
// useful when storing a long list of Ducks.  We'll use the C++ std::vector
// class to create a list of pointers to Ducks.

std::vector<Duck*> duckList;

Duck *duck1 = NULL;
duck1 = new MallardDuck();
duck1->setName("Mallard Junior");
duckList.push_back(duck1);
```

```
// If we want, we can skip the first step of setting the pointer to NULL
// and write the code more compactly like this:
Duck *duck2 = new RubberDuck();
duck2->setName("Squeaky");
duckList.push_back(duck2);

Duck *duck3 = new DecoyDuck();
duck3->setName("Mr. Quiet");
duckList.push_back(duck3);

// Now, we can work with the whole list of ducks very easily
for (int i=0; i<duckList.size(); i++) {
    duckList[i]->fly();
    duckList[i]->performQuack();
}
```

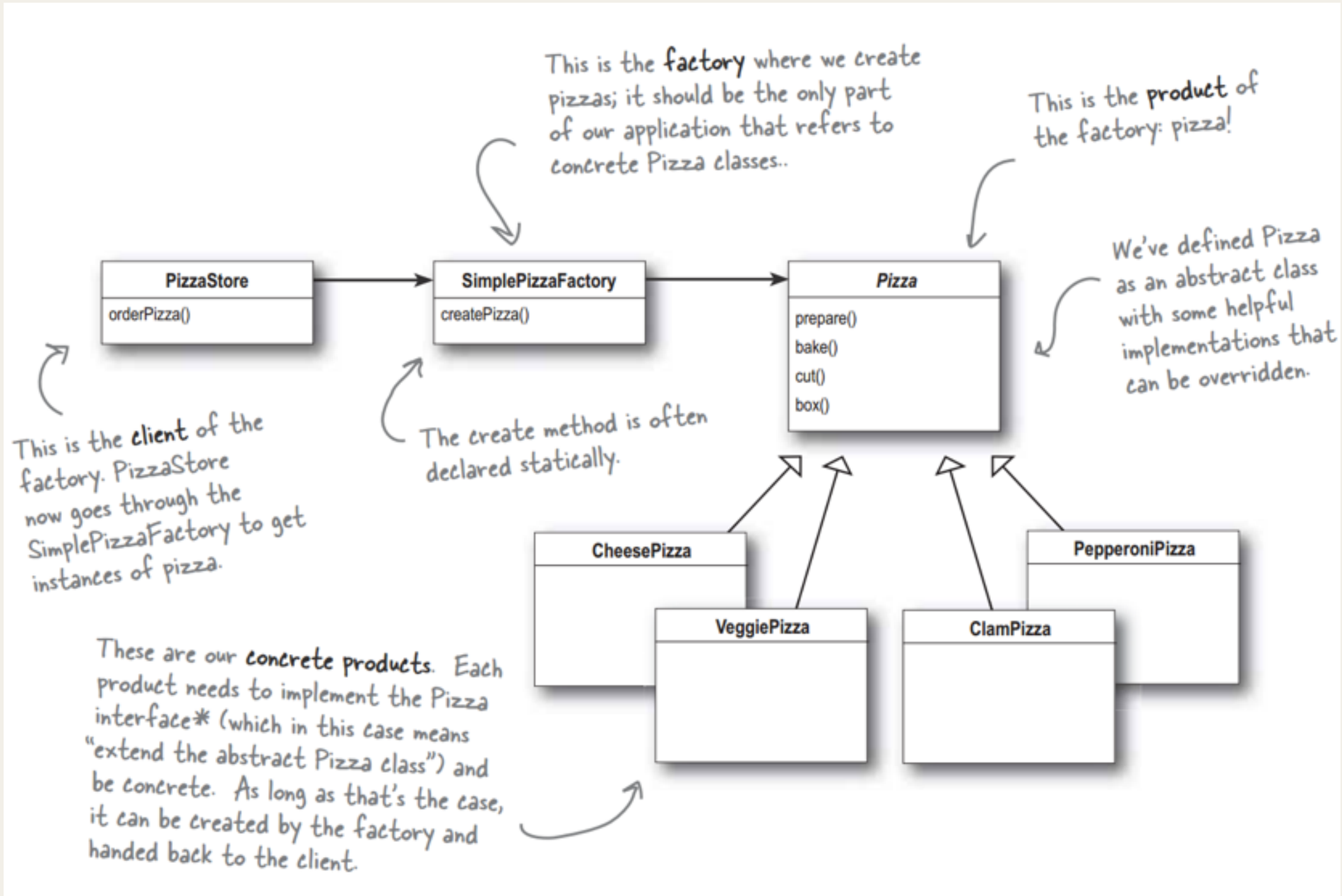
```
int size = 100;
int *ps = 0;
int size2 = 0;
int *ps2 = 0;
ps = &size;
ps2 = &size2;
size2 = *ps;
if (size == size2)
  cout << "size == size2" << endl;
else
  cout << "size != size2" << endl;
if (ps == ps2)
  cout << "ps == ps2" << endl;
else
  cout << "ps != ps2" << endl;
if (*ps == *ps2)
  cout << "*ps == *ps2" << endl;
else
  cout << "*ps != *ps2" << endl;
```

What does this print out?

