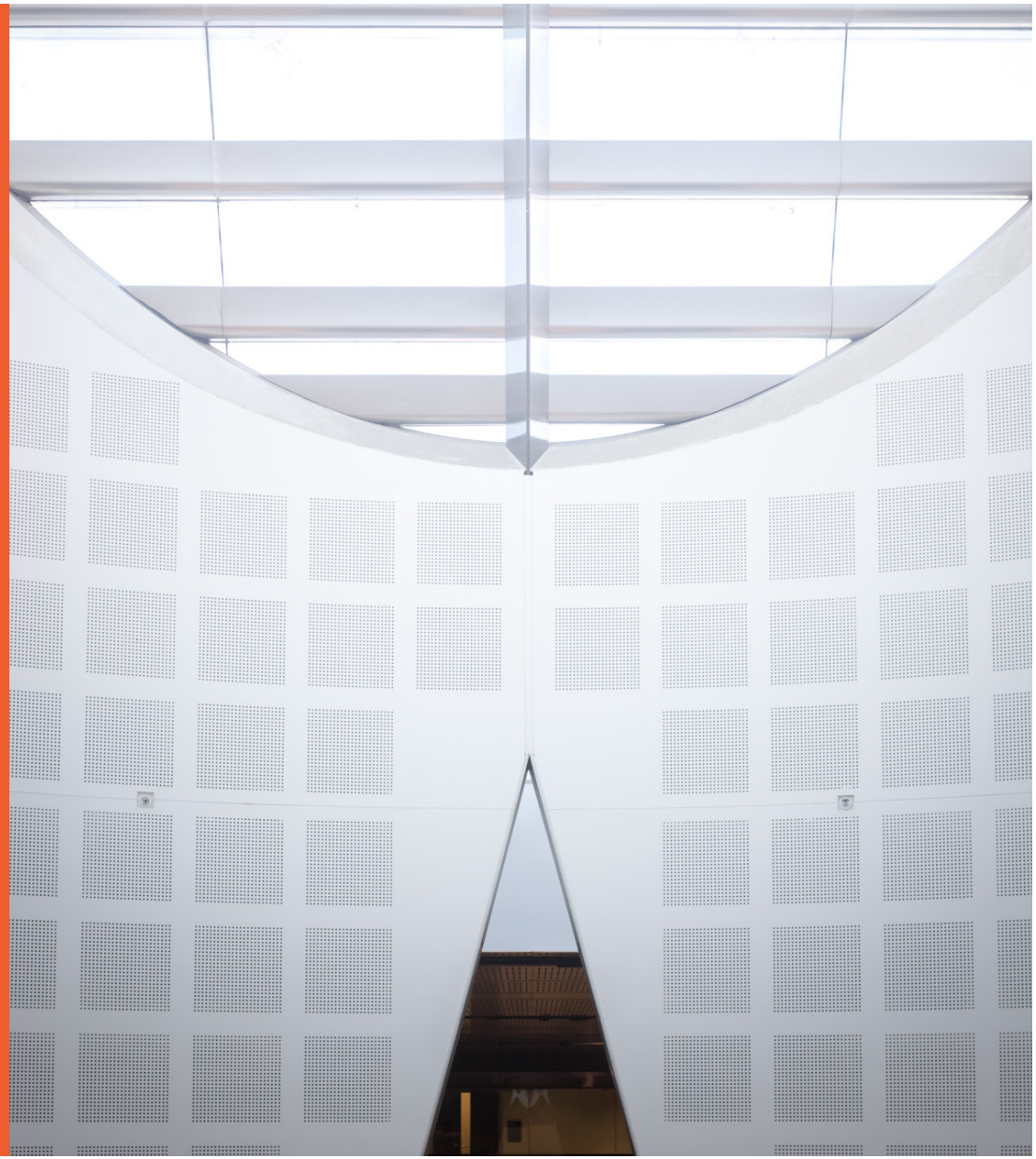
Software Design and Construction 1
SOFT2201 / COMP9201
Introduction to Design Patterns

Dr. Xi Wu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Agenda

- Design Patterns
  - GoF Design Patterns
- Creational Deign Patterns
  - Factory Method Pattern
  - Builder Pattern

What is Design Pattern?

