

Design patterns in C++



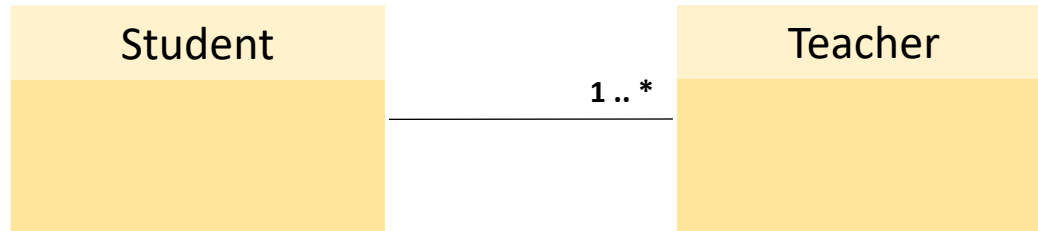
Prerequisites

Assumes you are reasonably proficient in at least one object-oriented programming language, and you should have some experience in object-oriented design as well



Design patterns in C++

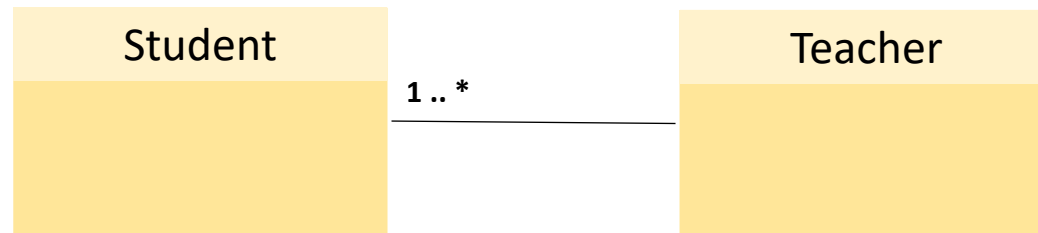
Association . Aggregation . Composition . Inheritance . Generalization . Specialization



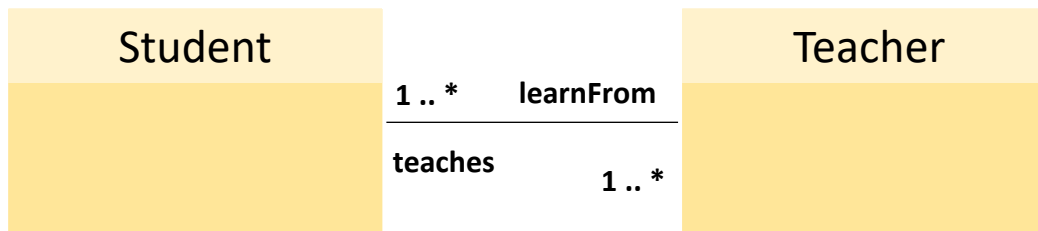
```
class Student
{
    std::vector<Teacher>
    m_teachers;
};
```

If two classes are communicate with each other, then they are associated

- Can be represented with line between these classes
- Also can have multiplicity of an association



```
class Teacher
{
    std::vector<Student>
    m_teachers;
};
```



```
class Teacher
{
    std::vector<Student>
    m_teachers;
public:
    void learFrom()
    {
    }
};
```

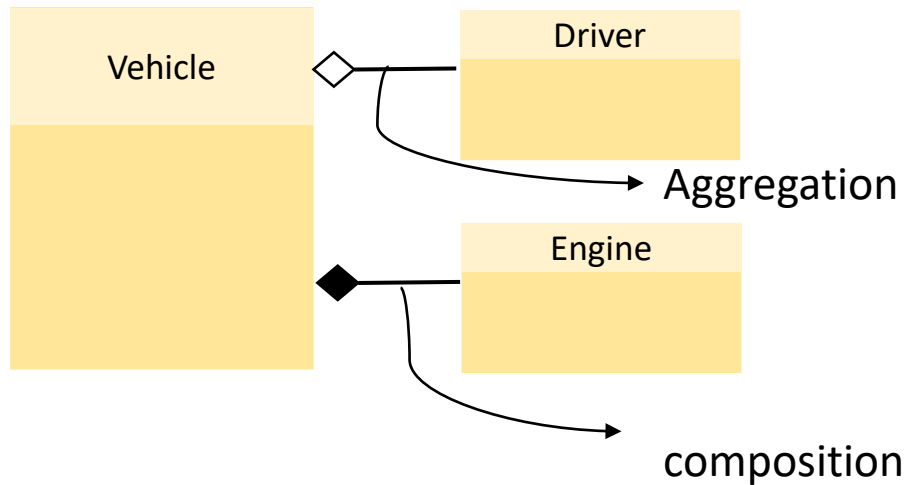
```
class Student
{
    std::vector<Teacher>
    m_teachers;
public:
    void teaches()
    {
    }
};
```

