2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Open-Closed Principle (OCP)
- Singleton design pattern
- Factory Method design pattern
- Template Method design pattern

SOLID Object-Oriented Design Principles

- **S**ingle Responsibility Principle (SRP, see Lecture 2)
- **O**pen Closed Principle (OCP, treated in this lecture)
- **L**iskov Substitution Principle (LSP, see Lecture 4)
- **I**nterface Segregation Principle (ISP, treated later)
- **D**ependency Inversion Principle (DIP, see Lecture 8)

The Two Ways of Using a Class

1. By regular client code:
   - Instantiate **`new`** objects, and call methods on them
   - Contracts usually specify this interface
   - Client code can access **`public`** members only
2. By code in subclass that extends the class:
   - Access **`super`** functionality
   - Contracts for this use are harder to specify (usually not done)
     E.g., `execute()` in abstract command of `CommandPattern`
   - Subclass code can access **`protected`** members

Open Closed Principle

- Modules should be open for extension (via inheritance)

  Inheritance allows reuse of implementation:

  - representation (instance variables)
  - operation definitions (method bodies)

  Can add instance variables/methods, and override methods *without modifying the superclass*

- Modules should be closed for modification (by clients)

  `private`, or at least **`protected`**, instance variables

  Do not modify code in order to add or adapt functionality

Taxonomy of Design Patterns

- Creational patterns

  Factory Method, Singleton

- Structural patterns

  Composite, Façade, Adapter, Decorator

- Behavioral patterns

  Strategy, Iterator, Observer, Command, State, Template Method

- Concurrency patterns

  "`SwingWorker`"

Singleton Pattern Motivation

- One Controller
- One Undo-Redo facility
- One Logger

Concern: How to prevent creation of multiple instances?

Singleton Anti-Patterns

- Ignore the issue; rely on client code to create only one instance in a global variable
- Make all instance variables and methods in the class **`static`**

Singleton Design Pattern (adapted from Eddie Burris)