THE UNIVERSITY OF SYDNEY

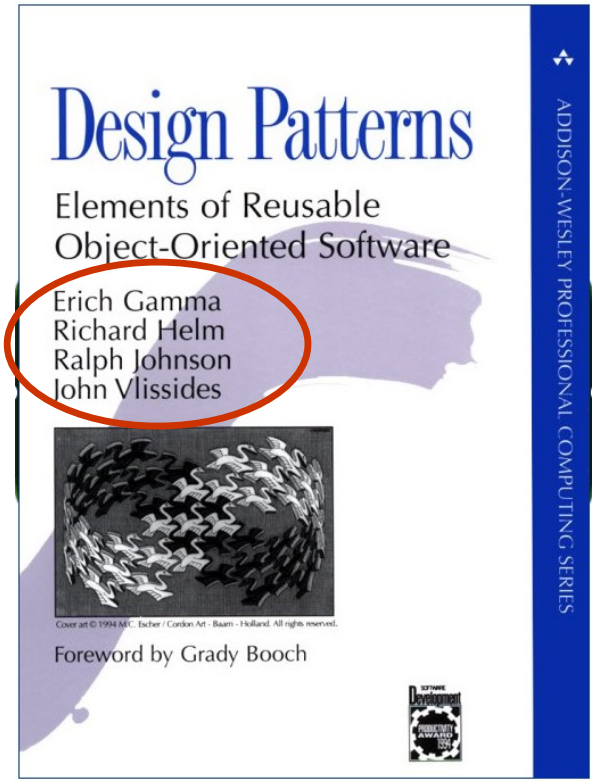
Design Patterns

- A pattern is a description of a problem and its solution
- Tried and tested ideas for recurring design problem
- Not readily coded solution, but rather the solution path to a common programming problem
- Design or implementation **structure** that achieves a particular purpose

Essential Components of a Pattern

- The **pattern name**
  - e.g., Factory Method
- The **problem**
  - The pattern is designed to solve (i.e., when to apply the pattern)
- The **solution**
  - The components of the design and how they related to each other
- **Consequence**
  - The results and trade-offs of applying the pattern
  - Advantages and disadvantages of using the pattern

Gang of Four Patterns (GoF)



- Official design pattern reference
- Famous and influential book about design patterns
- Recommended for students who wish to become experts
- We will cover the most widely - used patterns from the book
- 23 patterns in total - our unit will focus on 11
- GoF Design Patterns → Design Patterns as short

Design Patterns – Classification based on purpose

Scope	Creational	Structural	Behavioural
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

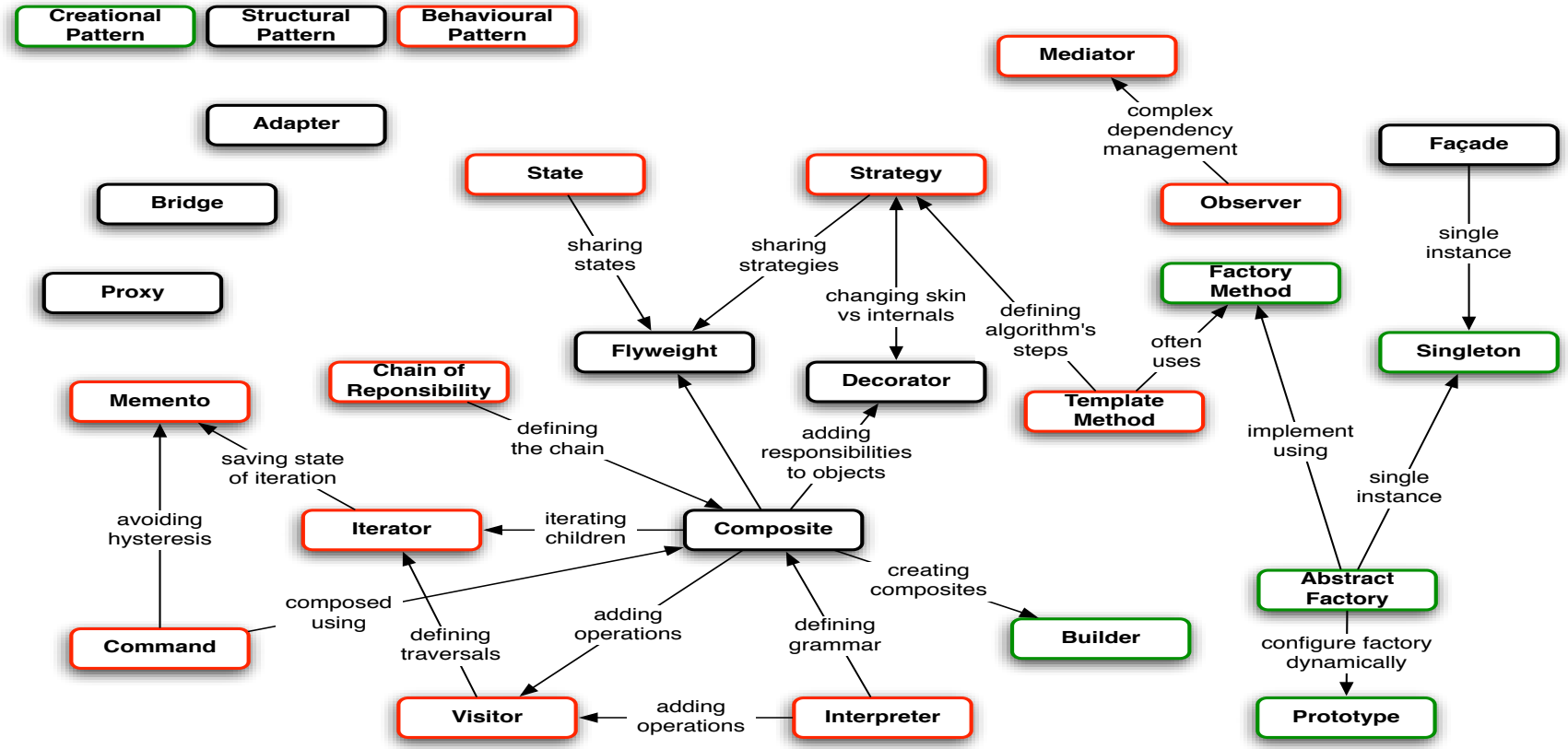
Design Patterns – Classification based on purpose

