2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Data Decomposition
- Java `class` mechanism, more advanced features:
  - Mutable versus immutable; variables versus objects
  - Polymorphism
  - Overriding methods
- More design patterns:
  - Template Method Design Pattern
  - Factory Method Pattern

Procedural Abstraction versus Data Abstraction

- A procedure/function groups a bunch of *executable statements*.
  - Gives the group a *name*, by which it can be *reused* (abstractly).
  - Possibly has *parameters*, to alter its internal *behavior*.
  - When containing many statements, it should be *decomposed*: functional decomposition
- A record/struct/tuple groups a bunch of *variables* (data items).
  - Gives the group a *name*, by which it can be *reused* (abstractly).
  - Possibly has *parameters*, to alter its internal *structure*.
  - When containing many variables, it should be *decomposed*: data decomposition

In Java, we (have to) use methods and classes for this.

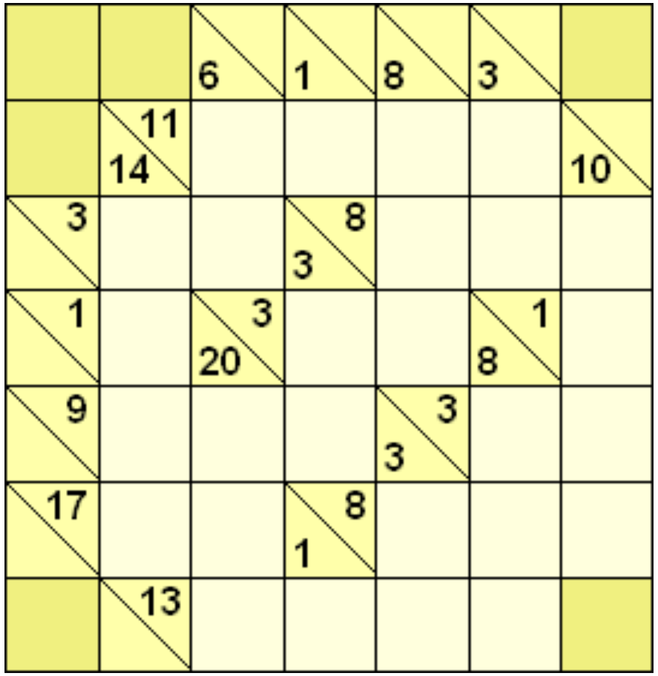
Data Decomposition

- Decide which data items to put in what module (class)
- Deicide which operations on data to put in what module (class)
- Minimize coupling/dependency
- Maximize cohesion

Data Decomposition: Example

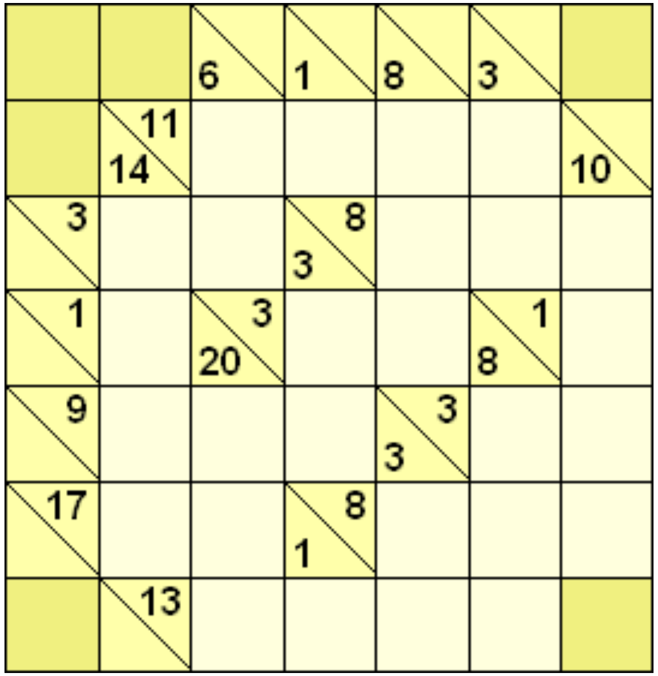
State of a *Kakuro* puzzle, being solved:

- Can be expressed as one bunch of variables of primitive types.



