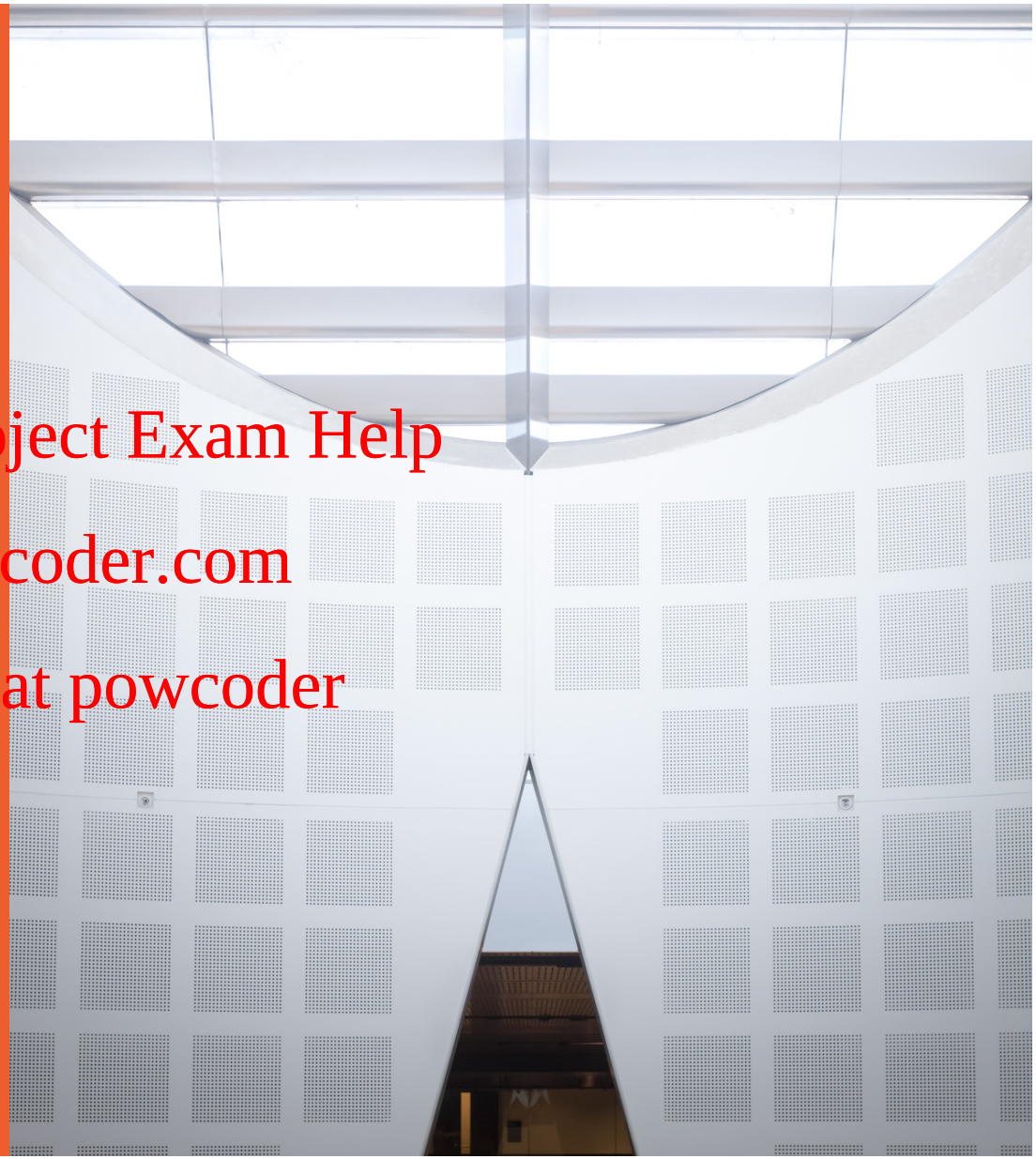
Software Design and Construction 1
SOFT2201 / COMP9201

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.

Announcement

- Details of Assignment 2 will be released this week on canvas (an announcement will be sent to you on both canvas and Ed discussion forum once it is released)
- This week's tutorial will be divided into two parts: one-hour tutorial questions discussion + one-hour tutor Q&A session
- One-hour Helpdesk session (mainly focus on implementation questions/helps) opens from this week onwards
  - 1pm to 2pm every Thursday on zoom (link can be found on canvas)

Agenda

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

What is Design Pattern?

