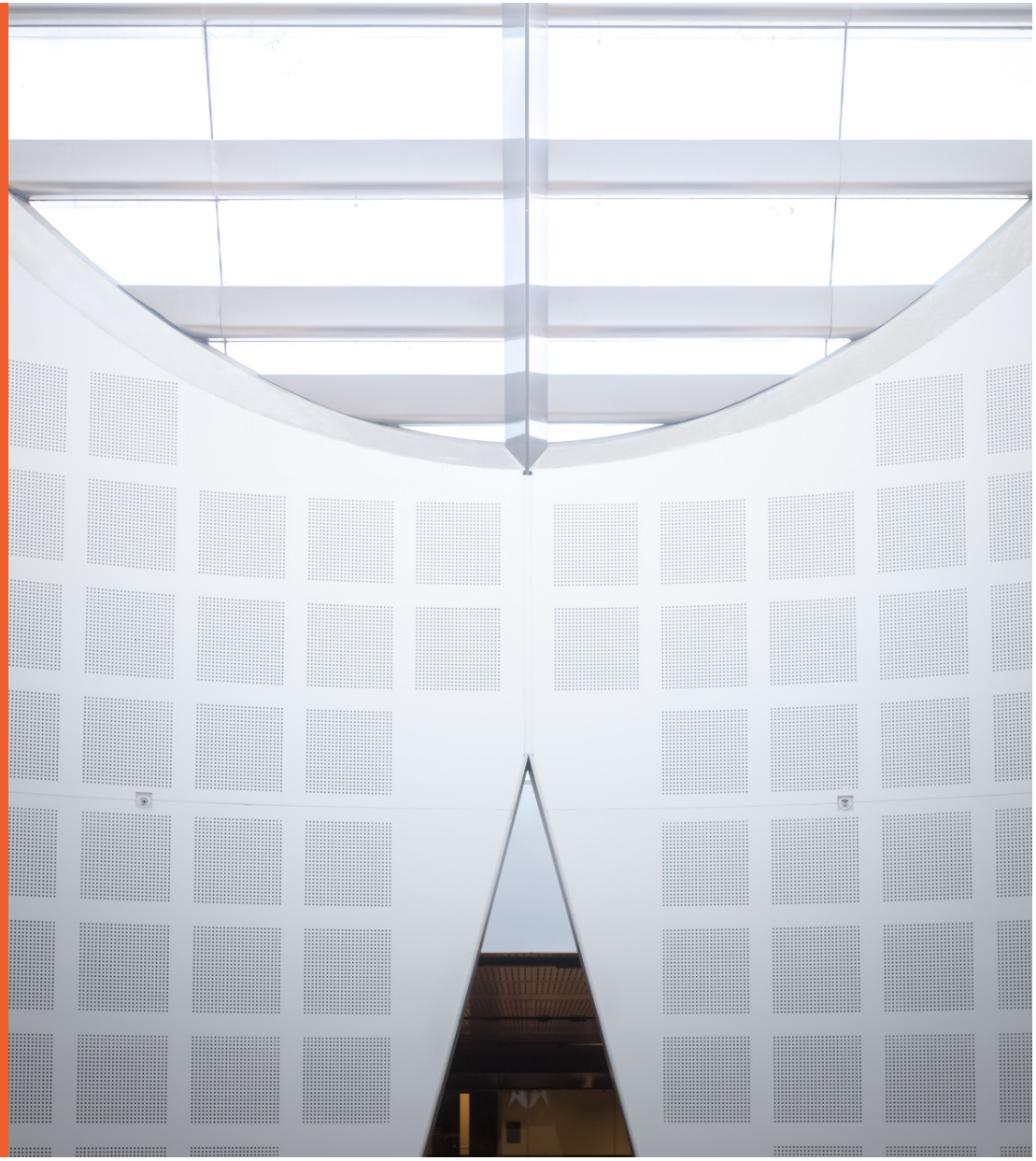
Software Design and Construction 2
SOFT3202 / COMP9202
Review of Design Patterns

Prof Bernhard Scholz

School of Computer Science

THE UNIVERSITY OF SYDNEY



Agenda

- OO Principles
- Design Principles
- Overview of Design Patterns
- GoF Design Patterns (review)

OO Principles

Review

Slides from SOFT2201 by Bernhard Scholz

THE UNIVERSITY OF SYDNEY

OO Principles

- Encapsulation
- Inheritance
- Variable Binding & Polymorphism
- Virtual Dispatch
- Abstract Classes
- Interfaces

Encapsulation

- Wrap data/state and methods into a class as a single unit
- Multiple instances can be generated
- Protect state via setter/getter methods

```
class Rectangle {
  double length; double width;
  double getLength() { return length; }
  double getWidth() { return width; }
  double area() { return length * width; }
}
```