Intent

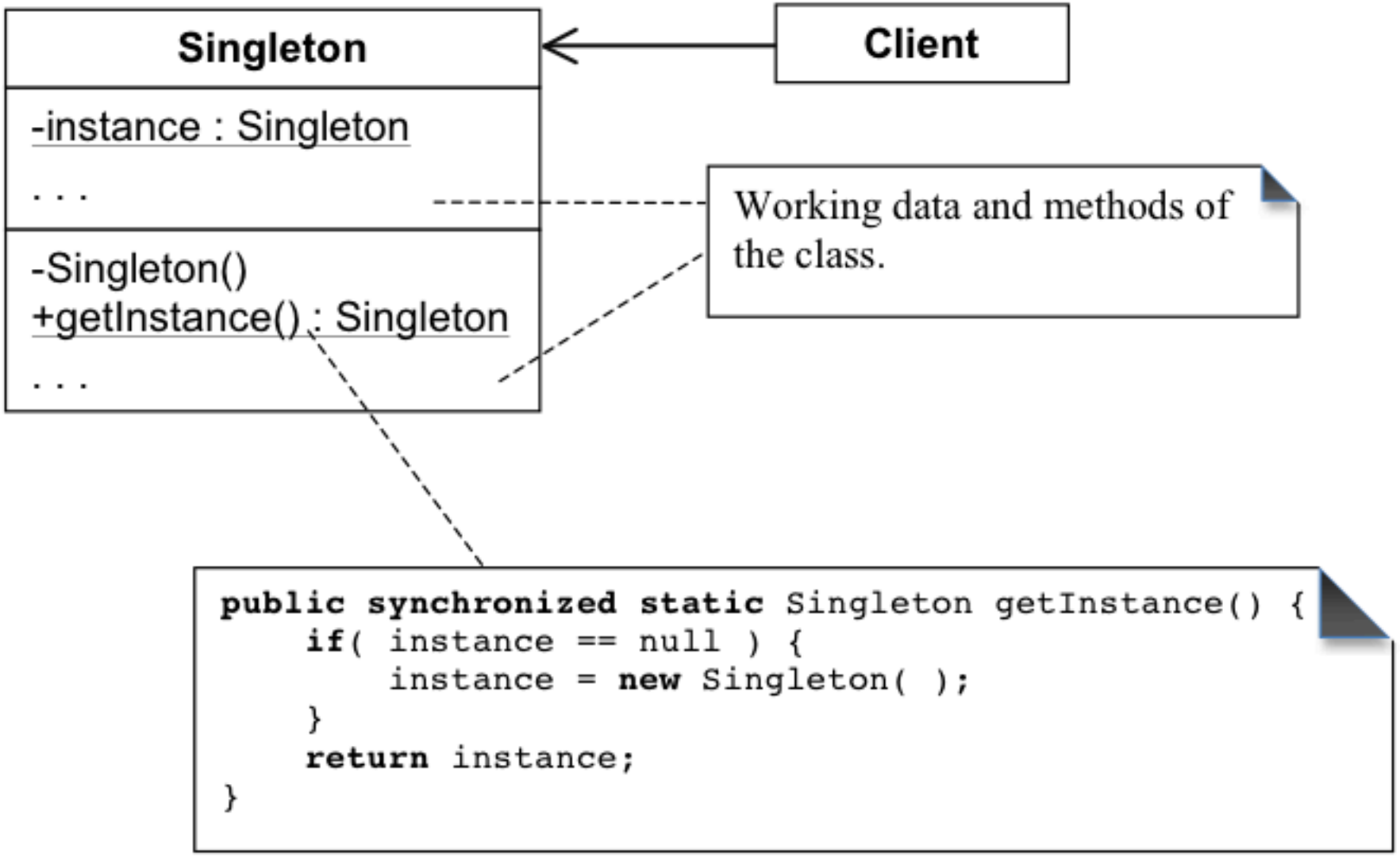
The Singleton design pattern

- ensures that *not more than one instance* of a class is created;
- it provides a *global point of access* to this instance.

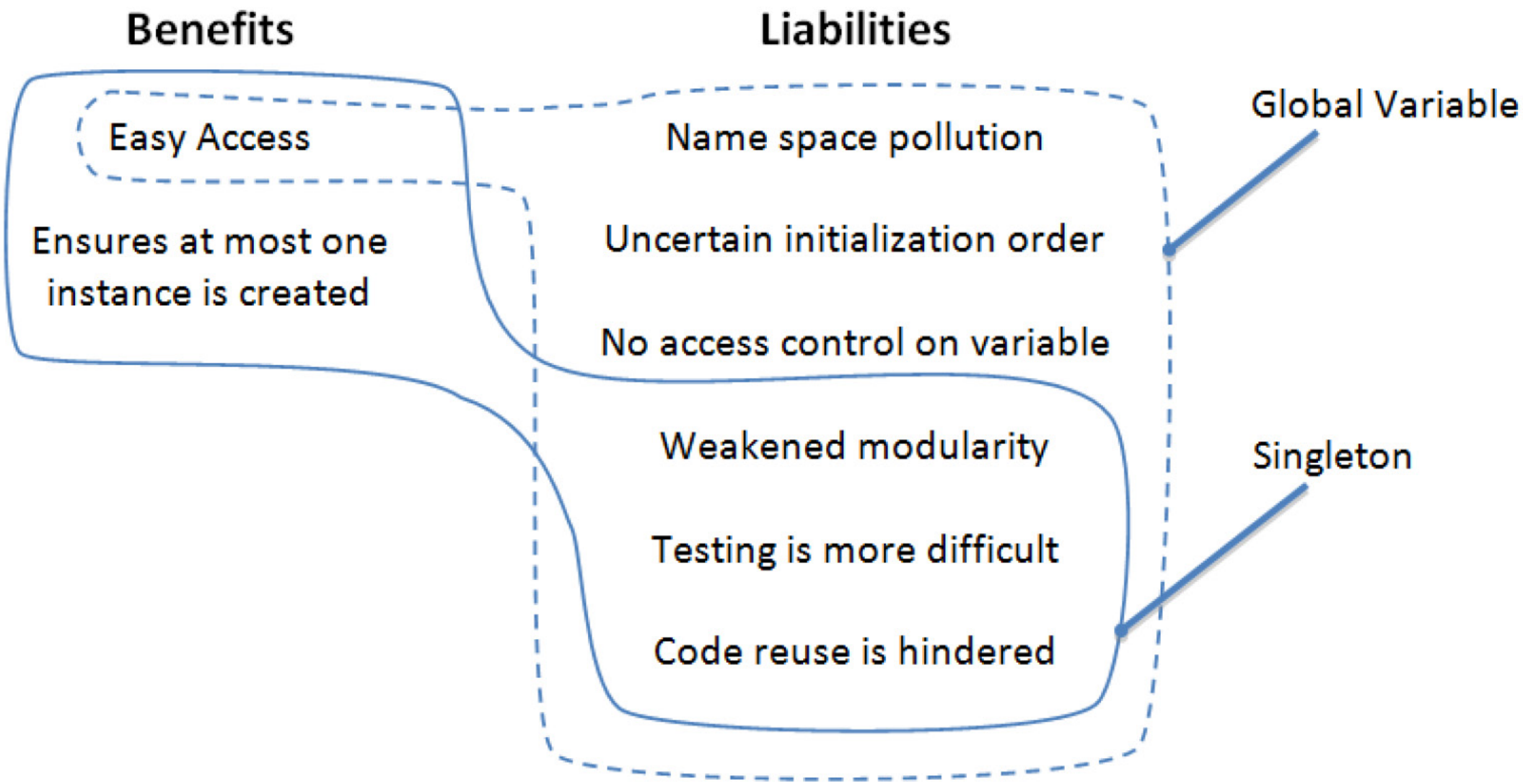
Singleton Pattern Solution (adapted from Burris)