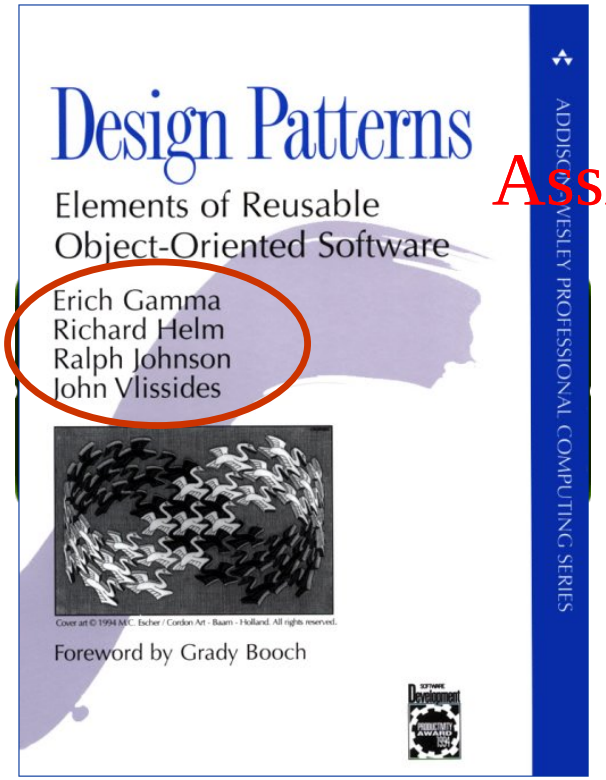
Design Patterns

- 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

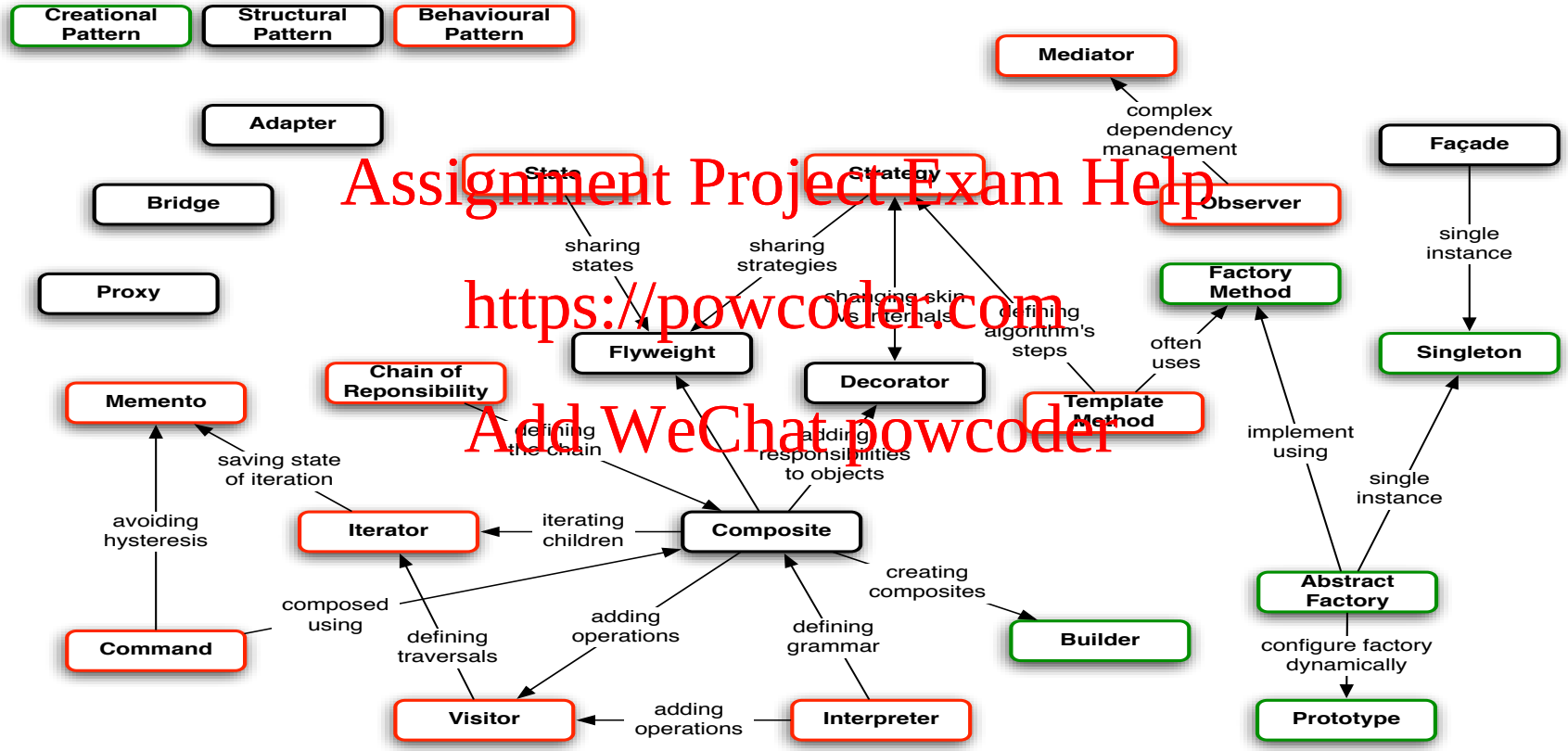
Design Patterns – Classification based on purpose