Inheritance

- Sub-class inherits from super-class: methods, variables, …
- Reuse structure and behavior from super-class
- Single inheritance for classes

```
class Rectangle extends Object {
  double length; double width;
  double area() { return length * width; }
}

class ColouredRectangle extends Rectangle {
  int colour;
  int colour() { return colour; }
}
```

Variable Binding, Polymorphism

- Object (on heap) has single type when created
  - type cannot change throughout its lifetime
- Reference Variables point to null or an object
- Type of object and type of reference variable may differ
  - E.g Shape x = new Rectangle(4,2);
- Understand the difference
  - Runtime type vs compile-time type
- Polymorphism:
  - reference variable may reference differently typed object
  - Must be a sub-type of

Virtual Dispatch

- Methods in Java permit a late binding
- Methods in sub-class override methods of super classes
- Virtual dispatch selects which method to invoke

```
public class Shape extends Object {
  double area() { }  }
public class Rectangle extends Shape {
  double area() { } }
…
Shape X = new Shape();
Shape Y = new Rectangle();
double a1 = X.area() // invokes area of Shape
double a2 = Y.area() // invokes area of Rectangle
```

Abstract Classes

- Method implementations are deferred to sub-classes
- Requires own key-word abstract for class/method
- No instance of an abstract class can be generated

```
abstract class Shape extends Object {
  public abstract double area();
}

public class Rectangle extends Shape {
  double width; double length;
  double area() { return width * length; }
}
```

Interfaces

- Java has no multi-inheritance
  - Interface is a way-out
  - introduction of multi-inheritance via the back-door
- Interfaces is a class contract
  - implements a set of methods.
  - defines set of behavior
- Interfaces can inherit from other interfaces
  - But ***no cyclic*** inheritance of interfaces
- Methods declared in an interface are always public and abstract
- Variables are permitted if they are static and final only

Example: Interface

- Inheritance in interfaces

```
// definition of interface
public interface A {
  int foo(int x);
}

public interface B extends A{
  int hoo(int x);
}
```

- Interface B has methods foo() and hoo()

OO Design Principles

Revisit

.

THE UNIVERSITY OF SYDNEY

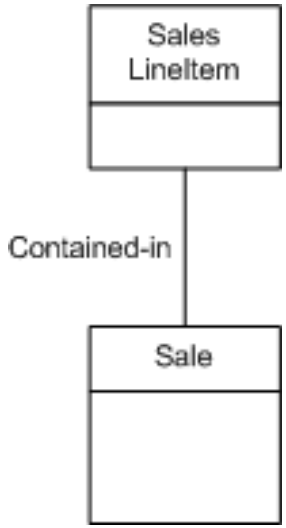
GRASP: Methodological Approach to OO Design

- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Guidelines for assigning responsibility to classes and objects
- Document and standardize principles in OOD
- The five basic principles:
  1. Creator
  2. Information Expert
  3. High Cohesion
  4. Low Coupling
  5. Controller

GRASP: Creator Principle

- Problem
  - Who creates an object A in a system?

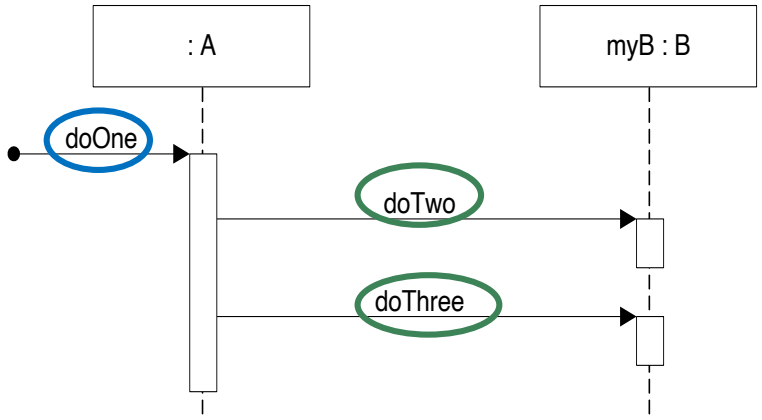
- Solution
  - Assign class B the responsibility to create an instance of class A when
    - B "contains" A
    - B "records" A
    - B "closely uses" A
    - B "has the Initializing data for" A



GRASP: Information Expert Principle

- Problem
  - When should we delegate responsibilities of methods, attributes, and so on.

- Solution
  - Assign responsibility by determining the information required of responsibility
    - where is it stored?
  - Placing the responsibility on the class with the most information available



Cohesion

- How strongly related and focused the responsibilities of a single class?