E - GRASP

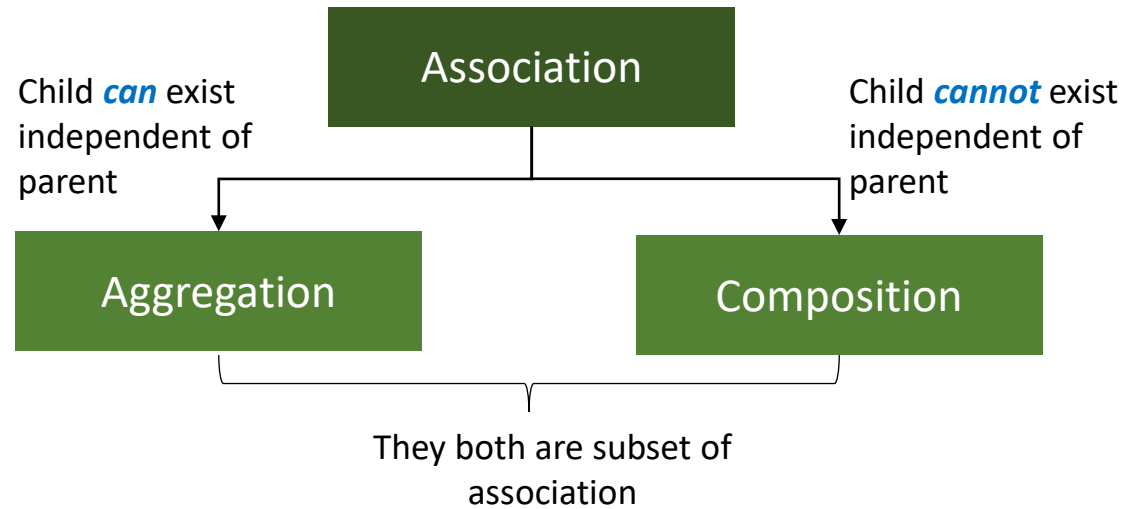


Design patterns in C++

Association . Aggregation . Composition . Inheritance . Generalization . Specialization



```
class Vehicle
{
private:
    std::shared_ptr<Driver> m_pDriver
= nullptr;
    Engine m_engine;
};
```



E - GRASP

Mastering Embedded Application Development



Design patterns in C++

Association . Aggregation . Composition . Inheritance . Generalization



Design patterns in C++

Association . Aggregation . Composition . Inheritance . Generalization



What you feel after learning design patterns

- You won't ever think about object-oriented design in the same way as you were thinking earlier
- You'll have insights that can make your own designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place

Remember that this isn't a material to read once and throw away. We hope you'll find yourself referring to it again and again for design insights and for inspiration.



This material has two main parts

- The first part **[Part A]** describes what design patterns are and how they help you design object-oriented software. It includes a design case study that demonstrates how design patterns apply in practice.
- The second part **[Part B]** of the book is a catalog of the actual design patterns. The catalog makes up the majority of the material. Its chapters divide the design patterns into three types: **creational**, **structural**, and **behavioral**



Design patterns in C++

Continue...

If you aren't an experienced object-oriented designer, then start with the simplest and most common patterns:

- Abstract Factory
- Adapter
- Composite
- Decorator
- Factory Method
- Observer
- Strategy
- Template Method



Introduction

- Reusable designing object-oriented software is **hard**, and designing object-oriented software is even **harder**
- It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?
- One thing expert designers know **not** to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts.
- A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.



Design patterns in C++

Continue...