Data Decomposition: Example

State of a *Kakuro* puzzle, being solved:



- Can be expressed as one bunch of variables of primitive types.
- Decompose:
  - Grid consisting of cells
  - Cell is either blocked, empty, or holds a digit
  - List of clues
  - Clue gives sum for a group of cells

Data Abstraction in Java: References, Sharing, Aliasing

- A variable of a primitive type has as value a value of that type.
- A variable of **`class`** type T does *not* have an object of type T as value, but
  - either *the* **name** *of an object of type* T,
  - or the special value **`null`**.

  This name is a *unique identifier* of that object (a.k.a. *reference*).
- This allows *sharing*, a.k.a. *aliasing*: two distinct variables name the same object. It can help improve (memory and time) efficiency by avoiding the copying of data. But it complicates reasoning.
- Operator "==" on expressions of a class type compares object *names*, and not the object states: `"abc"!= "abc"`

Data Abstraction in Java: Aliasing Example

```java
Card u = new Card(Rank.Ace, Suit.Spades);
Card v = new Card(Rank.Ace, Suit.Spades);
Card w = new Card(Rank.Queen, Suit.Hearts);
Card y = u;

// u, v, and w name (refer to) distinct objects:
//   u != v  &&  v != w  &&  w != u

// u and v refer to objects having the same value (state)

// u and w refer to objects having different values

// u and y name the same object (aliasing): u == y
```

NEW TOPIC

Changeability of variables versus objects

Note the difference between (changing)

- the state of a variable
  (i.e., its primitive value or the name of an object it referes to) and
- the state of an object.

These can change independently:

- A variable can be made to refer to another object (unless it is **`final`**), without changing the state of the objects involved.
- The state of an object can be changed (unless its class is immutable), without changing the variables that refer to it.

Data Abstraction: Mutability

- A data type T implemented as a class is said to be immutable when none of its methods **modifies `this`** or a parameter of type T. I.e., the state of a T object cannot change after construction.

  N.B. A T method could still modify a parameter of another type.

  E.g.: `String`; `immutable/Fraction`

  Immutable types need various/parameterized constructors, factory methods, or producers.

- It is said to be mutable otherwise.

  E.g.: `StringBuilder` vs `StringBuffer`; `mutable/Fraction`

In Java, arrays are partly immutable (length), partly mutable (items)

Mutable versus Immutable

	Immutable	**Mutable**
Efficiency	$-$	$+$
Reasoning	$++$	$--$
`equals()`	$--$	$++$
`clone()`	$++^*$	$--$

Concerns: Sharing, (partial) copying of data

*Immutable classes never need to `clone()`.

Data Abstraction: equality and similarity

Two objects are —at the time of comparison— said to be

- equal when they are *behaviorally indistinguishable*:
  - every sequence of operations (queries and commands), when applied to both objects, yields the same result.
  - `a.equals(b)`; e.g. `"abc".equals("abc")`
- similar when they are *observationally indistinguishable*:
  - every sequence of *queries* only (no commands), when applied to both objects, yields the same result.
  - `a.similar(b)`

Data Abstraction: Comparison of (im)mutable objects

- In general: `a == b` implies `a.equals(b)`, but not vice versa.
- Mutable objects are equal when they are the same object (==):
  - Otherwise, you can change one without changing the other.
  - `a.equals(b)` inherited from `Object` (returning `a == b`) suffices
- In general: `a.equals(b)` implies `a.similar(b)`, but not vice versa.
- Immutable objects are equal when they are similar (same state):
  - Immutable types must override implementation of `equals()` to coincide with `similar()`.

Data Abstraction: `equals()`

- `equals()` must be

  reflexive: `a.equals(a)`

  symmetric: `a.equals(b) == b.equals(a)`

  transitive: if `a.equals(b) && b.equals(c)` then `a.equals(c)`

- A class can have several (overloaded) `equals()` methods.

