CSC301

Creational design patterns: Factory Method, Builder, Singleton

Logistics

- A lot happening until the end of the term:
  - Team deliverable due on Wednesday, Nov 16
  - Term test on Monday, Nov 21.
  - Individual final demo, the week of Nov 28
    - Reminder: Code freeze on Monday, Nov 28, at 10 am.
  - Team demos, the week of Dec 5

Individual Final Demo Registration

- Registration starts Tuesday, Nov 15, at 1:30 pm
  - Add your GitHub username to this spreadsheet
  - No typos, no extra spaces, names are case-sensitive
  - GitHub username (not UTorId, not student number)
  - One time slot per student
- Registration **ends on Friday morning, Nov 18, at 10 am**
  - If you don't register by that time, you will lose 3% (out of 15%) of the mark. No exceptions!

Construction vs. *Use*

- When designing a system, we distinguish between *construction* and *use*.
  - Construction = Creating an instance of some concrete class.
  - Use = Calling methods on an object
    - Do not need to know the object's concrete class,
    - Only need to know that implements certain interface(s).

- The distinction between *construction* and *use* (aka maintenance) happens in construction engineering as well:
  - Construction site is very different than a building full of tenants.
  - Different needs & priorities, different rules, different services needed, etc.

Construction vs. *Use*

- Generally good design (especially for large systems):
  - Most of the application is written in terms of interfaces.
  - Introduce objects whose sole responsibility is constructing other objects.
    - These are the only objects that depend on concrete implementations.
  - Make the *boundary* between construction and use clear.
    - Ensure it is easy to swap in different implementations

- Related design pattern: Dependency injection
  - *Warning:* If you're using *dependency injection* for small project, you might be over-engineering.

Designing Interfaces

- Why is it good to design your application in terms of interfaces?
  - Allows you to focus on the domain problem, and ignore implementation details
  - Can use different implementations as gradual improvements
  - Can use different implementations to support different integrations
    Ex: Different sign-in providers such as Google, Facebook, GitHub, etc.
  - Makes it easy to test the application logic, because we can test doubles as implementations

Stating The Problem

- An application needs to create instances of type T.
  - T is an interface (or an abstract class)
  - The application *must not* depend on any specific implementation of T.
    (i.e. it must NOT construct objects of type T by itself)
  - The application depends only on T, and work with any implementation of T.

Why Is It a Problem?

- What’s wrong with depending on a specific implementation?
  - The specific implementation might not exist,
  - you might want to replace it in the future,
  - It might not be the one you want to use for testing,
  - When the specific implementation changes, need to recompile the whole application.
- In other words … Lack of flexibility.