- **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

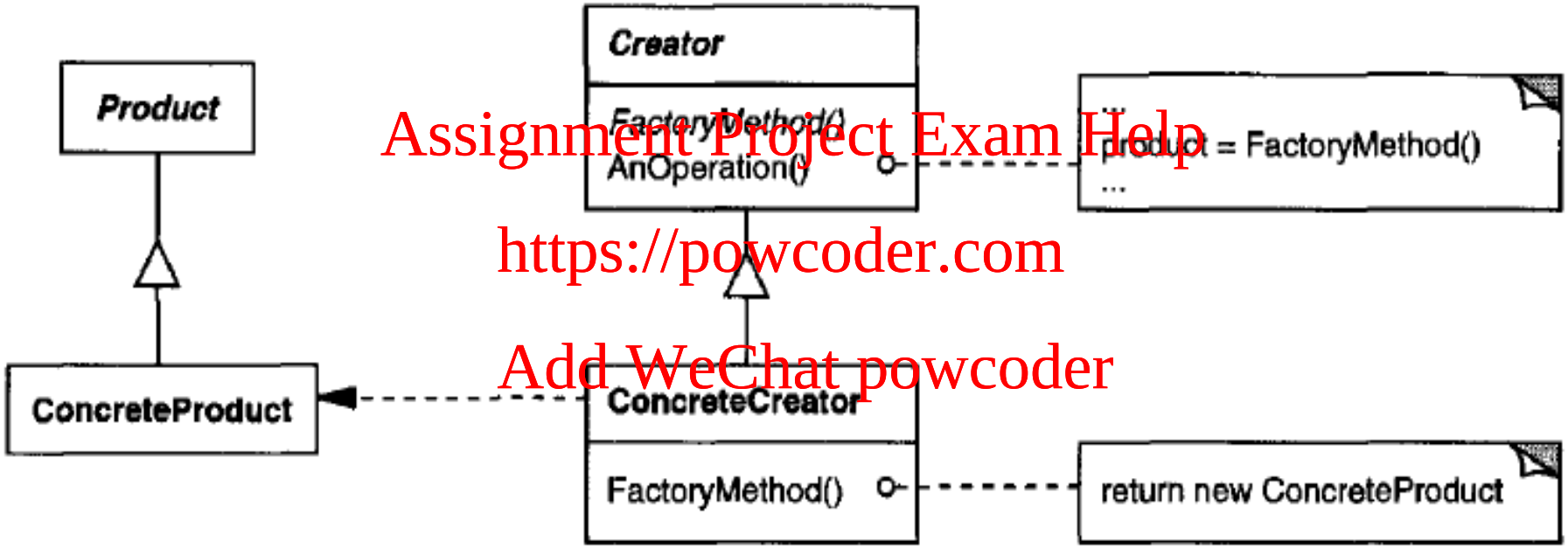
Factory Method Pattern

- Purpose/Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Let a class defer instantiation to subclasses
- Also known as
  - Virtual Constructor
- Motivated Scenario
  - Suppose you have a general application framework (like Office) that can create a variety of different applications and document types. To create a spreadsheet applications we might define SpreadsheetApp and SpreadsheetDoc; for a word processing application we'd have WordProcApp and WordProcDoc. Both use the common framework.