Data Abstraction: `clone()` to create a copy

- To offer method `clone()`, a class must implement `Cloneable`; otherwise, `clone()` will throw `CloneNotSupportedException`.
- `clone()` returns a copy of **`this`**.
- The copy should be similar but not identical to **`this`**.
- The default implementation inherited from `Object` constructs a new object and copies all instance variables (shallow copy).
- This is sufficient for immutable objects.
- Mutable objects should implement their own `clone()` (doing a deep copy)
- However, it is better to avoid `clone`.

Alternatives to `clone()`

- http://en.wikipedia.org/wiki/Clone_(Java_method)
- http://www.javapractices.com/topic/TopicAction.do?Id=71
- http://www.xenoveritas.org/blog/xeno/java_copy_constructors_and_clone
- http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html
- Copy constructor, e.g. for mutable fractions:

```java
    /**
     * Constructs (a copy of) a fraction for given fraction.
     *
     * @param fraction  given fraction to copy
     * @post new fraction with value {@code fraction} constructed
     */
    public Fraction(final Fraction fraction) {
        this(fraction.getNumerator(), fraction.getDenominator());
    }
```

Programming techniques: General versus Java-specific

The aim of this course is to teach *general* programming techniques.

That is, techniques that are useful for many programming languages.

However, each programming language requires its own ‘mind set’ (often, that is why it was invented).

There is a danger that Java-specific matters take the upper hand.

Java details are hard to avoid:

- Concrete applications of techniques help to understand them.
- ‘Real’ programs are written in a concrete programming language.

Inheritance

- A new class can be defined by extending an existing class:

  ```
  public class RuntimeException extends Exception
  public class StatisticsWithVariance extends Statistics
  ```

  Default: **extends** `Object`

- Terminology: `StatisticsWithVariance` is a subclass of `Statistics`; `Statistics` is the superclass of `StatisticsWithVariance`

- Subclass inherits all instance variables and methods in superclass, both method *signatures* and method *implementations* (if present).

  It is strongly recommended to inherit also the method *contracts*.

  The compiler does not enforce inheritance of method contracts.

- A subclass can extend only *one* class; it can add new methods and instance variables, and override inherited methods.

Inheritance versus Copy-Edit

Some aspects of inheritance can be mimicked by copy-edit:

```java
class SuperClass {
    private int instanceVariable;
    public void method() { ... }
}

class Subclass extends SuperClass {
    private int extraVariable;
    public int function() { return ...; }
}

class SubclassCopyEdit { // NO extends; NOT a subclass of SuperClass
    private int instanceVariable; // COPIED from SuperClass
    public void method() { ... } // COPIED from SuperClass

    private int extraVariable; // ADDED after copying
    public int function() { return ...; } // ADDED after copying
}
```

Polymorphism: More Than Copy-Edit

- Each variable has a compile-time type: primitive or reference.
- During execution, each (initialized) variable has a value.
- "A variable of a primitive type always holds a value of that exact primitive type."
- "A variable of a *class type* T can hold a **`null`** reference or a reference to an instance of class T or of any class that is a *subclass* of T."
- "A variable of an *interface type* can hold a **`null`** reference or a reference to any instance of any class that *implements* the interface."

Liskov Substitution Principle (LSP)

Let `U` be a subclass (possibly in multiple steps) of `T`.

Type `U` is called a subtype of type `T`, when

> In each place where an object of type `T` can be used,
>
> you can substitute an object of type `U`,
>
> without affecting the correctness of the program.

It is good practice to ensure that subclasses are also subtypes.

A subtype is not only *syntactically* a substitute (it compiles),
but also *semantically* (it works).

Meaning of method in subclass must match meaning in superclass.