- Make the constructor of the class **`private`**
- to prevent clients from creating instances [. . .] directly.
- Add a **`public static`** method `getInstance()`
- that returns an instance of the class.
- The first time `getInstance()` is called
- an instance of the class is created, cached, and returned.
- On subsequent calls, the cached instance is returned.

Singleton Pattern Class Diagram (from Burris)



Singleton Pattern Benefits & Liabilities (from Burris)



Singleton Pattern Concerns

- Concurrency in multithreaded applications: **synchronized**

```
1     /* This factory method is not thread-safe */
2     public static Singleton getInstance() {
3         if ( instance == null ) {
4             instance = new Singleton();
5         }
6         return instance;
7     }
```

- Many classes depending on the Singleton (its global nature)

How to Avoid Dependency on Globals

- Global constants, global variables, global singleton objects

  Other modules can depend directly on these.

  Such dependence hinders testing and reuse.

- To avoid, apply Strategy pattern and invert dependencies.

  Make other modules depend on an (abstract) interface;
  hence, they do not depend on the concrete globals.

  Supply actual concrete globals as parameter to other modules.

Factory Method Pattern Motivation

- **new** statement requires constructor of a *concrete* class

```
Set<Node> visited = new HashSet<>();
```

- This makes client code depend on the concrete class

  (There is an exception, where **new** is somewhat less harmful)

- Makes it harder to test, e.g., with another/simpler concrete class
- Makes it harder to reuse
- Violates the Dependency Inversion Principle (DIP)
- Program to an interface (abstraction), not to an implementation

Concern: How to decouple creation from the concrete class?

When `new` Does Not Create a Dependency on Implementation

Anonymous inner class:

```
1         FunctionalityA.doA(n, new CallbackB() {
2             int count;
3             public void doB(String data) {
4                 ++ count;
5                 System.out.println(count);
6             }
7         });
```

There is no dependence on an external class.

There is dependence on the chosen implementation in inner class.

Factory Method Anti-Patterns

- Ignore the issue; just use **`new`**

Factory Method Design Pattern (adapted from Eddie Burris)