- **Creational patterns**
  - Abstract the instantiation process
  - Make a system independent of how its objects are created, composed and represented
- **Structural patterns**
  - How classes and objects are composed to form larger structures
- **Behavioral patterns**
  - Concerns with algorithms and the assignment of responsibilities between objects

Design Patterns – Classification based on relationships

- Patterns often used together
  - E.g., Composite often used with Iterator or Visitor
- Alternative Patterns
  - E.g., Prototype often alternative to Abstract Factory
- Patterns results in similar designs
  - E.g., Structure diagram of composite and Decorator are similar

Design Patterns – Classification based on relationships



* Adapted from the GoF book: Design Patterns

Selecting an Appropriate Design Pattern

It is useful to have some guidelines of how to select one. Here are some thoughts on how you might do that:

- Consider the ways in which the design patterns solve problems.
  - We will go into details on this for several design patterns
- Decide on what the intent of each design is.
  - Without knowing what the motivation of the design pattern is, it will be hard to know whether it is right for you
- Look at the relationships among patterns
  - It makes sense to use patterns that have a clear relationship, rather than one that you have manufacture ad hoc

Selecting an Appropriate Design Pattern

- Consider patterns with similar purpose
  - Creational, Structural and Behavioral are quite different purposes so you should consider which one you need
- Look at why redesign might be necessary
  - Knowing why a redesign might be needed will help you select the right design to avoid having to redesign later
- Why can vary?
  - Your design should be open to variation where necessary
  - Choose a design that will not lock you into a particular one, and enable you to make variations without changing your design

Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Façade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location

Design Aspects Can be Varied by Design Patterns

Purpose	Pattern	Aspects that can change
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private info. is stored outside an object, & when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Creational Patterns

THE UNIVERSITY OF SYDNEY

Creational Patterns

- Abstract the instantiation process
- Make a system independent of how its objects are created, composed and represented
  - Class creational pattern uses inheritance to vary the class that's instantiated
  - Object creational pattern delegates instantiation to another object
- Provides flexibility in *what gets created*, *who creates it*, *how it gets created* and *when*

Creational Patterns

Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Singleton	Ensure a class only has one instance, and provide global point of access to it
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype

Factory Method

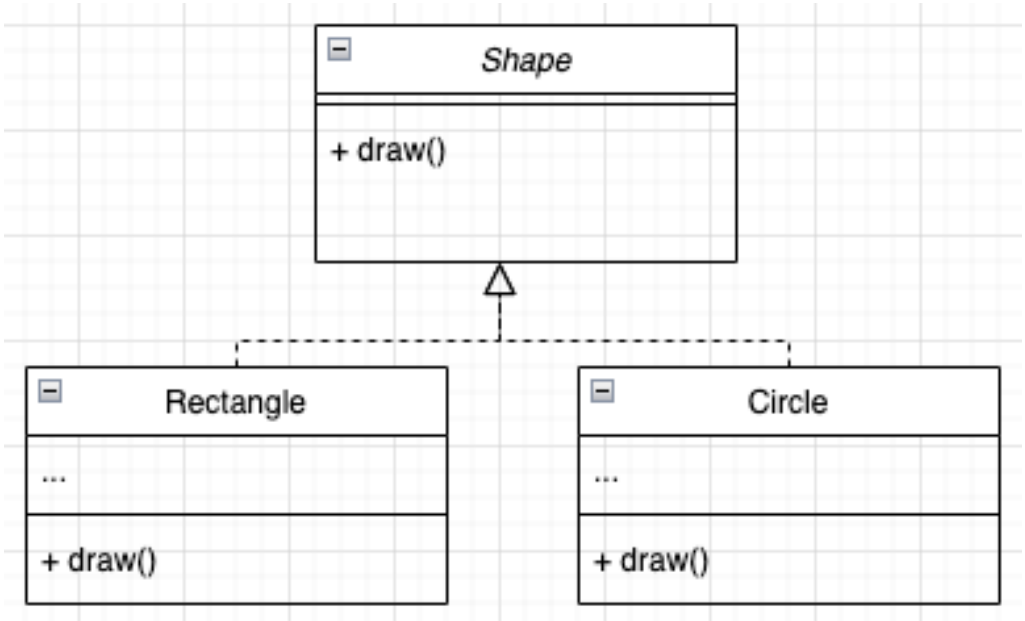
Class Creational Pattern

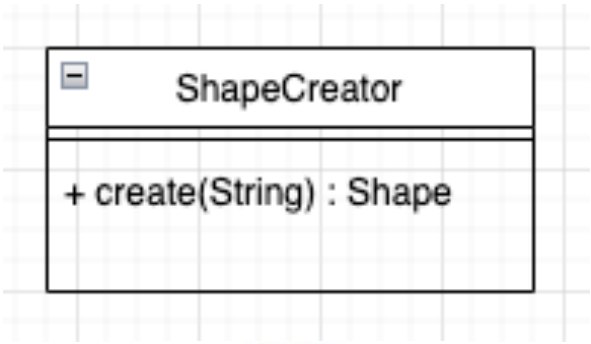
An interface for creating an object

THE UNIVERSITY OF SYDNEY

Motivated Scenario

- Suppose you have a general shape creator that can create a variety of different shapes.





Motivated Scenario

```
public class ShapeCreator {
    1 usage
    public Shape create(String type) {
        Shape shape = null;
        switch (type) {
            case "rectangle":
                shape = new Rectangle();
                break;
            case "circle":
                shape = new Circle();
                break;
        }
        return shape;
    }
}
```

```
public interface Shape {
    1 usage  2 implementations
    public void draw();
}
```

```
public class Circle implements Shape{
    1 usage
    public void draw() { System.out.println("I am drawing a circle"); }
}
```

```
public class Rectangle implements Shape {
    1 usage
    public void draw() { System.out.println("I am drawing a rectangle"); }
}
```

Client Perspective:

```
ShapeCreator creator = new ShapeCreator();
Shape shape = creator.create("circle");
shape.draw();
```