Liskov Substitution Principle Violated

Java compiler will not complain about this (subclass but not subtype):

```
public class BrokenFraction extends Fraction {

    @Override
    public void add(Fraction f) {
        super.multiply(f);
    }

    @Override
    public void multiply(Fraction f) {
        super.add(f);
    }

}
```

Generic Classes and Interfaces (no details, just the concept)

- A type can be defined to have one or more type parameters.
- Example: `List<E>`, `ArrayList<E>`
- The generic type parameter must be replaced by a specific type.
- Example: `ArrayList<String>`

– O – O – O – O – O –

NEW TOPIC

Strategy Design Pattern (Recap)

- Problem: Accommodate multiple implementations of same ADT
- Problem: Possibility to select implementation at runtime
- Solution:
  - Put specification of ADT in (abstract) class or interface.
    `IntRelation`
  - Put implementations in subclasses of specification.
    `IntRelationMapOfSets extends IntRelation`
  - Declare variable with specification type.
    `IntRelation friendship`
  - Assign to variable a class of an implementation type.
    `friendship = new IntRelationMapOfSets();`

Strategy Design Pattern: Example

```java
public class FaceBook {

    private IntRelation friendship; // accommodates all implementations

    public FaceBook(final IntRelation friendship) {
        this.friendship = friendship; // should be a concrete implementation
    }

    public void connect() {
        ...
            friendship.add(a, b); // still open which implementation it uses
        ...
    }
}

...
    FaceBook myFB = new FaceBook(new IntRelationMapOfSets());
    // myFB.connect() uses add() as implemented in IntRelationMapsOfSets
```

How to Avoid Repetition of Code

```java
    public void processWaferInMachineX(final Wafer wafer) {
        do {
            // ... condition the wafer in X ...;
            // ... measure the wafer in X...;
        } while (... ! successful measurement ...);
        // ... expose the wafer in X...;
    }

    public void processWaferInMachineY(final Wafer wafer) {
        do {
            // ... condition the wafer in Y ...;
            // ... measure the wafer in Y ...;
        } while (... ! successful measurement ...);
        // ... expose the wafer in Y ...;
    }
```

How to apply DRY (Don't Repeat Yourself)?

Template Method Pattern Motivation

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

  (Hence, parameters are not a solution.)
- Difference is in some steps

Concern: How to avoid code duplication?

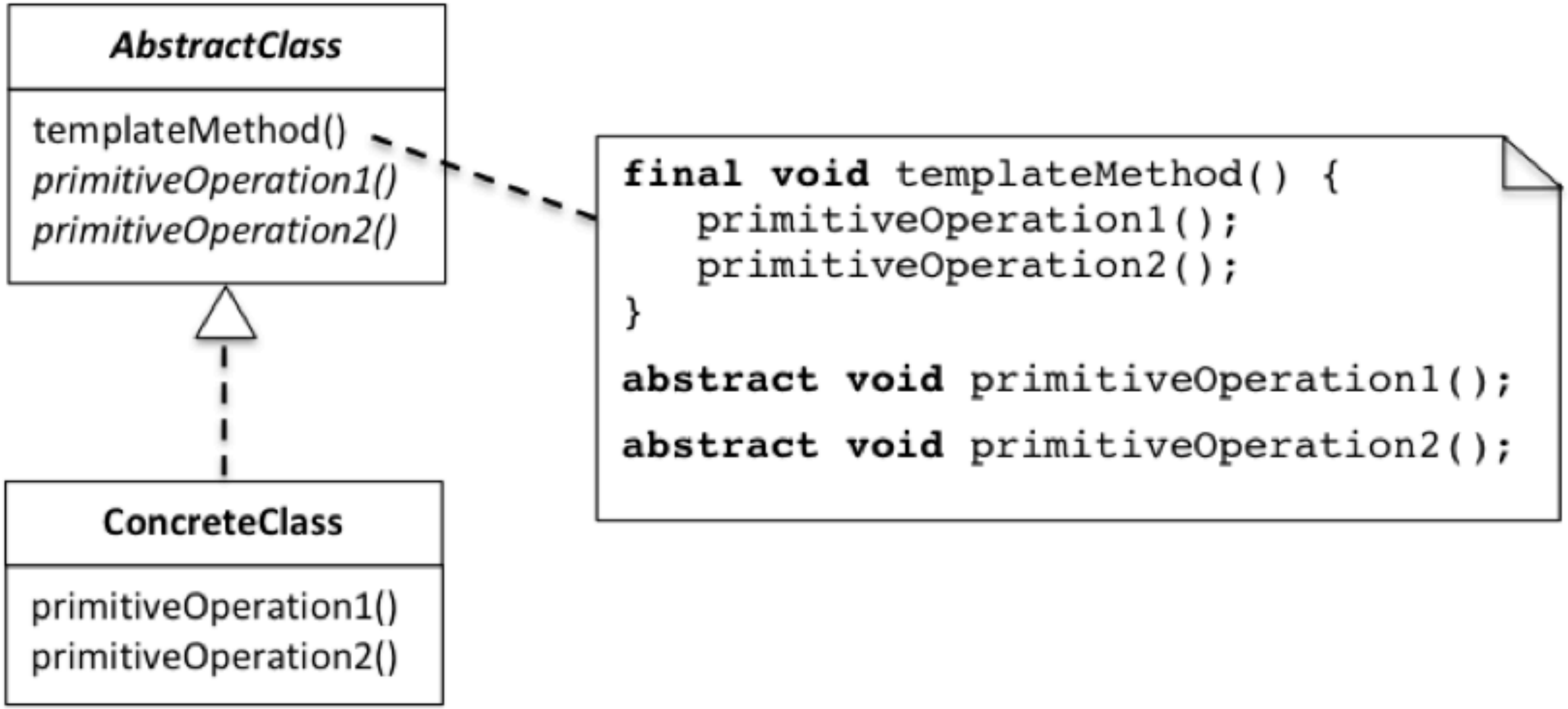
Template Method Design Pattern (adapted from Eddie Burris)