- How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Assign responsibilities so that cohesion remains high
  - If necessary, break classes apart

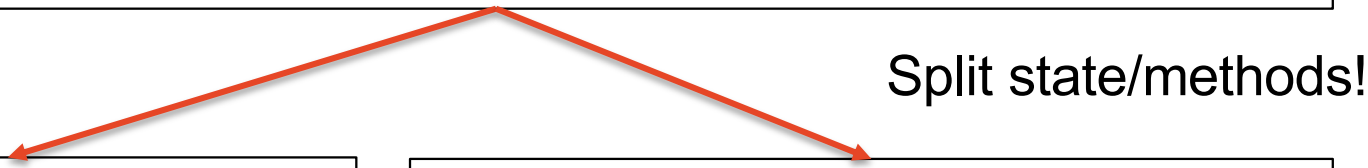
High Cohesion

- Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Solution
  - Assign responsibilities so that cohesion remains high

Example: Cohesion

```
class DateLocation {
  Date date;                    // no cohesion between
  Location location;            // date & location
  Date incDate() { return date+1; }
  Location incLocation() { return location+1; }
}
```

Bad Cohesion!



Split state/methods!

```
class Date {
  Date date;
  Date incDate() {
   return date+1; }
}
```

```
class Location {
  Location location;
  Location incLocation() {
   return location+1; }
}
```

Good Cohesion!

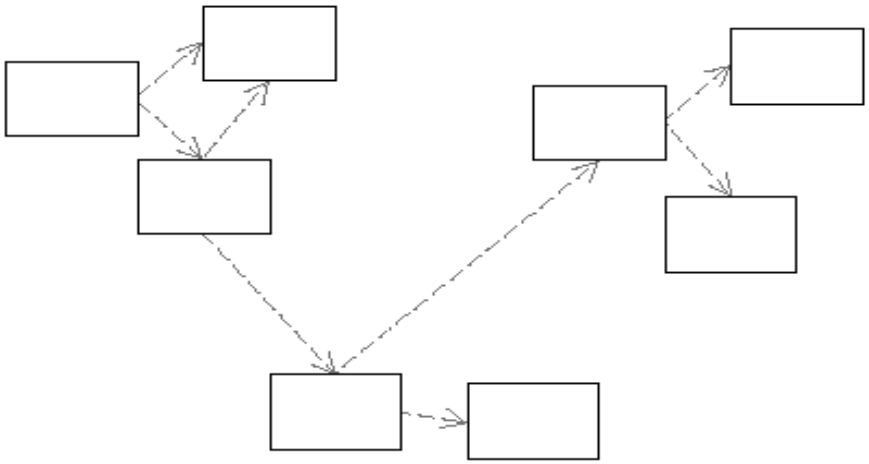
Dependency

- A dependency exists between two classes if a change to one causes a change to the other

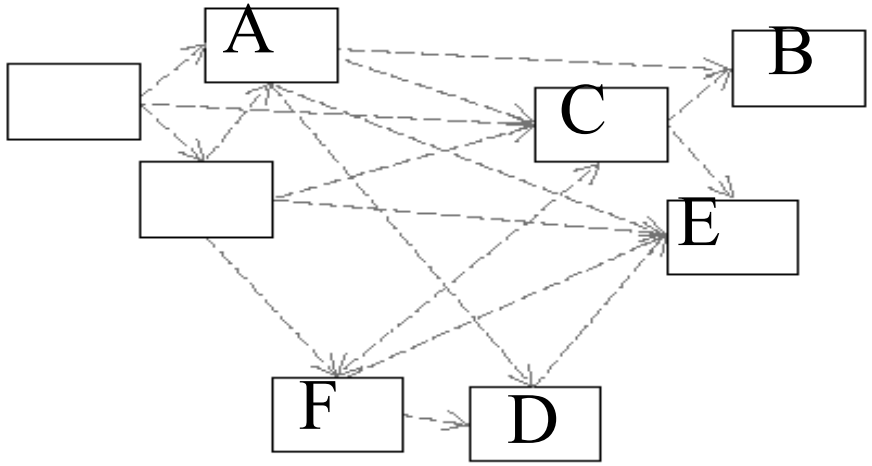
- Various reason for dependency
  - Class send message to another
  - One class has another as its data
  - One class mention another as a parameter to an operation
  - One class is a superclass or interface of another

Coupling

- How strongly one element is connected to, has knowledge of, or depends on other elements?
- Dependencies in UML class diagram:



Low coupling



High coupling

GRASP: Low Coupling Principle

- Problem
  - How to reduce the impact of change, to support low dependency, and increase reuse?