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

Factory Method Structure



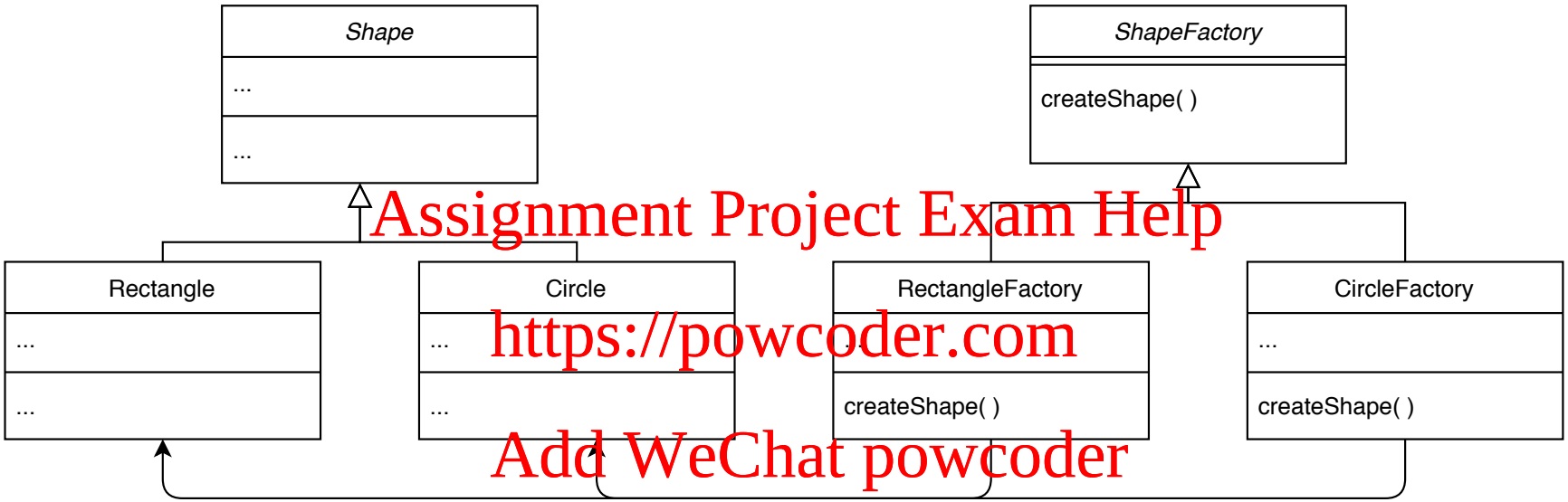
Factory Method Participants

- **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

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

Example




```
ShapeFactory sf = new RectangleFactory( );
Shape x = sf.createShape( );
```