Intent

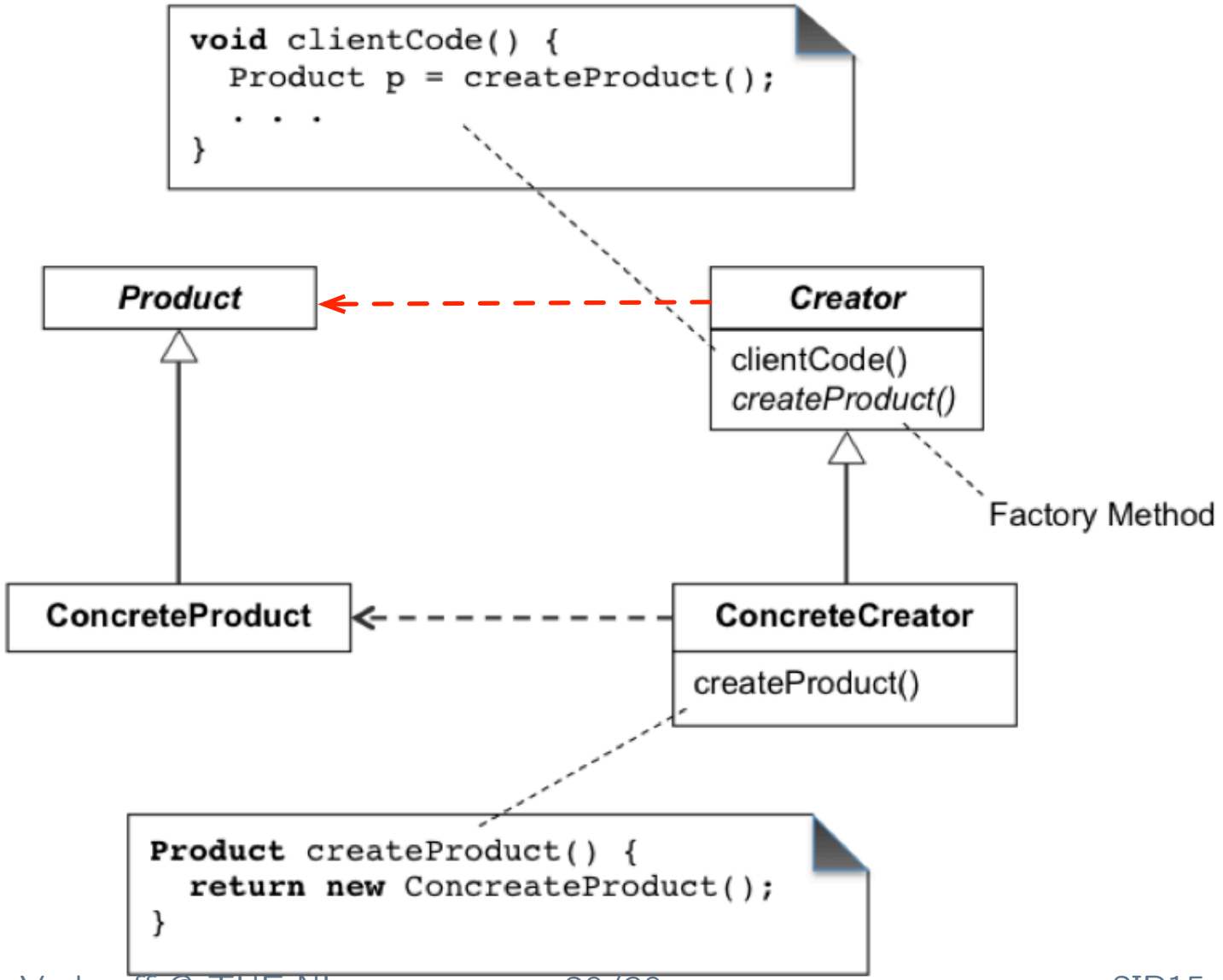
The Factory Method design pattern [quoting the Gang of Four]

- "defines an interface for creating an object,
- but lets subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses."

Design Principle: Encapsulate What Varies

- Identify the aspects of your application that vary, and
- separate them from what stays the same.

Factory Method Pattern (from Burris; added dependency)



Why the Factory Method Works

- Because of how the method definition is found for a method call
- When a method is called on an object,

  its definition is searched by *ascending* the inheritance hierarchy

  starting from the (run-time) class type of the object:

  The first (nearest) definition found is used
- In single-inheritance languages (Java), search path is unique

  Multiple-inheritance languages (like C++) may be problematic

Why the Factory Method Works: Example

- Given object `ConcreteCreator c;` consider call `c.clientCode()`
- `ConcreteCreator` does not define `clientCode()`
- Definition of `clientCode()` in found in superclass `Creator`
- `clientCode()`, in turn, calls `createProduct()`
- Definition of `createProduct()` is found in `ConcreteCreator`

Factory Method Alternative

- Consider object creation a Strategy (cf. Strategy pattern)
- Put abstract creator/factory in an abstract class or interface
- Give class constructor a parameter with abstract factory as type
- Client code passes concrete factory into the class

Template Method Pattern Motivation

- Some code fragments may resemble each other, without being duplicates
- Resemblance is in overall structure
- Difference is in some steps

Concern: How to avoid code duplication?