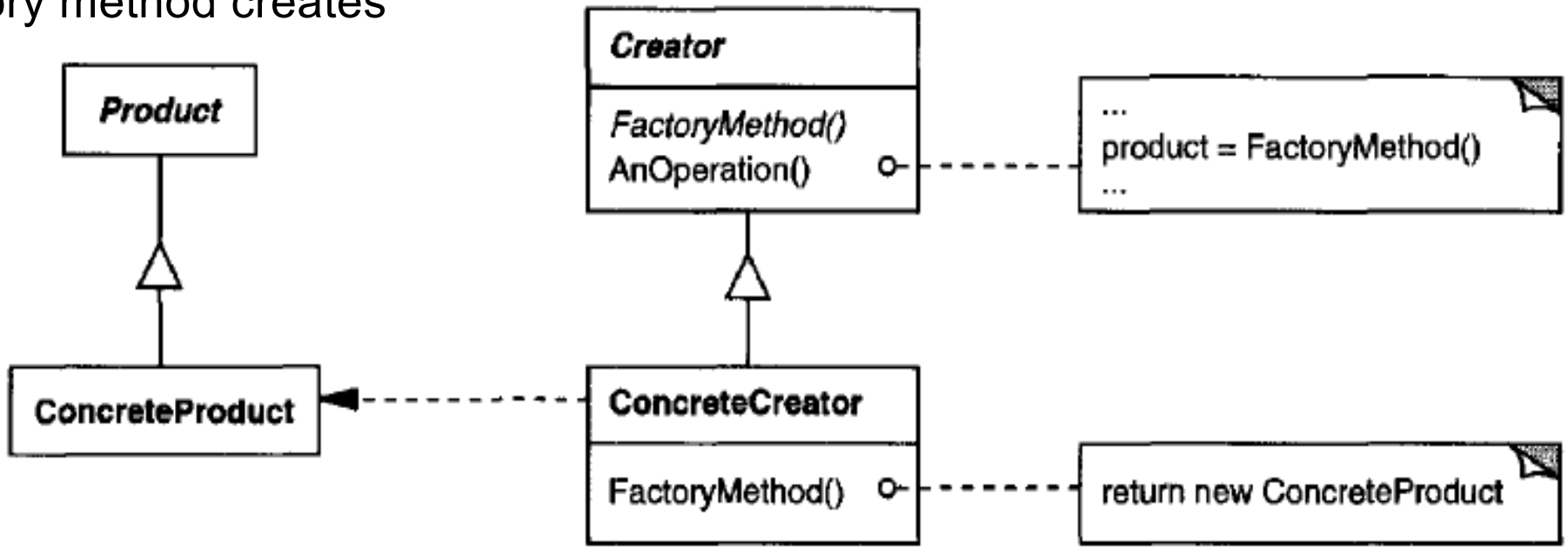
Factory Method Pattern

- Purpose/Intent
  - Define an interface for creating an object
  - Let subclasses decide which class to instantiate.
  - Let a class defer instantiation to subclasses
- Also known as
  - Virtual Constructor

Factory Method Structure

Defines the interface of objects the factory method creates

Declares the factory method, which returns an object of type Product.