The Strategy Pattern

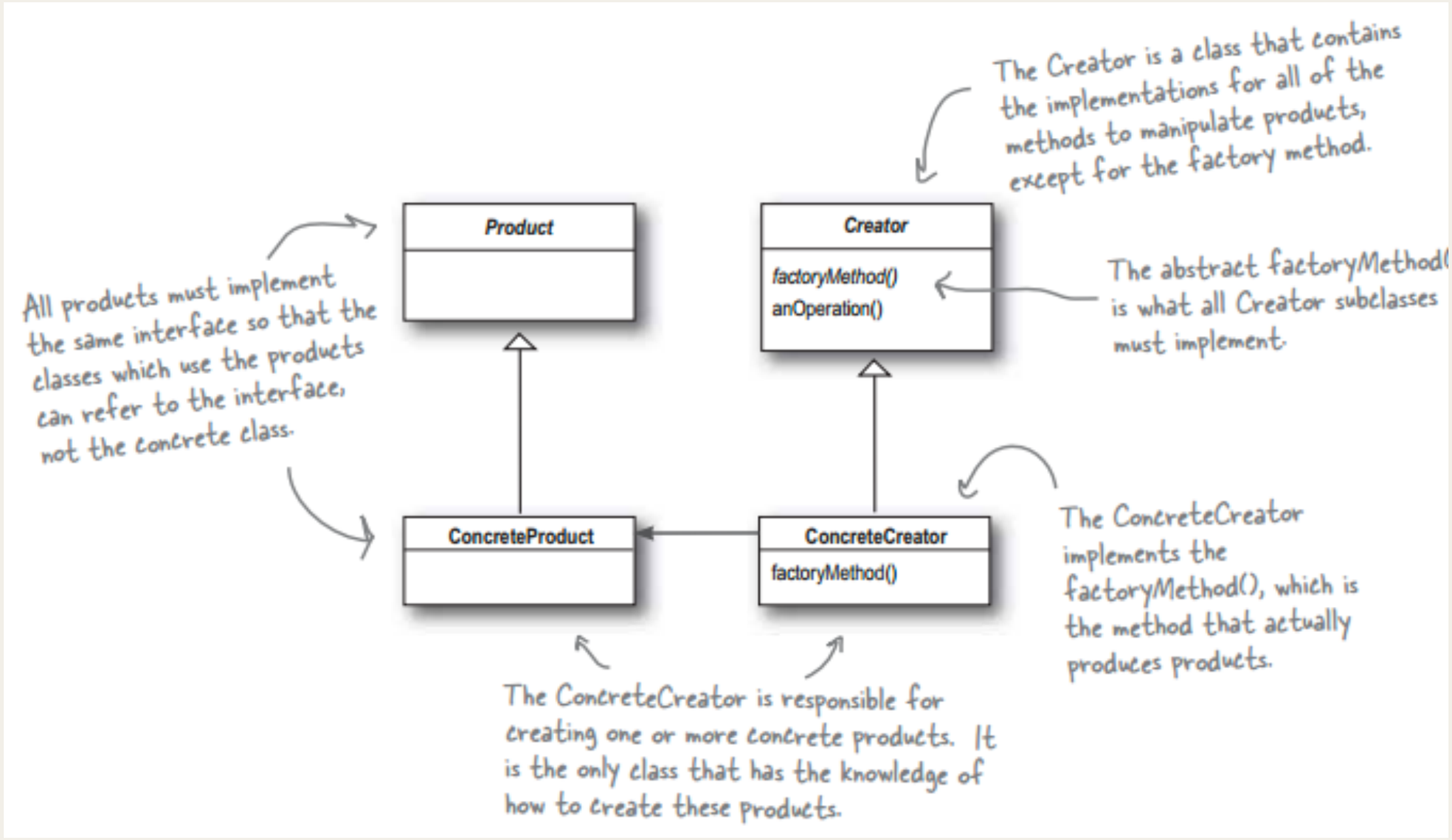
- **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.