- Solution
  - Assign responsibility so that coupling remains low

Coupling and Cohesion

- Coupling describes the inter-objects relationship
- Cohesion describes the intra-object relationship
- Extreme case of “coupling”
  - Only one class for the whole system
  - No coupling at all
  - Extremely low cohesion
- Extreme case of cohesion
  - Separate even a single concept into several classes
  - Very high cohesion
  - Extremely high coupling
- Domain model helps to identify concepts
- OOD helps to assign responsibilities to proper concepts

Design Patterns Review

Revisit

Slides from SOFT2201

THE UNIVERSITY OF SYDNEY

Catalogue of Design Patterns

Design Pattern	Description
Gang of Four (Gof)	First and most used. Fundamental patterns OO development, but not specifically for enterprise software development.
Enterprise Application Architecture (EAA)	Layered application architecture with focus on domain logic, web, database interaction and concurrency and distribution. Database patterns (object-relational mapping issues)
Enterprise Integration	Integrating enterprise applications using effective messaging models
Core J2EE	EAA Focused on J2EE platform. Applicable to other platforms
Microsoft Enterprise Solution	MS enterprise software patterns. Web, deployment and distributed systems

https://martinfowler.com/articles/enterprisePatterns.html

Aspects of Enterprise Software

- Domain Logic
  - Business rules, validations and computations
  - Some systems intrinsically have complex domain logic
  - Regular changes as business conditions change
- EAA Patterns
  - organizing domain logic
- Data Model Patterns
  - Data modeling approach with examples of domains

SOFT3202 / COMP9202

- GoF Patterns
  - Flyweight, Bridge, Chain of Responsibility
- Concurrency
  - Lock, Thread Pool
- Enterprise
  - Lazy load, Value Object, Unit of Work (MVC, SOA)

Review of GoF Design Patterns

THE UNIVERSITY OF SYDNEY

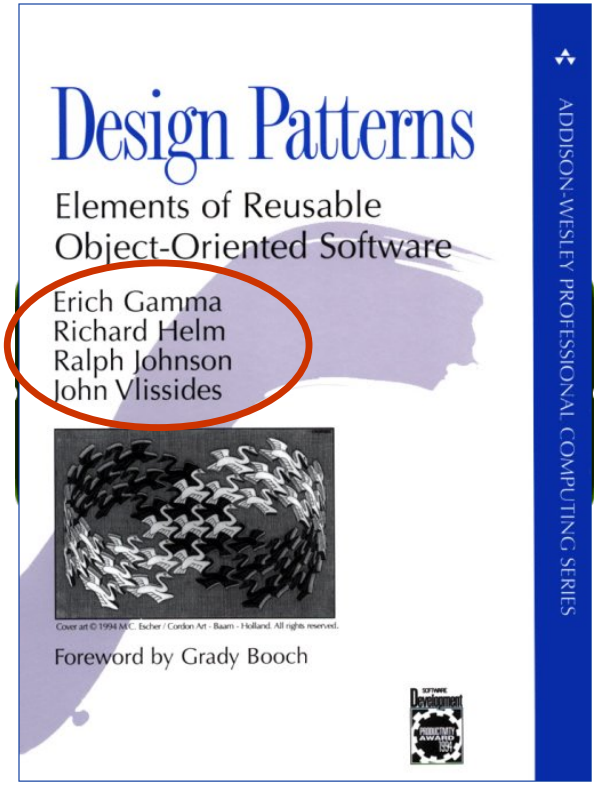
Design Patterns

- Proven solutions to general design problems which can be applied to specific applications
- Not readily coded solution, but rather the solution path to a common programming problem
- Design or implementation **structure** that achieves a particular purpose
- Allow evolution and change
  - Vary independently of other components
- Provide a shared language among development communities – effective communication

Elements of a Pattern

- The **pattern name**
- The **problem**
  - When to apply the pattern
- The **solution**
  - The elements that make up the design
- **Consequence**
  - The results and trade-offs of applying the pattern

Gang of Four Patterns (GoF)



- Official design pattern reference
- Famous and influential book about design patterns
- GoF Design Patterns → Design Patterns

Design Patterns – Classification

Scope / Purpose	Creational	Structural	Behavioral
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