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 a.k.a. hook methods.
- Subclasses [can] extend or replace some or all of these hooks.

Template Method Pattern (from Burris)



Why the Template Method Pattern Works

- Depends on how the method definition is found for a method call.
- When method m is called on object o,

  m's definition is searched by *ascending* the inheritance hierarchy

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

  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 Template Method Pattern Works: Example

- Given object `ConcreteClass c;` consider call `c.templateMethod()`
- `ConcreteClass` does not define `templateMethod()`
- Definition of `templateMethod()` in found in superclass `AbstractClass`
- `templateMethod()`, in turn, calls `primitiveOperationX()`
- Definition of `primitiveOperationX()` is found in `ConcreteClass`

Template Method Pattern 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.
- This way, individual steps can be varied separately.

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.

– O – O – O – O – O –

NEW TOPIC

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:

```
FunctionalityA.doA(n, new FunctionalityB() {
    int count;
    @Override
    public void doB(String data) {
        ++ count;
        System.out.println("Item " + count + " counted");
    }
});
```

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`** for concrete class
- Consequently, things may be harder to change, test, and reuse

Factory Method Design Pattern (adapted from Eddie Burris)

Intent

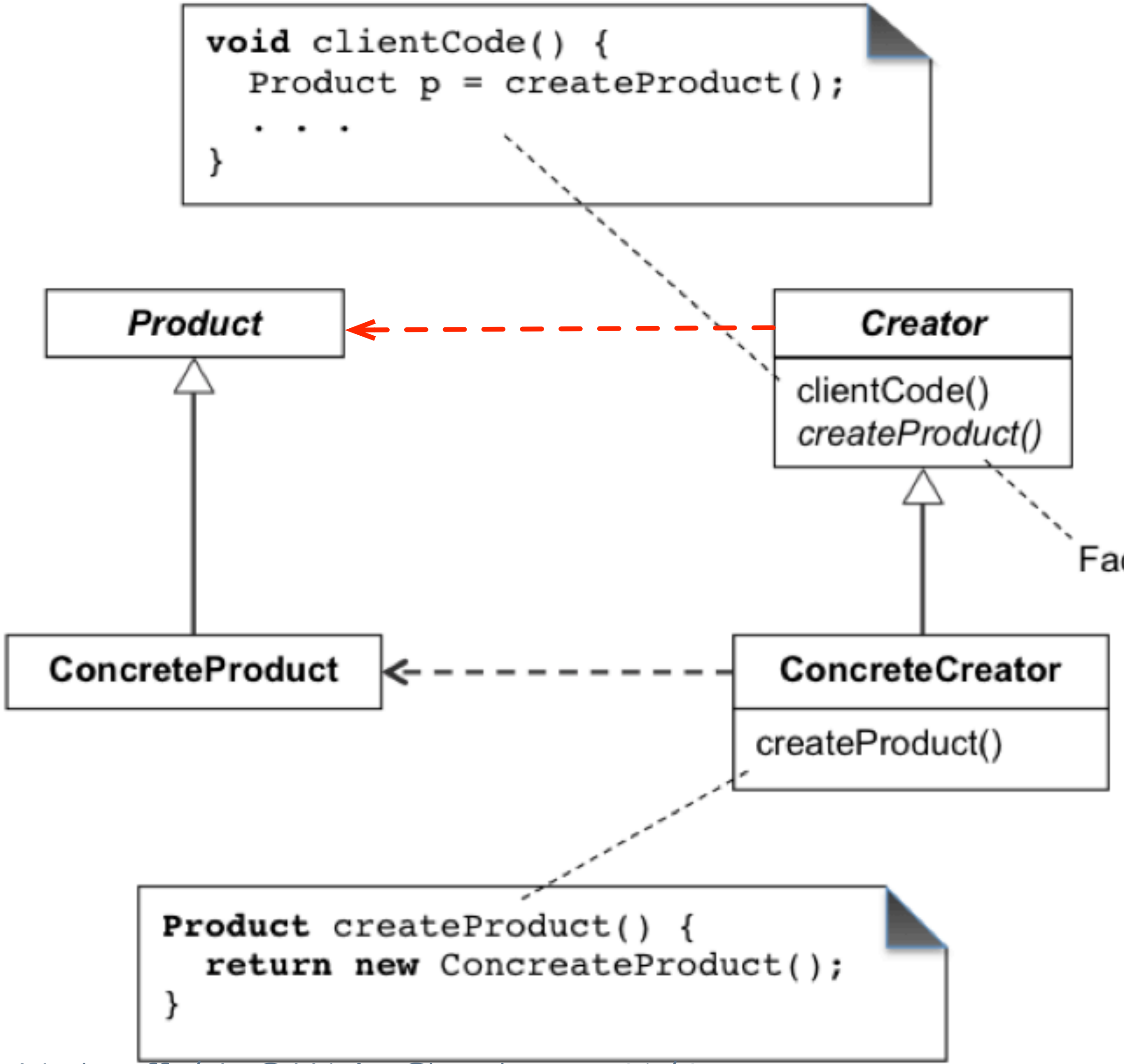