Builder

Object Creational Pattern

THE UNIVERSITY OF SYDNEY

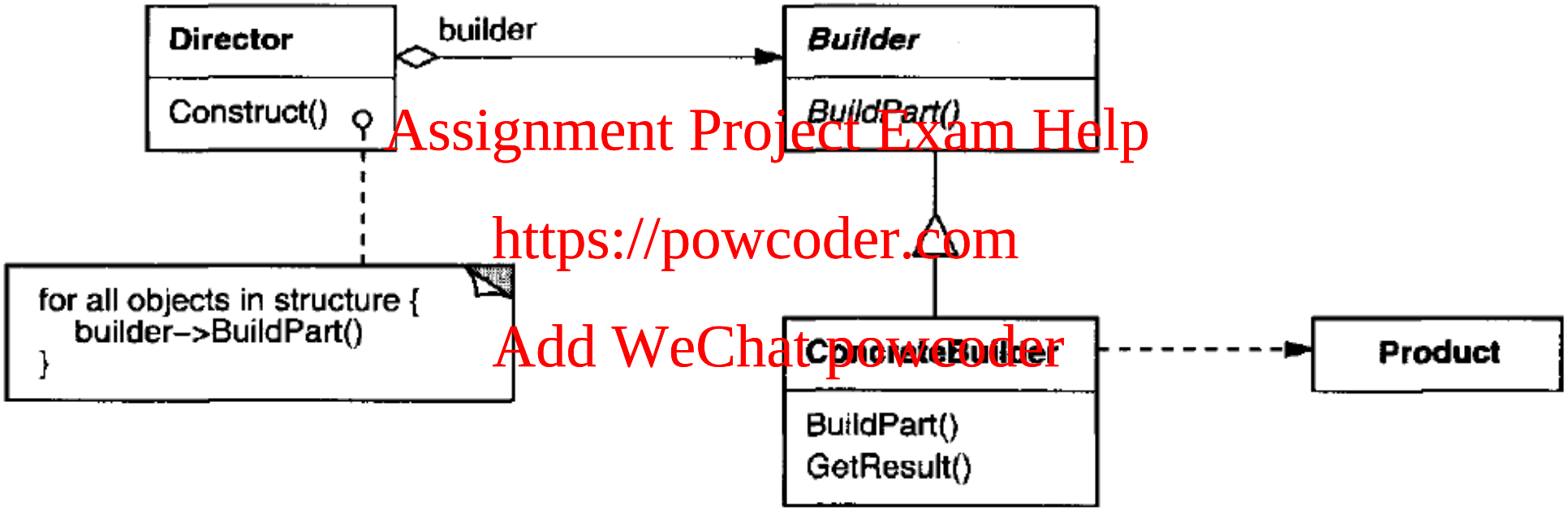
Builder

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

Builder

- 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 -- 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