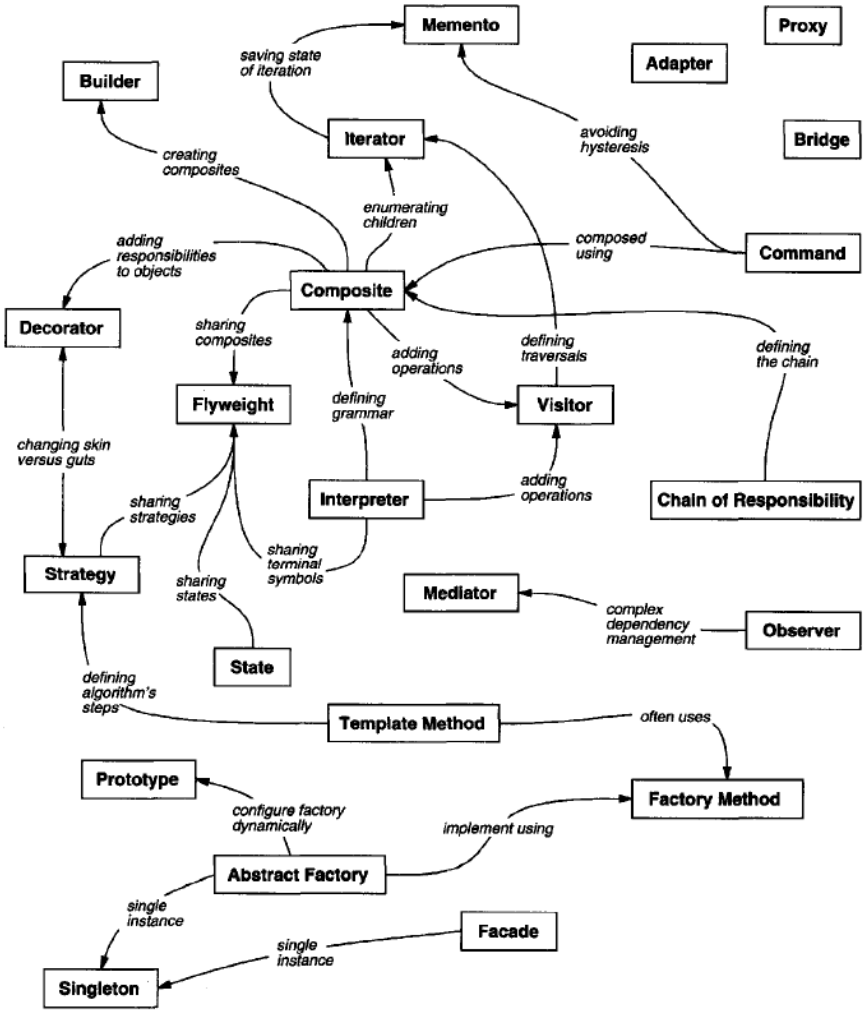
Design Patterns – Classification

Describes of 23 design patterns

- **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 – Different Classification (2)

Design patterns classification based on relationships



Selecting Appropriate Design Pattern

- Consider how design pattern solve a problem
- Read through Each pattern's intent to find relevant ones
- Study the relationship between design patterns
- Study patterns of like purpose (similarities and differences)
- Examine a cause of redesign (what might force a change to a design? Tight-coupling, difficult to change classes
- Consider what should be variable in your design (see table in next slides)

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
Behavioural	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
- Becomes more important as systems evolve to depend on object composition than class inheritance
- Provides flexibility in *what gets created*, *who creates it*, *how it gets created* and *when*
  - Let you configure a system with "product" objects that vary in structure and functionality

Creational Patterns

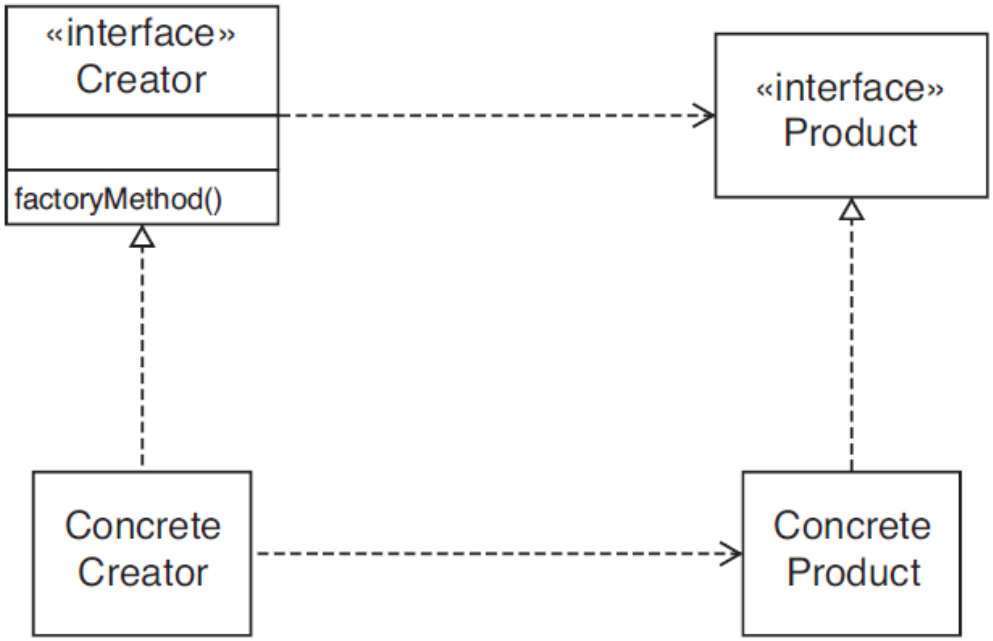
Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Singleton	Ensure a class only has one instance, and provide global point of access to it
Factory Method	Define an interface for creating an object, 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