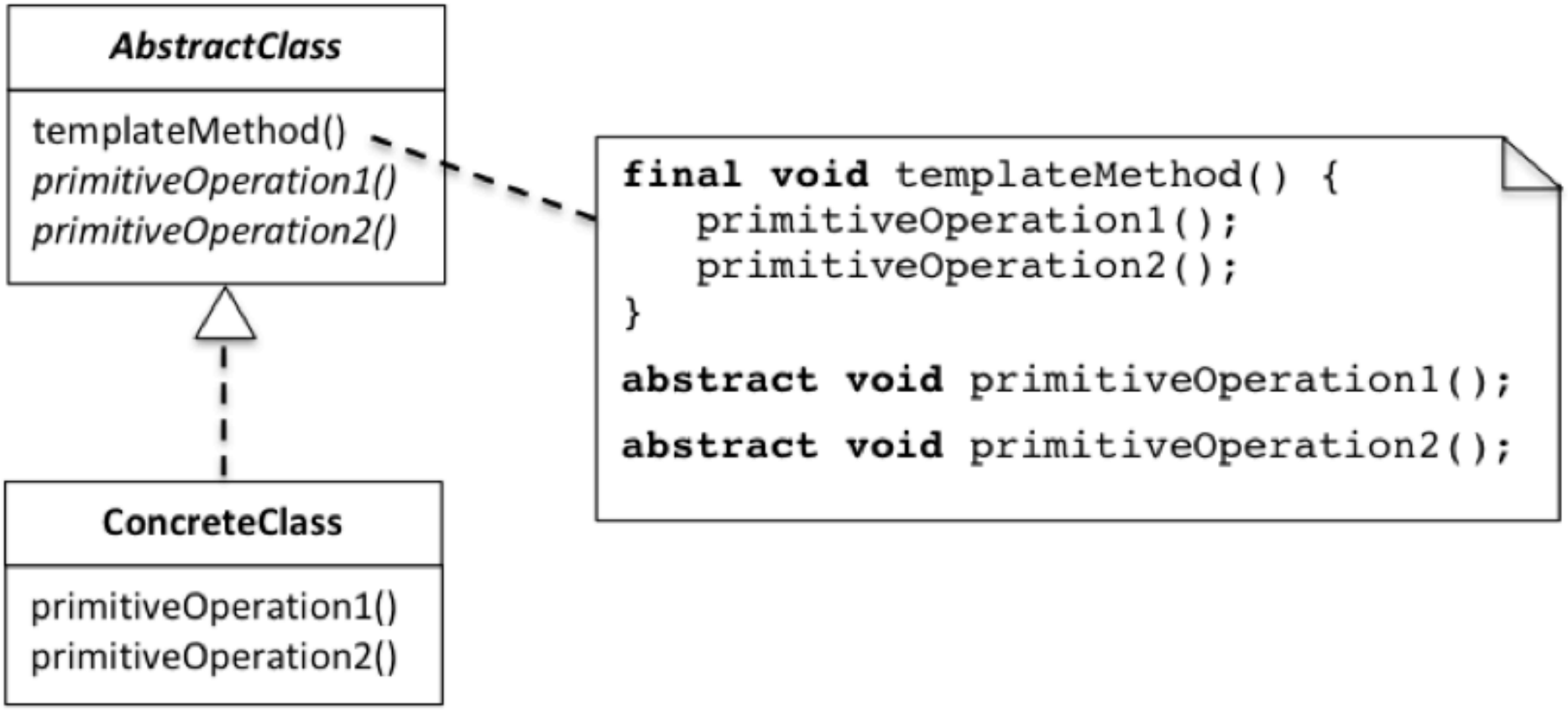
Template Method Design Pattern (adapted from Eddie Burris)

Intent

With the Template Method design pattern

- the structure of an algorithm is represented once
- with variations on the algorithm implemented by subclasses.
- The skeleton of the algorithm is declared in a template method
- in terms of overridable operations.
- Subclasses [can] extend or replace some or all of these operations.

Template Method Pattern (from Burris)



Template Method Alternative

- Consider each step a Strategy (cf. Strategy pattern)
- Put abstract steps in an abstract class or interface
- Give class constructor parameters with abstract steps as type
- Client code passes concrete steps into the class

Template Method Pattern versus Factory Method Pattern

Factory Method pattern is special case of Template Method pattern

- Factory Method encapsulates object creation.
- Factory Method is a step in terms of Template Method pattern.
- Main functionality, which calls Factory Method, is Template Method.

Template Method Pattern versus Strategy Pattern

Both allow variation in choice of algorithm, or algorithmic steps

- Template Method allows subclass to vary steps via overriding.
- Strategy allows client to choose algorithm via polymorphism.

Summary

- Open-Closed Principle (OCP)
- Singleton design pattern
- Factory Method design pattern
- Template Method design pattern