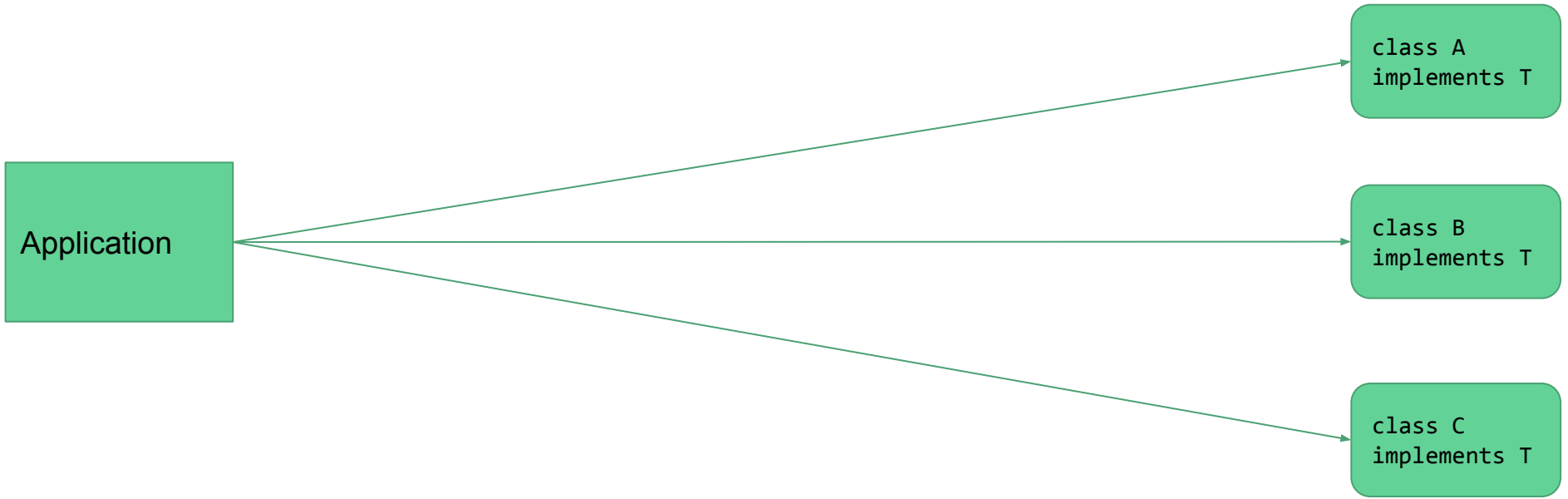
Solution

- Intuition: Application delegates the responsibility (of creating objects of type T) to someone else.
    - Let “someone else” invoke constructors (of various implementations of T).
    - Depend on that “someone else”, instead of any specific implementation of T.
    - For flexibility, the “someone else” should be an interface (and not a concrete class).

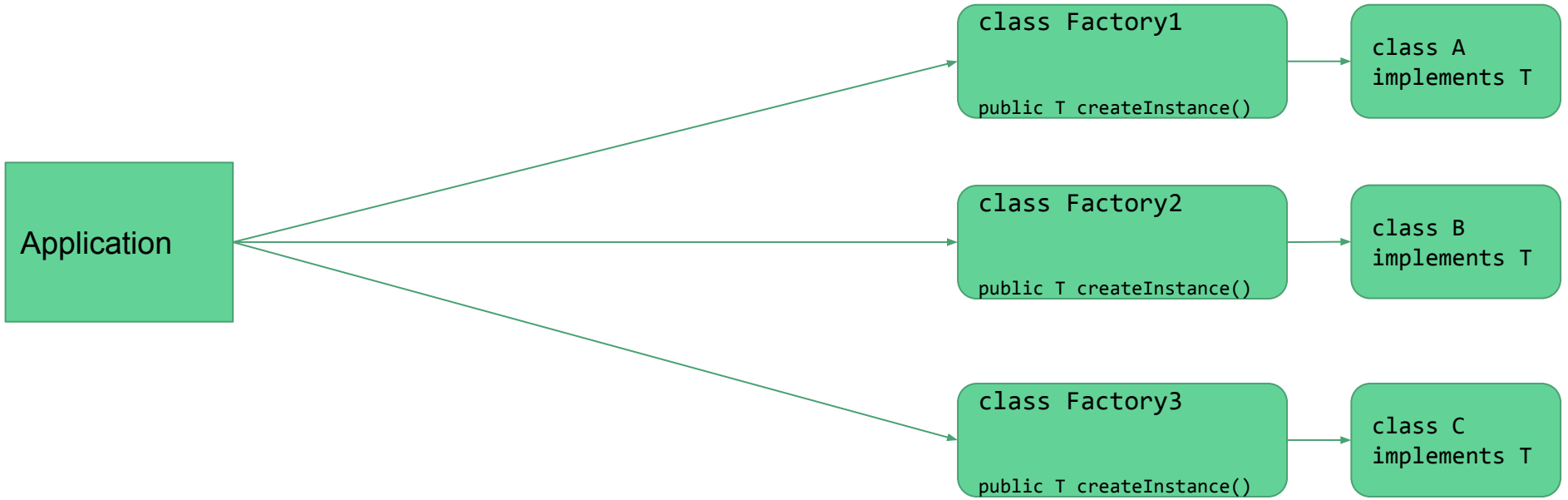
The *Factory Method* design pattern is one way to implement our intuition.