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Factory Method

- 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
- 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

Factory Method Pattern – Structure



Factory Method Pattern – 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. Creator may also define a default implementation of the factory method that returns a default *ConcreteProduct* object
  - May call the factory method to create a Product object
- **ConcreteCreator**
  - Overrides the factory method to return an instance of a Concrete Product

Other 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

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Structural Patterns

THE UNIVERSITY OF SYDNEY

Structural Patterns

- How classes and objects are composed to form larger structures
- Structural *class* patterns use inheritance to compose interfaces or implementations
- Structural *object* patterns describe ways to compose objects to realize new functionality
  - The flexibility of object composition comes from the ability to change the composition at run-time

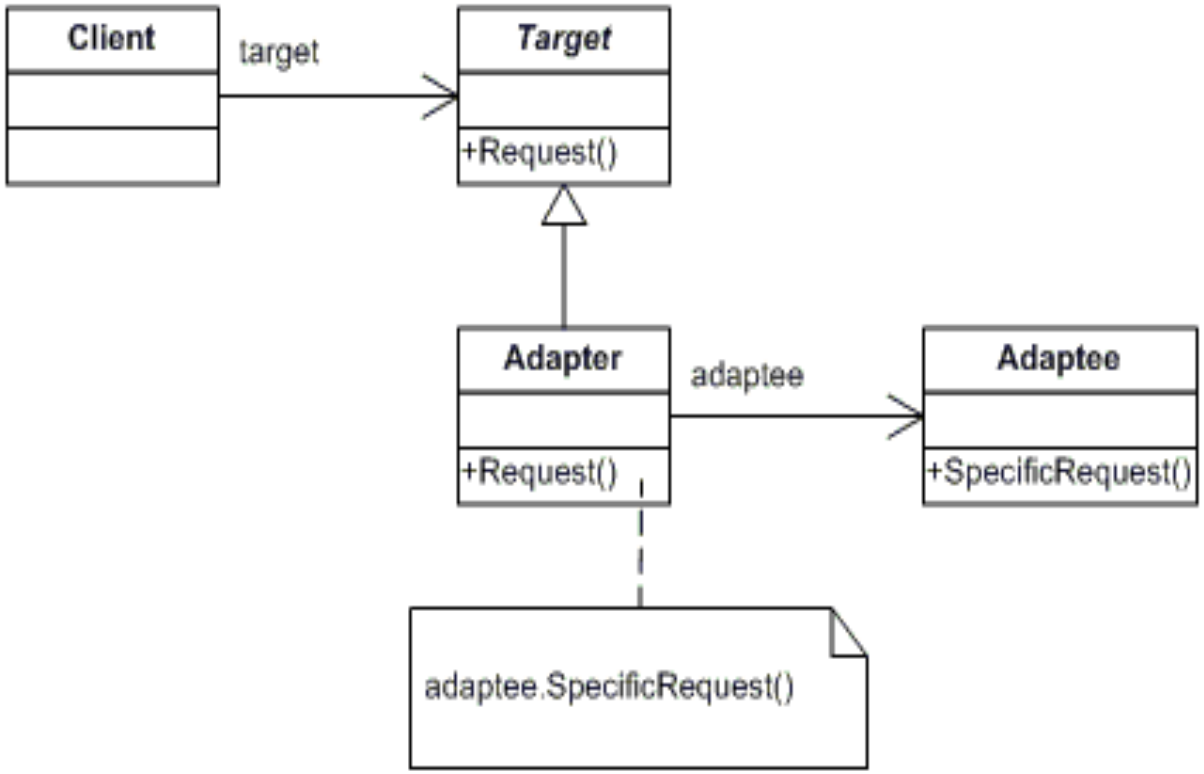
Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.**
Façade	**Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.**
Composite	Compose objects into tree structures to represents part-whole hierarchies. Let client treat individual objects and compositions of objects uniformly
Proxy	Provide a placeholder for another object to control access to it
Decorator	**Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)**
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Flight weight	Use sharing to support large numbers of fine-grained objects efficiently

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect
  - Classes work together that couldn't otherwise because of incompatible interfaces
- Applicability
  - To use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, i.e., classes that don't necessarily have compatible interfaces
  - **Object adapter only** to use several existing subclasses, but it's unpractical to adapt their interface by sub-classing everyone. An object adapter can adapt the interface of its parent class.