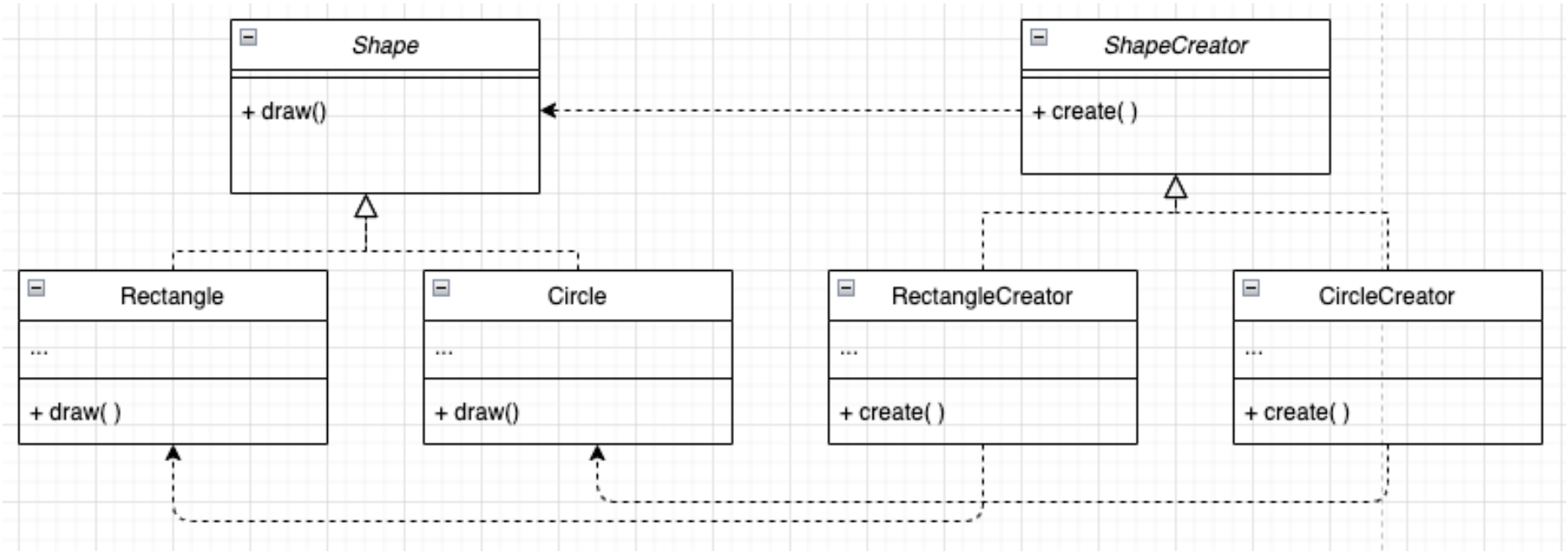
Implements the Product interface

Overrides the factory method to return an instance of the ConcreteProduct

Factory Method Participants

- **Product**
  - Defines the interface of objects the factory method creates
- **ConcreteProduct**
  - Implements the Product interface
- **Creator**
  - Declares the factory method, which returns an object of type Product.
- **ConcreteCreator**
  - Overrides the factory method to return an instance of the ConcreteProduct

Revisit the Motivated Example



Revisit the Motivated Example

```
public interface ShapeCreator {
    1 usage   2 implementations
    public Shape create();
}
```

```
public class CircleCreator implements ShapeCreator {
    1 usage
    @Override
    public Shape create() {
        return new Circle();
    }
}
```

```
public class RectangleCreator implements ShapeCreator{
    1 usage
    @Override
    public Shape create() {
        return new Rectangle();
    }
}
```

Client Perspective

```
ShapeCreator creator = new CircleCreator();
Shape shape = creator.create();
shape.draw();
```

Factory Method Pattern

- Applicability
  - A class cannot anticipate the class objects it must create
  - A class wants its subclasses to specify the objects it creates
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
- Benefits
  - Flexibility: subclasses get a hook for providing an extension to an object; connects parallel class hierarchies
- Limitations
  - Can require subclassing just to get an implementation

Two varieties of Factory

- Variety 1: the Creator class is abstract
  - This requires subclasses to be made because there is no reasonable default value.
  - On plus side, this avoids the problem of dealing with instantiating unforeseeable classes
- Variety 2: the Creator class is concrete
  - Creator may also define a default implementation of the factory method that returns a default *ConcreteProduct* object
  - This provides reasonable default behaviors, and enables subclasses to override the default behaviors where required

Factory Registry -- Selecting a Factory Method Source

- In essence, a mapping from identifier to implementation
- Responsibility for choice of Factory Method concentrated in one location

```
public class FactoryRegistry {
  private Map<Identifier, Factory> factories = new TreeMap<>();
  public void registerFactory(Factory factory, Identifier identifier) {…}
  public Factory getFactory(Identifier identifier) {…}
}
```