Let’s see some diagrams describing the *Factory Method* pattern ...

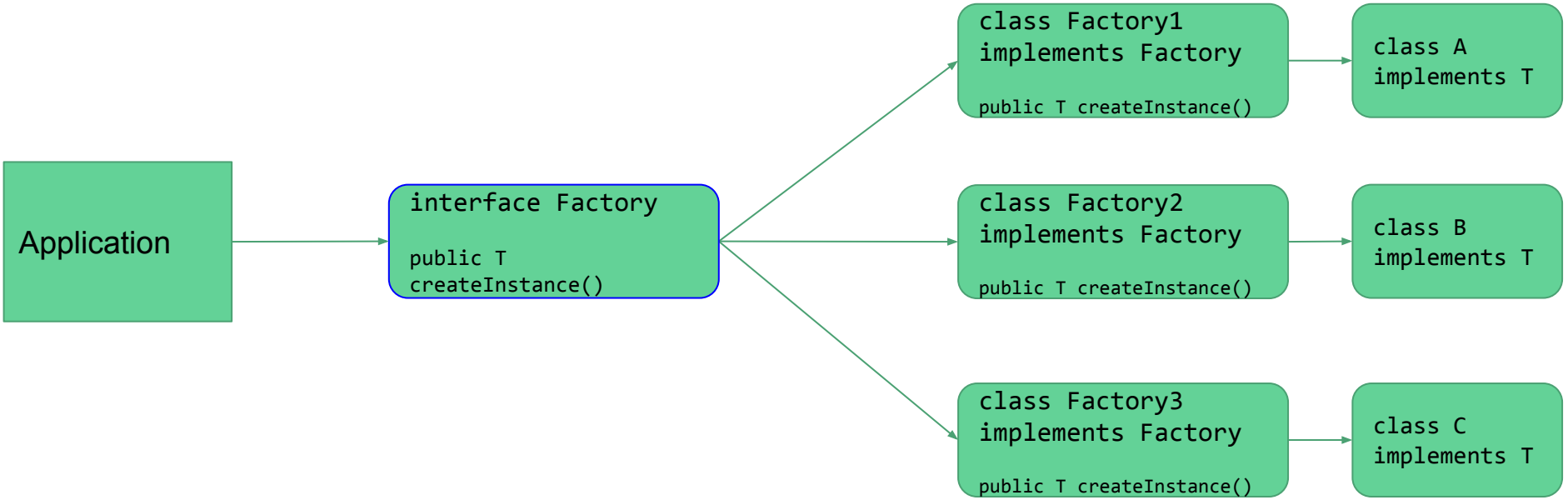
Step 1: The application invokes the constructors of different implementations of T.



Step 2: Application depends on factory objects to constructs instances of type T. Factory objects invoke the constructors of different implementations of T.