Object Adapter – Structure



Adapter – Participants

- **Target**
  - Defines the domain-specific interface that Client uses
- **Client**
  - Collaborates with objects conforming to the Target interface
- **Adaptee**
  - Defines an existing interface that needs adapting
- **Adapter**
  - Adapts the interface of Adaptee to the Target interface
- **Collaborations**
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee's operations that carry out the request

Structural Patterns (GoF)

Pattern Name	Description
Adapter	**Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.**
Façade	**Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.**
Composite	Compose objects into tree structures to represents part-whole hierarchies. Let client treat individual objects and compositions of objects uniformly
Proxy	Provide a placeholder for another object to control access to it
Decorator	**Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)**
Bridge	Decouple an abstraction from its implementation so that the two can vary independently
Flight weight	Use sharing to support large numbers of fine-grained objects efficiently

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Behavioural Design Patterns

THE UNIVERSITY OF SYDNEY

Behavioural Patterns

- Concerned with algorithms and the assignment of responsibilities between objects
- Describe patterns of objects and class, and communication between them
- Simplify complex control flow that's difficult to follow at run-time
  - Concentrate on the ways objects are interconnected
- **Behavioural Class Patterns**
  - Use *inheritance* to distribute behavior between classes (algorithms and computation)
- **Behavioural Object Patterns**
  - Use *object composition*, rather inheritance. E.g., describing how group of peer objects cooperate to perform a task that no single object can carry out by itself

Behavioural Patterns (GoF)

Pattern Name	Description
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable (let algorithm vary independently from clients that use it)
Observer	Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

Behavioural Patterns (GoF)

Pattern Name	Description
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
State	Allow an object to alter its behaviour when its internal state changes. The object will appear to change to its class
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates
Other patterns; Chain of responsibility, Command, Mediator, Template Method	

See Additional Review Slides: https://canvas.sydney.edu.au/courses/14614/pages/lecture-review-of-design-patterns?module_item_id=437271

Strategy Pattern

Object behavioural

THE UNIVERSITY OF SYDNEY

Strategy

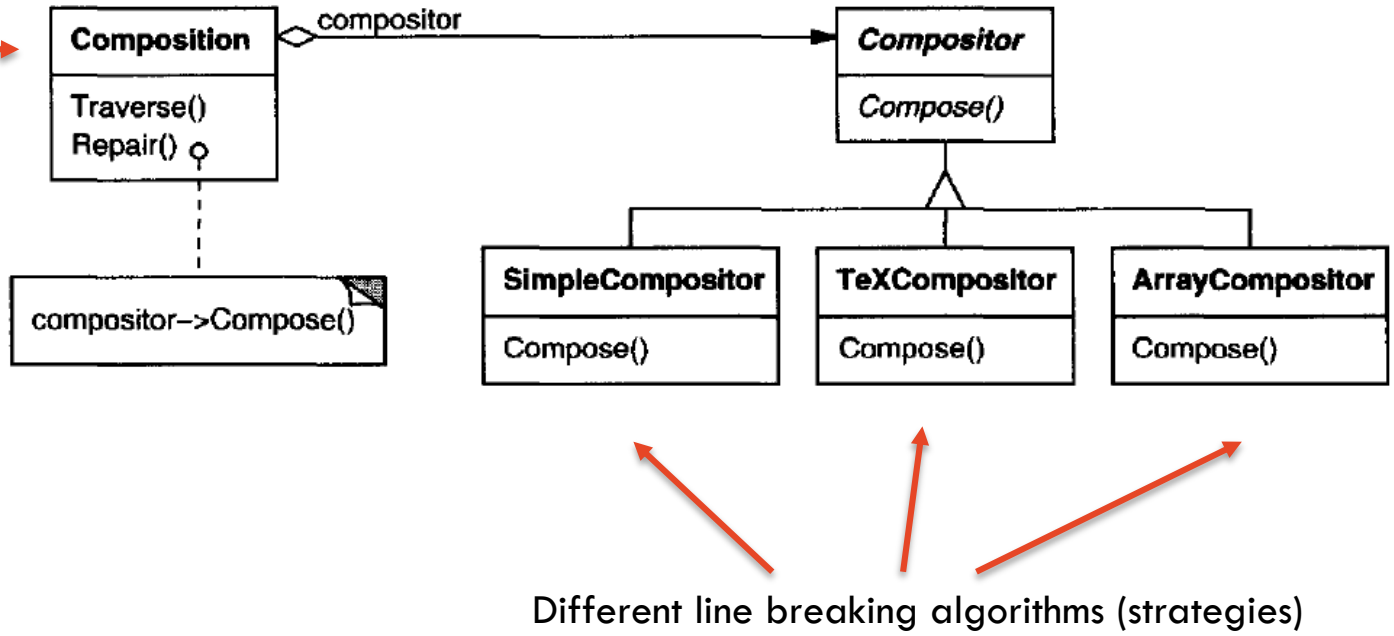
- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from clients that use it
- **Known as**
  - Policy
- **Motivation**
  - Design for varying but related algorithms
  - Ability to change these algorithms