SimplePizzaFactory

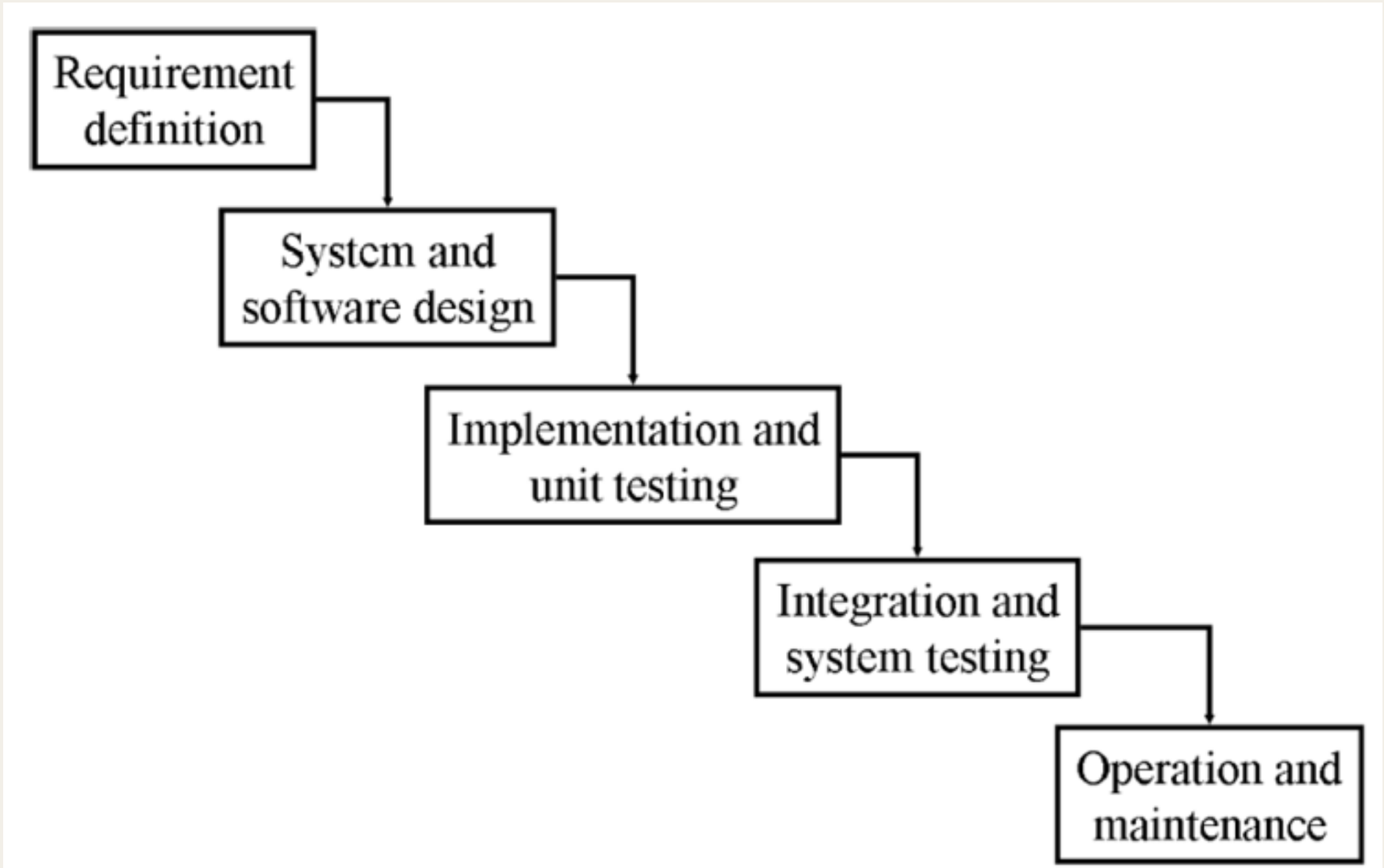


The Factory Method Pattern