The Factory Method design pattern [quoting the Gang of Four]

- “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 Pattern 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`

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.

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

Or: parameterize the class, and let client create new object of choice

Summary (1)

- Immutable versus mutable types
- `equals()`, `similar()`, `clone()`
- Single inheritance: **class** `SubClass` **extends** `SuperClass`
- **abstract class**, with **abstract** methods, `@Override`
- **interface**
- **class** `MyClass` **implements** `InterfaceA, InterfaceB`
- Polymorphism: *runtime* type of a variable value can be a subtype of the variable's *compile-time type*; search for methods at runtime
- Liskov Substitution Principle (LSP): subclass versus subtype

Summary (2)

Template Design Pattern

- Problem:
  - Allow sharing of algorithmic template among multiple implementations, while implementations vary the algorithmic steps.
- Solution:
  - Put template method in (abstract) superclass, implemented in terms of abstract steps.
  - Put *specification* of steps (hook methods) in superclass.
  - Put *implementation* of steps in a subclass that extends the superclass.

Summary (3)

Factory Method Design Pattern

- Problem:
  - Allow varying of object creation among multiple implementations
- Solution:
  - Specify abstract factory method in (abstract) superclass, with goal of creating a new product.
  - Use the factory method in the superclass for product creation.
  - Put concrete factory method in a subclass that extends the superclass.