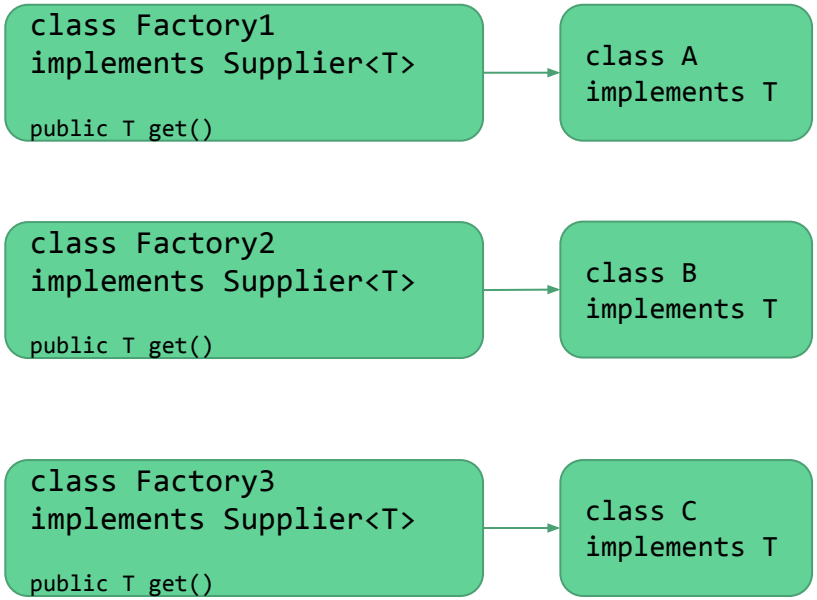


Step 3: Application depends only on the factory interface. Any factory implementation can be used to constructs instances of type T.

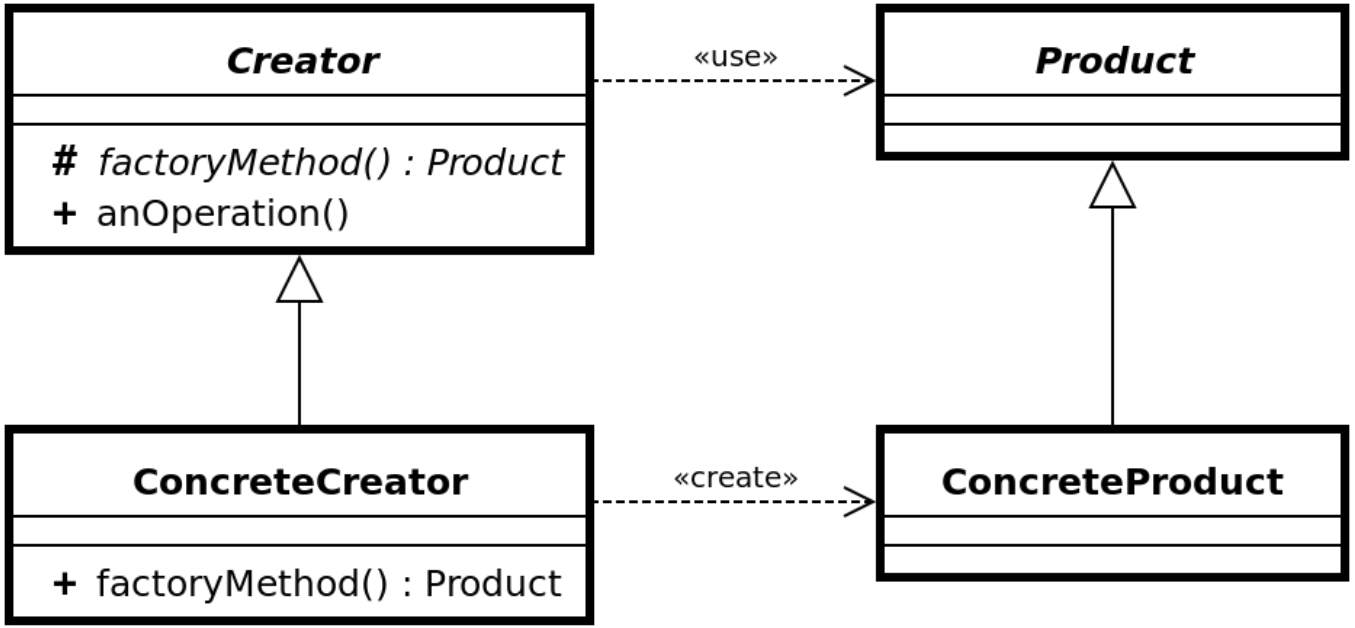


We can simplify the codebase using functional interfaces.