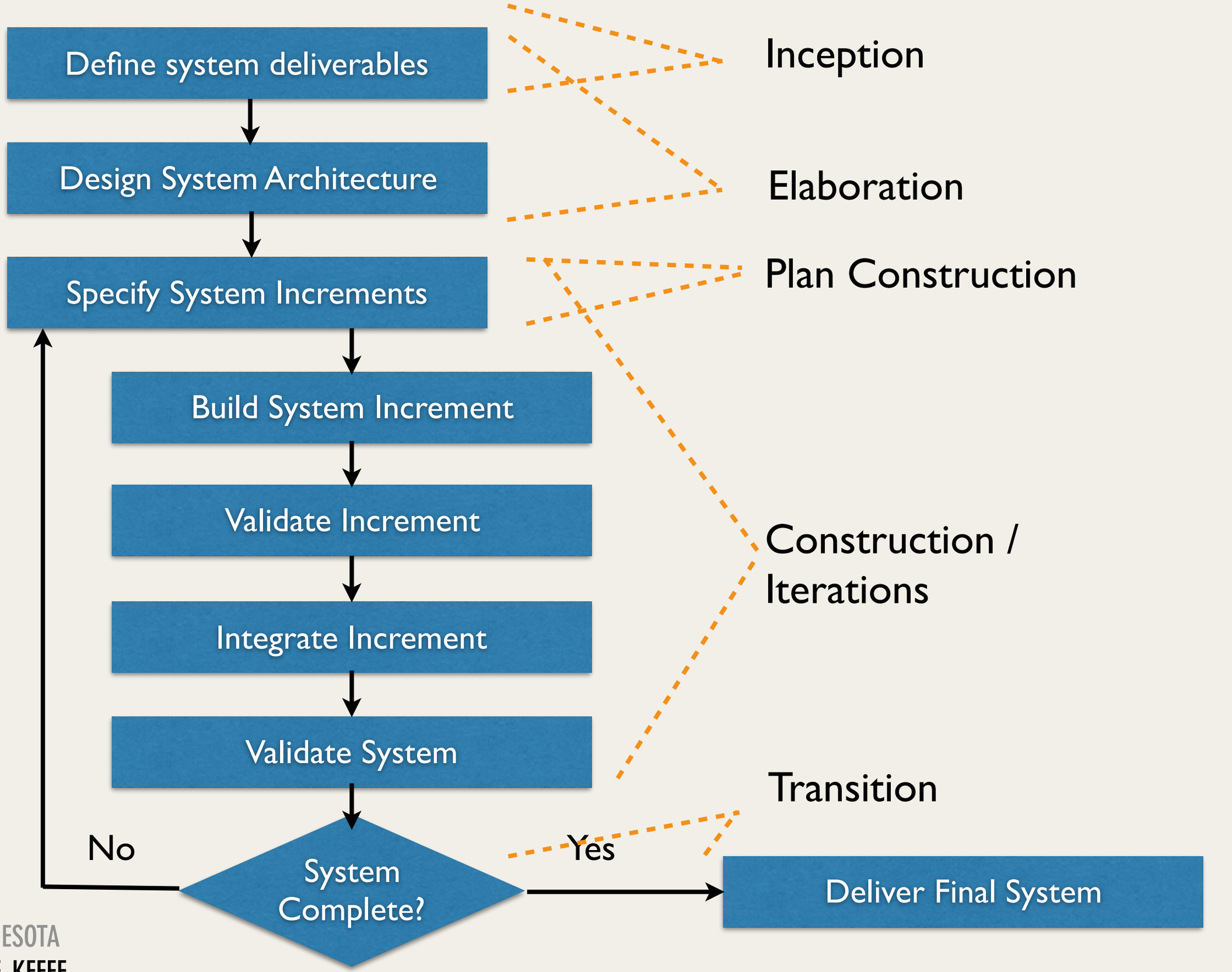
- **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate.