Strategy – Example (Text Viewer)

- Many algorithms for breaking a stream of text into lines
- Problem: hard-wiring all such algorithms into the classes that require them
  - More complex and harder to maintain clients (more line breaking algorithms)
  - Not all algorithms will be needed at all times

Strategy – Example (Text Viewer)

Maintain & update the line breaks of text

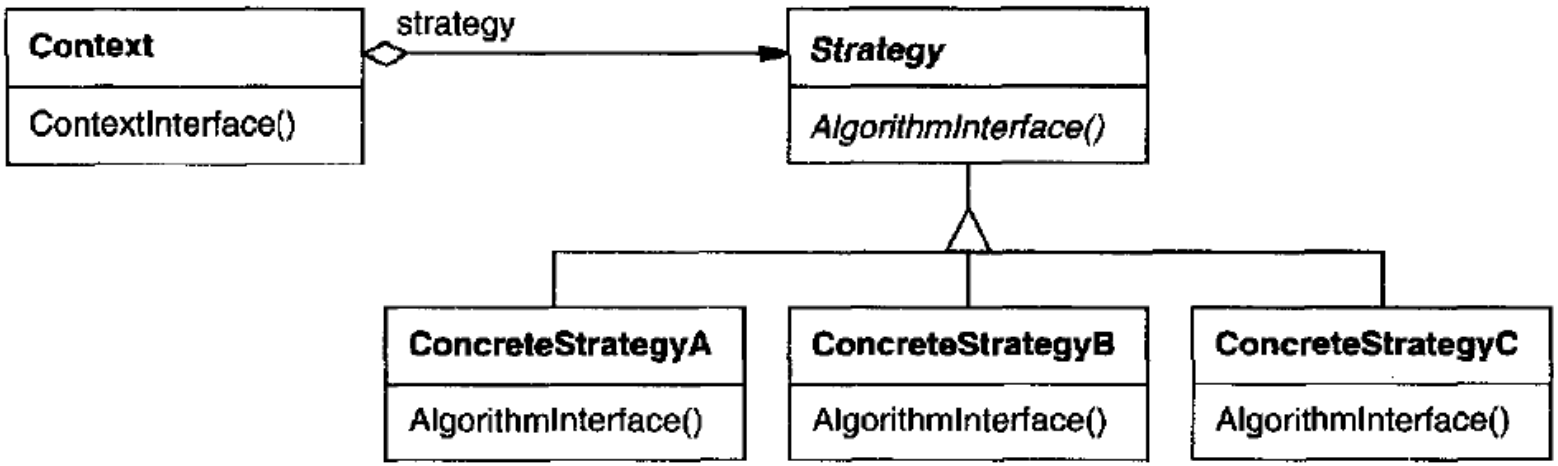


Different line breaking algorithms (strategies)

Strategy – Applicability

- Many related classes differ only in their *behavior*
- You need different *variant of an algorithm*
- An algorithm uses *data that should be hidden from its clients*
- A class defines many behaviors that appear as multiple statements in its operations

Strategy – Structure



Strategy – Participants

Participant	Goals
Strategy (Compositor)	Declares an interface common to all supported algorithms Used by context to call the algorithm defined by ConcereteStrategy
ConcereteStrategy (SimpleCompositor, TeXCompositor, etc)	Implements the algorithm using the Strategy interface
Context (Compositoion)	Is configured with a ConcereteStrategy object Maintains a reference to a Strategy object May define an interface that lets Strategy access its data

Strategy – Collaborations

- Strategy and Context interact to implement the chosen algorithm
  - A context may pass all data required by the algorithm to the Strategy
  - The context can pass itself as an argument to Strategy operations

- A context forwards requests from its clients to its strategy
  - Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively

References

- 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.

- 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.

- Martin Folwer, Patterns In Enterprise Software, [https://martinfowler.com/articles/enterprisePatterns.html]