Instead of defining defining the `Factory` interface (which has a single method that takes no arguments and returns an object of type T), use `Supplier<T>`.



Or, if you prefer UML ...



The *Factory Method* Design Pattern

- Delegate responsibility
  - Don’t construct objects of type T yourself!
  - Ask a factory object to do it for you.
- Make the factory an interface
  - With a single method that returns T.
  - Different implementations of the factory interface return different implementations of T.

Factory Method

Let’s see a code example ...

Abstract Factory

- Related design pattern: *Abstract Factory*
  - Very Similar to *Factory Method*, except that the factory interface defines multiple methods.
- Instead of creating just one (type of) object, we define an interface for creating “families of related or dependent objects”.
  - Can be implemented using Factory Methods (but there are other options as well).

Factory Method vs. *Abstract Factory*

- Difference in implementation - Inheritance vs. Composition
    - When implementing a *Factory Method* interface, the implementing class **is a** factory. (inheritance)
    - When implementing an *Abstract Factory* interface, the implementing class **has a** factory. (composition)

Let’s see another creational design pattern that solves a slightly different problem ...

Telescoping Constructor

- Telescoping Constructor
  - A constructor with many arguments
  - Possibly of the same type
- Ex:

```
new CanadianAddress("Apt 1A",
            "301", "College", "St.",
            "Toronto", "Ontario",
            "B4D C4T");
```

Telescoping Constructor

- Error prone
  - Correctness might depend on argument order
    - Developers might get it wrong
      (Or, waste time double checking)
    - Compiler cannot save you
- Harder to test
  - More arguments ⇒ Even more combinations to test
- Results in ugly code

The Builder Pattern

- Break construction into two separate tasks:
  - Collecting arguments
  - Creating the instance
- Solve the telescopic constructor problem by collecting the arguments one at a time
- Improve code quality

Builder

Let’s see a code example …

Builder, Argument Validation

- Perform simple validation when collecting arguments
  - Null, empty string, negative number, etc.
- Perform (semantic) validation in `build()`
  - Conditions that involve multiple arguments
  - Ex: Conflicting postal code and province

Singleton

- Another creational design pattern
  - Ensures that only one instance of a certain class can be created
- Ex:

```
public class Foo {
      private static Foo instance = null;

      public static Foo getInstance(){
            if(instance == null){
                  instance = new Foo();
            }
            return instance;
      }
}
```

Note: Implementation is slightly more complex for multi-threaded programs.

Singleton

- Controversial design-pattern
  - Feels like a global
  - Considered an anti-pattern by some people
- Think twice before using it
  - There are cases where singleton is the right solution, but they are fairly rare

After the break ...

Term test review session

Term Test

- Monday, Nov 21, in class
  - For those of you who haven't been to a lecture yet, that's BA1200
  - Due to room capacity restrictions, you must attend your lecture section (i.e. the one you are officially enrolled in)
- Test duration: 100 minutes
  - We'll start 15 minutes after the hour (that avoid disruptions from late students)
  - We'll finish 5 minutes before the hour (so we can pick up the papers in an orderly fashion)

Term Test - Material

- Git / GitHub
  - Basic Git usage: Clone, pull, push, add, commit, branching and merging
  - Basic GitHub concepts: Fork, pull-request
  - If you've been working throughout the semester, you shouldn't need much preparation.
- Coding & Software design
  - All design patterns (iterator, observer/observable, adapter, builder, factory method, etc.)
  - General concepts (serialization, composition vs. inheritance, lambda expressions, lazy-evaluation, programming to interfaces, etc.)
  - Writing testing code

Past Tests

- There are two past tests (from last year) available
  - Fall 2015
  - Winter 2016
- Notice that questions can be specified as a mix of:
  - English description
  - Interfaces
  - Unit tests
  - Main method (with comments indicating the expected output)
- Make sure you can read code
  - One of the things we want to test is your ability to *read and understand* code