The Waterfall Model

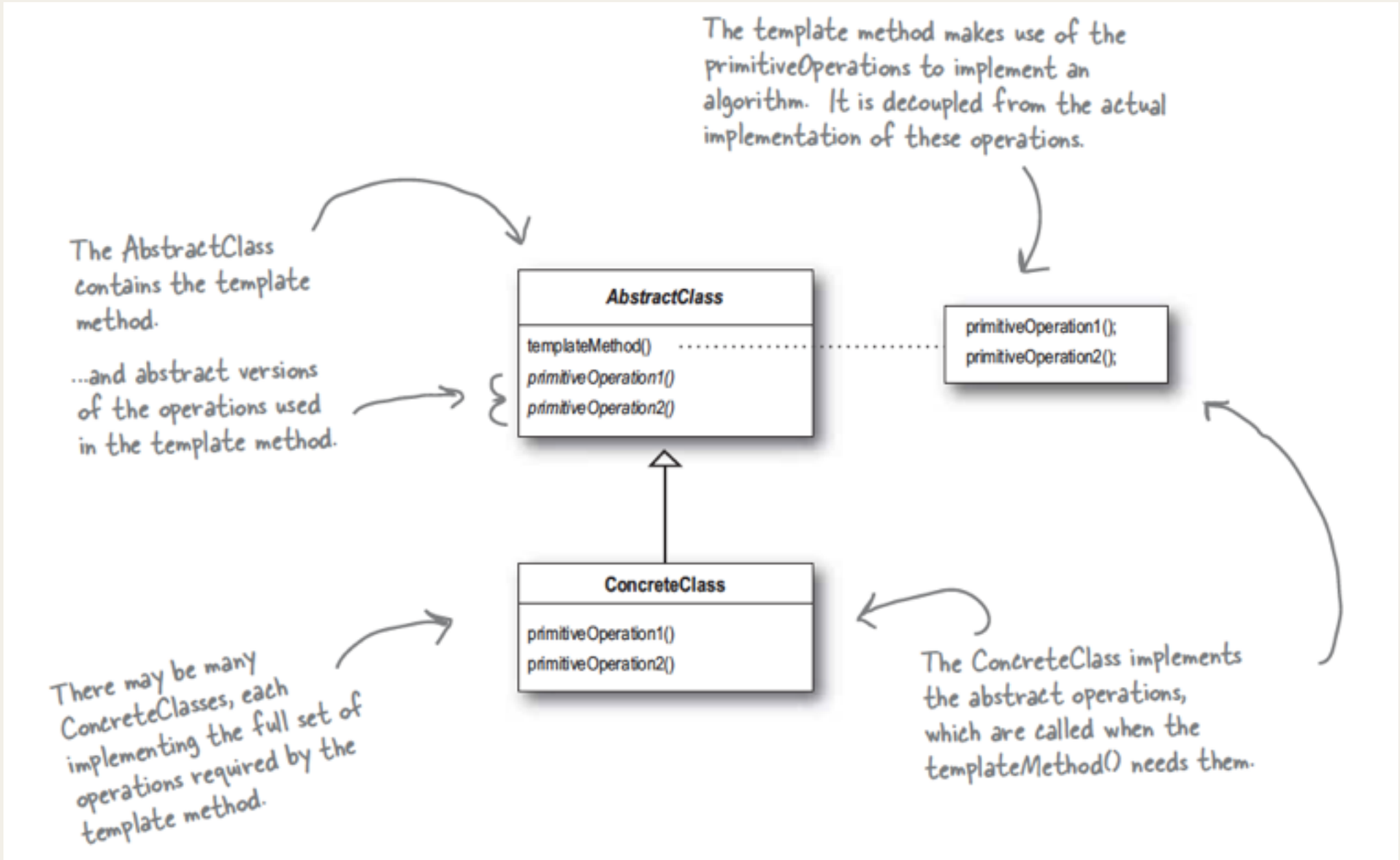


Iterative Development



Template Method Pattern

- **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses.



Poor Documentation from Bad Programming Style

```
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 1; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

Documentation without Comments

```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```