Software Engineering

CS305, Autumn 2020

Week 7

Class Progress…

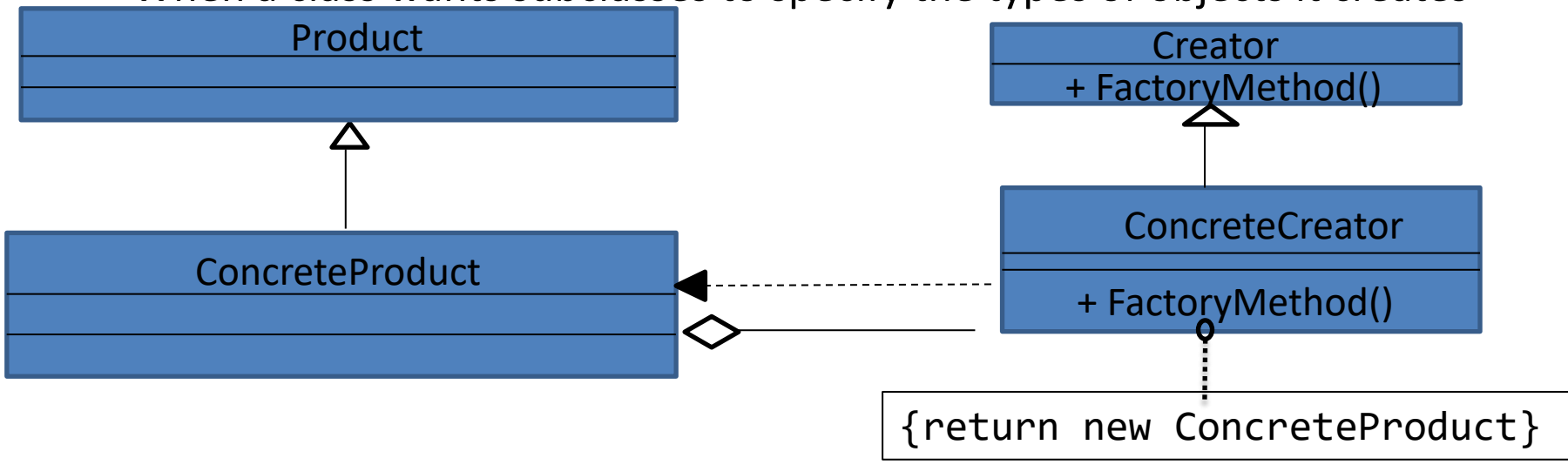
- Last week:
  - Architectural styles
    - Shared services and servers, repository, layered
  - Detailed design
    - Design patterns
    - Singleton

Class Progress…

- This week:
  - Design patterns, Design principles, Rational Unified Process

Factory Method Pattern

- Intent: define an interface for creating an object, and let applications decide which object type to create.
- Applicability
  - When the exact type of object to be created is known at runtime
  - When a class needs control over object creation
  - When a class wants subclasses to specify the types of objects it creates



Factory Method Pattern

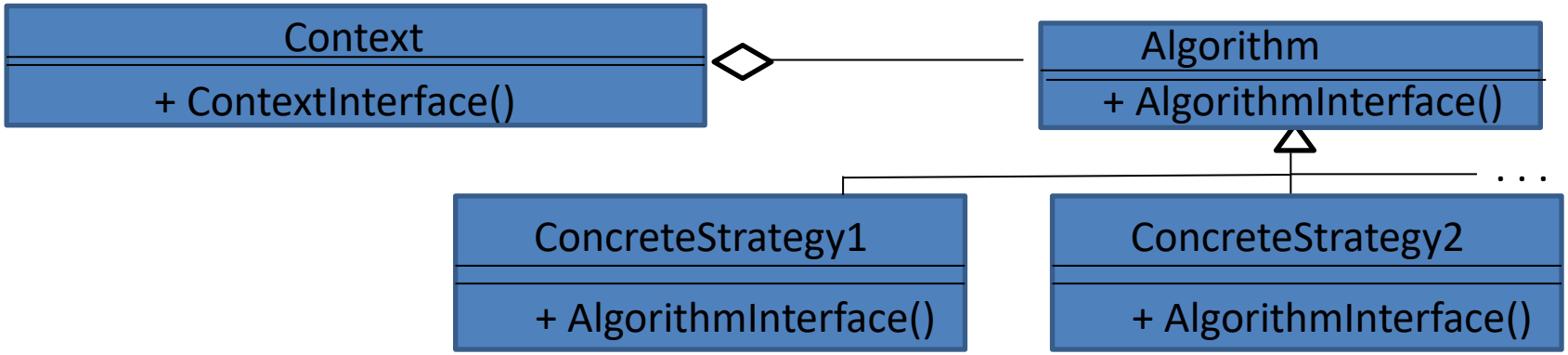
```
Vehicle* VehicleFactory(VehicleType type, Color c) {
        if(type == BUS)
                return new Bus(c);
        else if(type == CAR)
                return new Car(c);
        else
                return NULL;
}

int main() {
        Vehicle* redCar = VehicleFactory(CAR,RED);
        Vehicle* blueBus = VehicleFactory(BUS,BLUE);
}
```

Comment about the structure of `VehicleFactory`?