Builder

Object Creational Pattern

THE UNIVERSITY OF SYDNEY

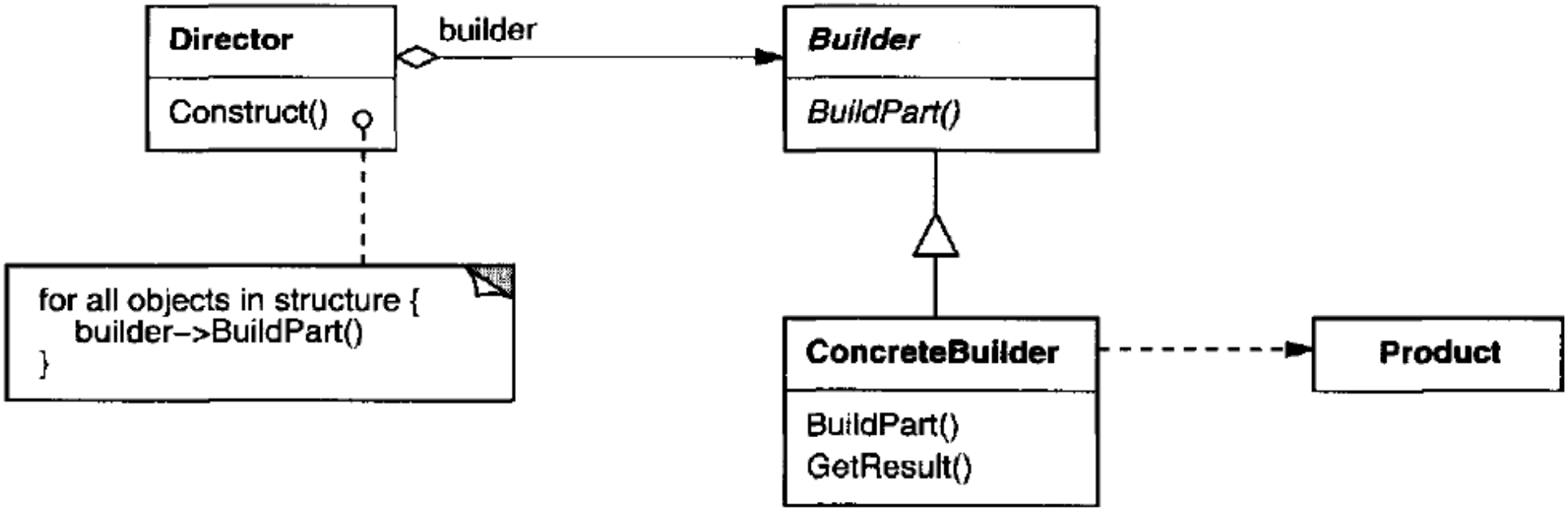
Motivated Scenario

- Suppose the design team is going outside to have a coffee break.
  - John would like a CoffeeA that contains no milk and no sugar
  - Xi would like a CoffeeB that contains no milk but full sugar
  - Sue would like a CoffeeC that contains full milk and full sugar

Builder

- Purpose/Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
  - You have a range of implementations that might expand, so must be flexible, or you want to create instances of complex objects

Builder -- Structure



Builder – Participants

- **Builder**
  - Specifies an abstract interface for creating parts of a Product object

- **ConcreteBuilder**
  - Constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates.
  - provides an interface (GetResult) for retrieving the product

Builder – Participants

- **Director**
  - Constructs an object using the Builder interface

- **Product**
  - Represents the complex object under construction.
  - ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Revisit the Motivated Example

"abstract" builder

```
public interface CoffeeBuilder {
    1 usage  3 implementations
    public void pickupCoffeeBean();
    1 usage  3 implementations
    public void grindCoffeeBean();
    1 usage  3 implementations
    public void addWater();
    1 usage  3 implementations
    public void filterCoffee();
    1 usage  3 implementations
    public void addMilk();
    1 usage  3 implementations
    public void addSugar();
}
```

one concrete builder as an example

```
public class CoffeeABuilder implements CoffeeBuilder{
    1 usage
    @Override
    public void pickupCoffeeBean() {System.out.println("We pick up the proper Coffee bean for CoffeeA");}
    1 usage
    @Override
    public void grindCoffeeBean() {System.out.println("CoffeeA bean needs to be grind for 1 minute");}
    1 usage
    @Override
    public void filterCoffee() {System.out.println("We use filterA for CoffeeA");}
    1 usage
    @Override
    public void addWater() {System.out.println("CoffeeA needs 100 C water");}
    1 usage
    @Override
    public void addMilk() {System.out.println("No milk added for CoffeeA");}
    1 usage
    @Override
    public void addSugar() {System.out.println("No sugar added for CoffeeA");
    }
}
```

Revisit the Motivated Example

director

```
public class CoffeeDirector {
    7 usages
    private CoffeeBuilder cb;

    1 usage
    public CoffeeDirector(CoffeeBuilder cb) {
        this.cb =  cb;
    }

    1 usage
    public void construct() {
        cb.pickupCoffeeBean();
        cb.grindCoffeeBean();
        cb.filterCoffee();
        cb.addWater();
        cb.addMilk();
        cb.addSugar();
    }
}
```

client perspective

```
CoffeeBuilder b = new CoffeeABuilder();
CoffeeDirector director = new CoffeeDirector(b);
director.construct();
```