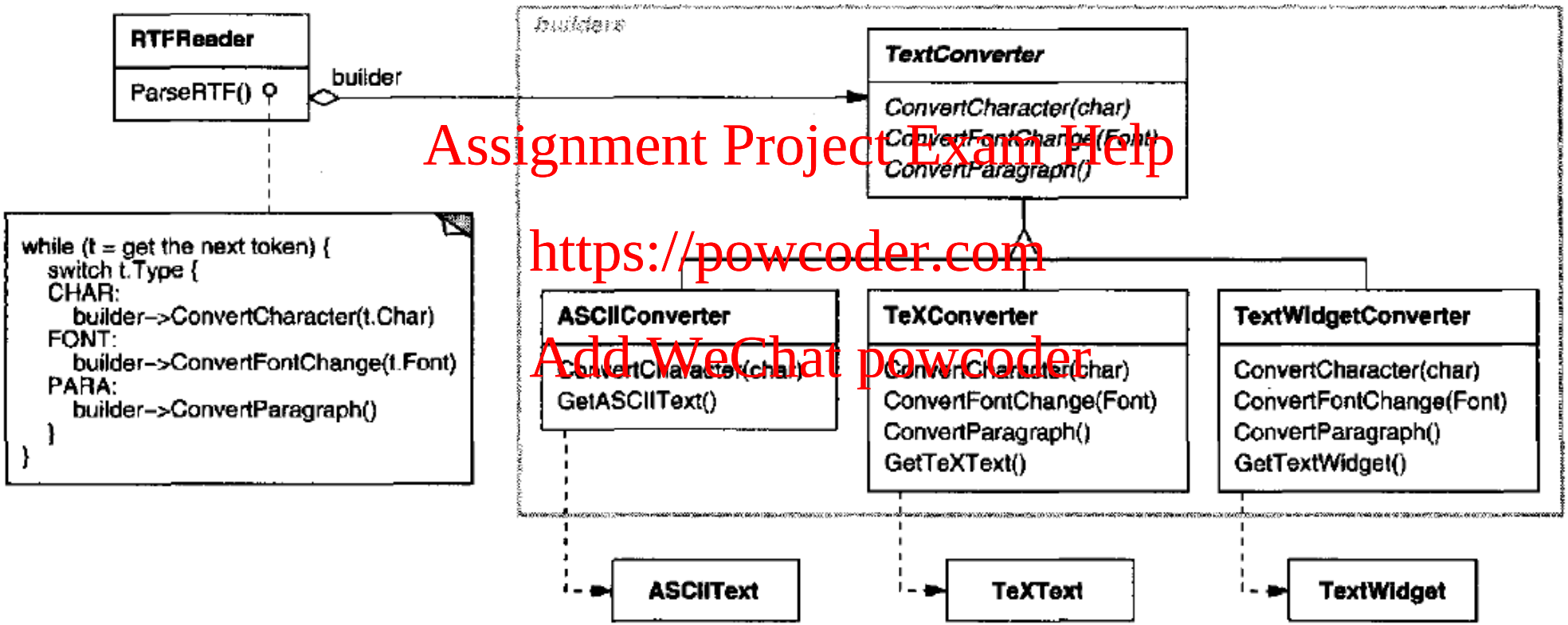
Builder – Participants

- **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

Builder – Rich Text Format Example



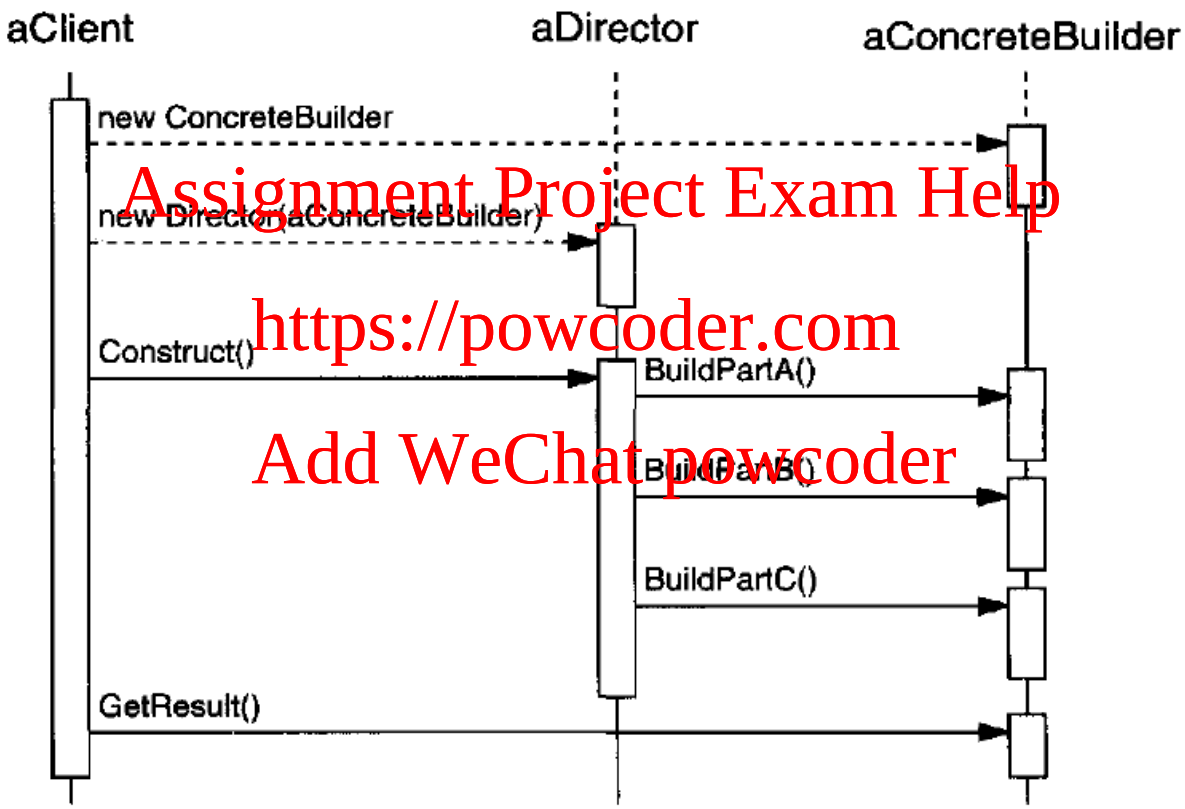
Builder – Rich Text Format Example

- A reader for the RTF (Rich Text Form) document exchange e format should be able to convert RTF to many text formats
- Problem: the number of possible conversions is open-ended, so it should be easy to add a new conversion without modifying the reader
- A solution is to configure the *RTFReader* class with a *TextConverter* object that converts RTF to another textual representation
- Subclasses of TextConverter specialize in different conversions and formats
- Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface

Builder – Rich Text Format Example

- Builder Pattern captures all the relationships
  - Each converter class is called a builder in the pattern, and the reader is called the director
  - It separates the algorithm for interpreting a textual format from how a converted format gets created and represented
  - The RTFReader's parsing algorithm can be reused to create different text representations from RTF documents – just configure the RTFReader with different subclasses of TextConverter

Builder – Collaboration



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
    - E.g., SGMLReader uses the same TextConverters to generate different formats (ASCIIText, TextWidget and TexXText)

Builder – Consequences (2)

- Finer control over the construction process
  - The builder pattern constructs the product step by step under the directors 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 Sunday
- Prepare questions and ask during tutor Q&A session this week
- Well organize time and start assignment 2 once it is released today
- Attend Helpdesk session if you have any questions/difficulties on implementation perspective

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.