Strategy Pattern

- Intent: encapsulate each one of a family of algorithms in a separate class and make their usage agnostic
- Applicability
  - When the exact type of object to be created is known at runtime
  - When a class needs control over object creation
  - When a class wants subclasses to specify the types of objects it creates



Some Commonly Used Patterns

- **Visitor** – separate the algorithm from the data structure on which it operates e.g. finding minimum in a binary tree, finding maximum in a binary tree, finding multiples of a given number in a binary tree.
- **Observer** - notify dependents when object changes
- **Iterator** – access elements of a collection without knowing about underlying representation
- **Proxy** – a surrogate controls access to an object

Choosing a Pattern

- Broad guidelines
  - Understand design context
  - Examine the patterns catalogue
  - Identify and study related patterns
  - Apply suitable pattern
- Avoid:
  - Overusing patterns

Design Principles

- **Performance vs. Maintainability tradeoff**
  - **Performance goal:** localize critical operations and minimize communications. Therefore, use coarse-grain rather than fine-grain components. *Coarse-grain components are difficult to maintain*
  - **Maintainability goal:** use fine-grain, replaceable components. *Fine-grain components localize communication*
- **Security vs. Availability tradeoff**
  - **Security goal:** secure critical assets in the inner layers when using a layered architecture.
  - **Availability goal:** include redundant components and mechanisms for fault tolerance. *Redundant components increase availability. However, security becomes difficult.*
- **Safety vs. Communication/Performance tradeoff**
  - **Safety goal:** localize safety-critical features in a small number of sub-systems. *Localizing means more communication and hence, degraded performance.*

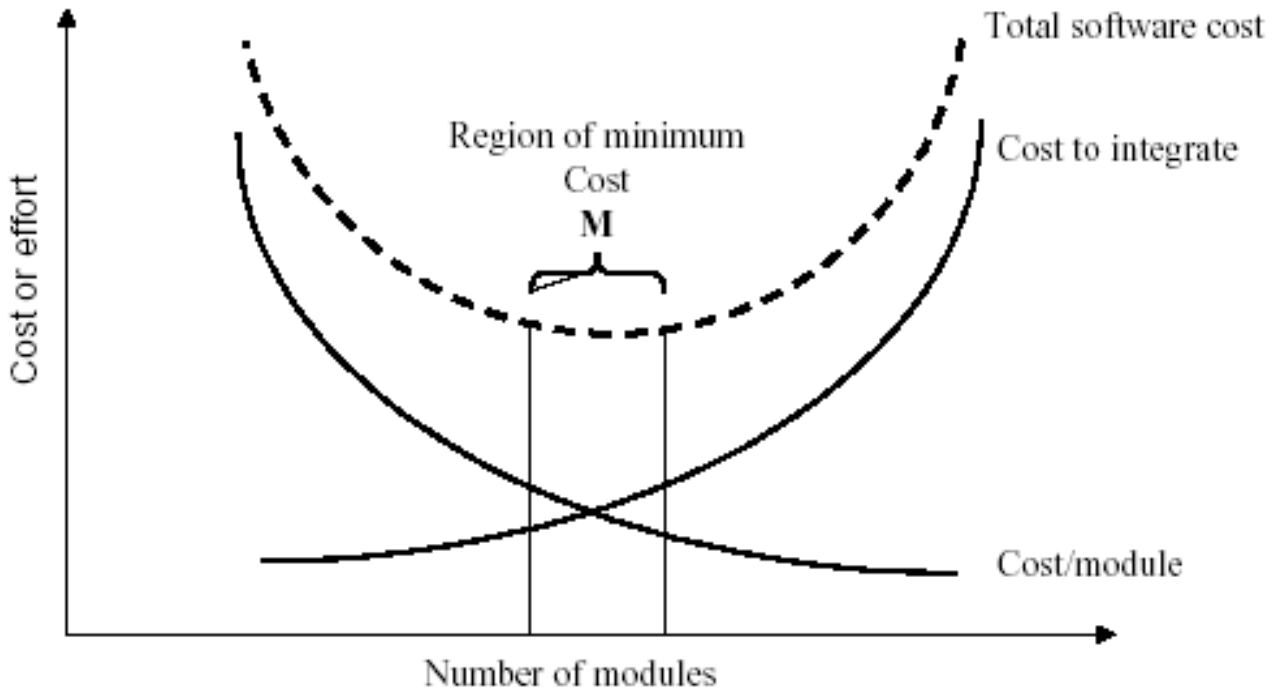
Design Principles

- Balance coupling and cohesion with non-functional requirements
- Consider information hiding
  - Provide abstraction / refinement
- Document design decisions *(design rationale)*

Design Principles

- *Coupling vs. Cohesion*
  - Recall that coupling is the extent to which two components depend on each other for successful execution. *Low coupling is good*
  - *Recall that Cohesion* is the extent to which a component has a single purpose or function. *High cohesion is good*

Modularity and Software Cost



- consider low coupling/high cohesion
  - module should be 'stand alone', errors contained as much as possible
- consider requirements
  - change in requirements should minimize number of modules affected

Slide courtesy: Alex Orso, CS3300_Lectures

Design Decisions - dimensions

- Architecture level:
  - Choose from repository, service, layered, …
- Component level:
  - identify components
- Connector level: determine control model
  - Choose from centralized, event-driven, …
- Subsystem level:
  - Choose from behavioral, object, … models

Design Principles - SOLID

- Single-responsibility Principle
  - *"A class should have single responsibility"* – to prevent from side-effects resulting from future requirements changes
- Open-Closed Principle
  - *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."* – should be able to add new functionality without modifying existing code

Design Principles - SOLID

- Liskov-Substitution Principle
  - objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
  - Pretty similar to Bertrand Meyer's *design-by-contract* principle
- Interface Segregation principle
  - *"Clients should not be forced to depend upon interfaces that they do not use."*
- Dependency Inversion Principle
  1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  2. Abstractions should not depend on details. Details should depend on abstractions.

Further reading: https://stackify.com/solid-design-principles/