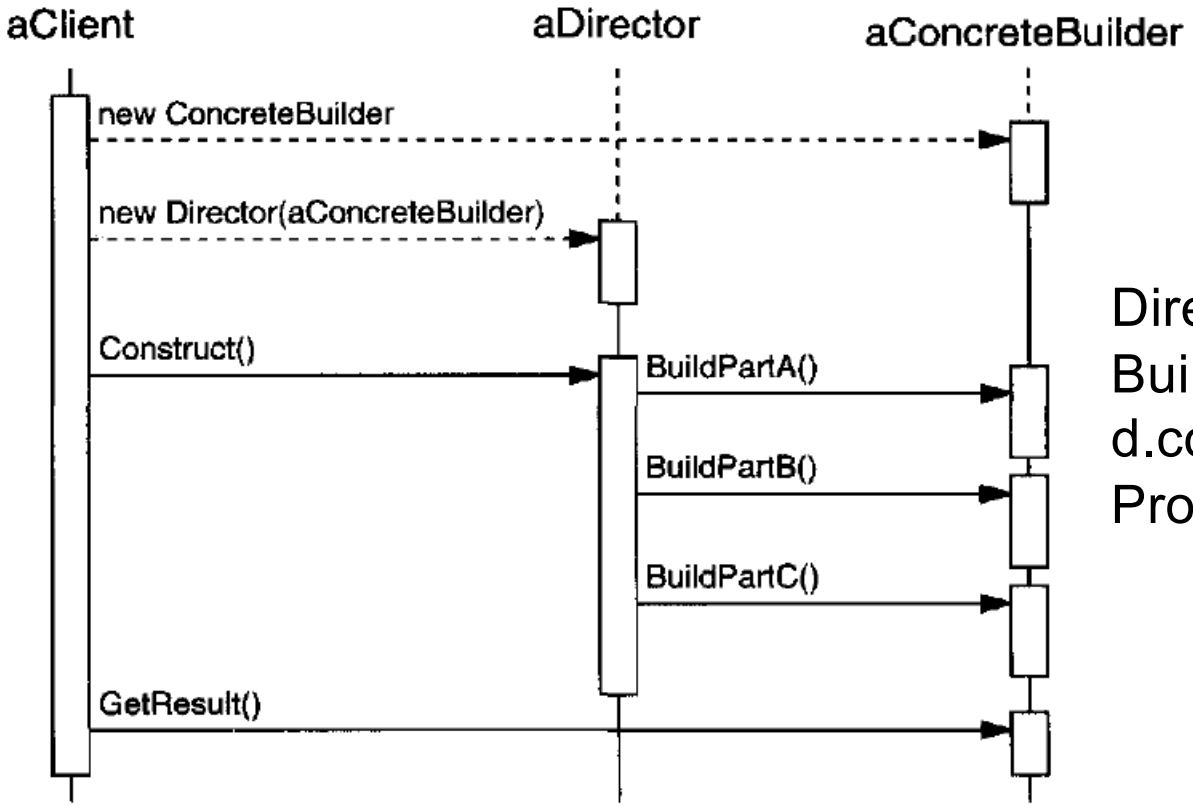
result:

```
We pick up the proper Coffee bean for CoffeeA
CoffeeA bean needs to be grind for 1 minute
We use filterA for CoffeeA
CoffeeA needs 100 C water
No milk added for CoffeeA
No sugar added for CoffeeA
```

Builder

- Applicability
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - The construction process must allow different representations for the object that's constructed
- Benefits
  - Gives flexibility that can be extended by implementing new subclasses of the Builder, and isolates code of construction from implementation
- Limitations
  - Not completely generic, less useful in situations where variety of implementations is not high

Builder – Collaboration




```
Director d = new Director( );
Builder b = new ConcreteBuilder1( );
d.construct(b);
Product p = b.GetResult( );
```

Builder – Consequences (1)

- Varying product's internal representation
  - Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder

- Isolation of code construction and representation
  - Each ConcreteBuilder contains all the code to create and assemble a particular kind of product.
  - Different Directors can reuse it to build Product variants from the same set of parts

Builder – Consequences (2)

- Finer control over the construction process
  - The builder pattern constructs the product step by step under the director's control
  - Only when the product finished does the director retrieve it from the builder
  - The builder interface reflects the process of constructing the product more than other creational patterns

Task for Week 5

- Submit weekly exercise on canvas before 23.59pm Saturday
- Prepare questions and ask during tutorial this week

What are we going to learn next week?

- Behavioral Design Patterns
  - Strategy Pattern
  - State Pattern

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition). Prentice Hall PTR, Upper Saddle River, NJ, USA.