- Design patterns make it easier to reuse successful designs and architectures.
- Expressing proven techniques as design patterns makes them more accessible to developers of new systems.
- Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.
- Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent
- Put simply, design patterns help a designer get a design "right" faster.



What this material does not covers ?

- It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming
- It doesn't have any application domain-specific patterns
- It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database.
- Each of these areas has its own patterns, and it would be worth while for someone to catalog those too.



What is a Design Pattern?

Christopher Alexander says “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

In general, a pattern has four essential elements:

1. The pattern name:
2. The problem
3. The solution
4. The consequences



Design patterns in C++

The pattern name

- Used to describe a design problem, its solutions, and consequences in a word or two
- Naming a pattern immediately increases our design vocabulary
- It lets us design at a higher level of abstraction
- Having a vocabulary for patterns lets us talk about them with our colleagues, in our documentation, and even to ourselves



Design patterns in C++

The problem

- Describes when to apply the pattern
- It explains the problem and its context
- It might describe specific design problems such as how to represent algorithms as objects
- It might describe class or object structures that are symptomatic of an inflexible design
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern



The solution

- Describes the elements that make up the design, their relationships, responsibilities, and collaborations
- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it



The consequences

- These are the results and trade-offs of applying the pattern.
- Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
- The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.



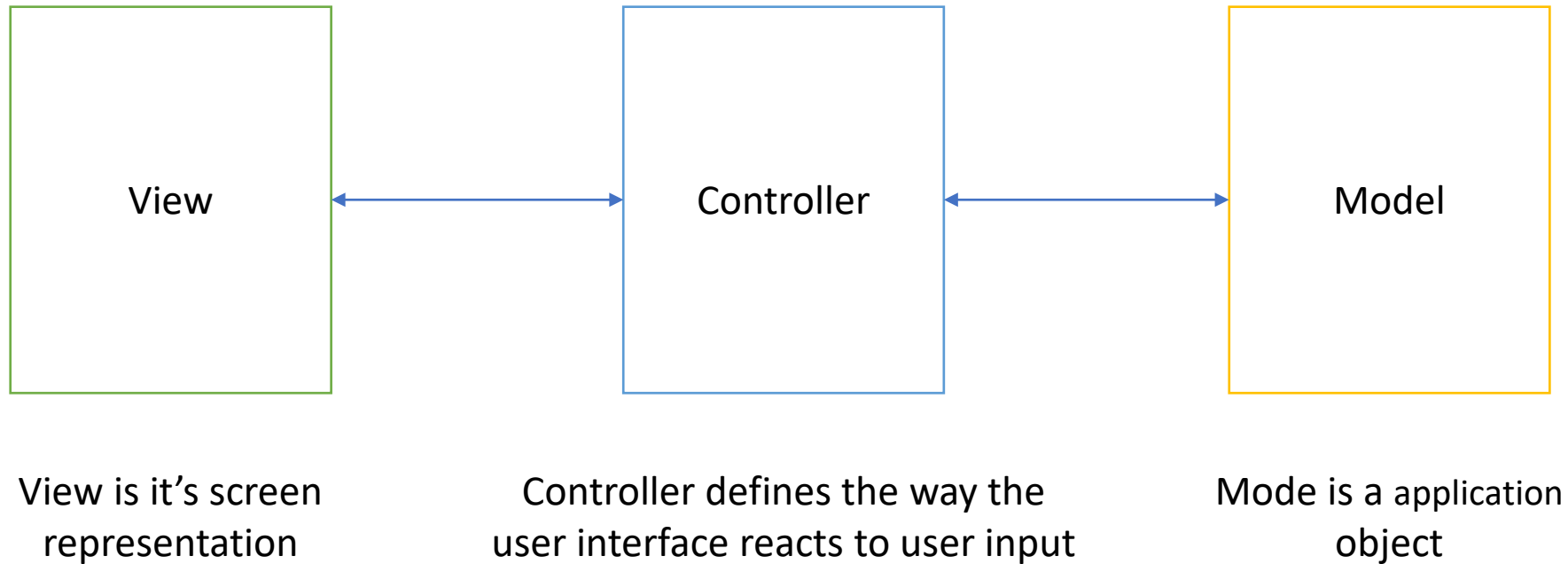
Design patterns in C++

Alert !

- Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block.
- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem
- The design patterns are descriptions of **communicating objects and classes that are customized to solve a general design problem in a particular context**



Design pattern in MVC



Design patterns in C++

Design pattern in MVC

Before MVC, user interface designs tended to **lump** these objects together. MVC **decouples** them to increase flexibility and reuse.

This example reflects the design that decouples views from models. But the design is applicable to a more general problem: **decoupling objects** so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the **Observer design pattern**

Another feature of MVC is that **views can be nested**. For example, a control panel of buttons might be implemented as a complex view containing nested button views. MVC supports nested views with the **CompositeView** class, a subclass of **View**. **CompositeView** objects act just like **View** objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs **whenever we want to group objects and treat the group like an individual object**. This more general design is described by the **Composite design pattern**.



Design patterns in C++

Continued...

A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events. The View-Controller relationship is an example of the Strategy design pattern


A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

MVC uses other design patterns, such as Factory Method to specify the default controller class for a view and Decorator to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.




Describing Design Patterns

How do we describe design patterns?

Using Graphical notations ? 

- Yes important and useful, but are not sufficient – They simply capture the end product of the design process as a relationship between classes and objects

To reuse the design, we must also record the **decisions**, **alternatives**, and **trade-offs** that led to it. 

Concrete **examples** are important too, because they help you see the design in action

We describe design patterns using a **consistent format**. Each pattern is divided into **sections** according to the template

The template lends a uniform structure to the information, making design patterns **easier to learn, compare, and use**.



Template

- Pattern Name and Classification
- Intent
- Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns



Design patterns in C++

Continued...

Pattern Name and Classification

A good name is vital, because it will become part of your design vocabulary.

The pattern's classification reflects the scheme we introduce in Section

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique. We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.

E - G R A S P

Mastering Embedded Application Development



Design patterns in C++

Continued...

Participants

The classes and/or objects participating in the design pattern and their responsibilities

Collaborations

How the participants collaborate to carry out their responsibilities

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

E - G R A S P

Mastering Embedded Application Development



Design patterns in C++

Continued...

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?



The catalog of Design Patterns

Abstract Factory (1)

Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Adapter (2)

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Bridge (3)

Decouple an abstraction from its implementation so that the two can vary independently

Builder (4)

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Chain of Responsibility (5)

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it

Command (6)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Design patterns in C++

Continued...

Composite (7)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (8)

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (9)

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Factory Method (10)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Flyweight (11)

Use sharing to support large numbers of fine-grained objects efficiently.

Interpreter (12)

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Design patterns in C++

Continued...

Iterator (13)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (14)

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento (15)

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (16)

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Prototype (17)

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Proxy (18)

Provide a surrogate or placeholder for another object to control access to it.

E - G R A S P

Mastering Embedded Application Development



Design patterns in C++

Continued...

Singleton (19)

Ensure a class only has one instance, and provide a global point of access to it.

State (20)

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (21)

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (22)

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (23)

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.



Organizing the catalog

The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria

- 1. Purpose:** Reflects what a pattern does. Patterns can have either **creational**, **structural**, or **behavioral** purpose
 - *Creational: Concern the process of object creation*
 - *Structural: Deals with the composition of classes or objects*
 - *Behavioral: Characterize the ways in which classes or objects interact and distribute responsibility*
- 2. Scope:** specifies whether the pattern applies primarily to **classes** or to **objects**.
 - *Class patterns: Deals with **relationships** between **classes and their subclasses**. These relationships are established through inheritance, or they are static—fixed at **compile-time**.*
 - *Object patterns: Deals with **object relationships**, which can be changed at **run-time** and are more dynamic*

Note: Most patterns are in object scope



Design patterns in C++

Continued...

	Purpose			
		<i>Creational</i>	<i>Structural</i>	<i>Behavioral</i>
Scope	<i>Class</i>	- Factory Method	- Adapter	- Interpreter - Template Method
	<i>Object</i>	- Abstract Factory - Builder - Prototype - Singleton	- Adapter - Bridge - Composite - Decorator - Façade - Flyweight - Proxy	- Chain of Responsibility - Command - Iterator - Mediator - Memento - Observer - State - Strategy - Visitor

E - G R A S P



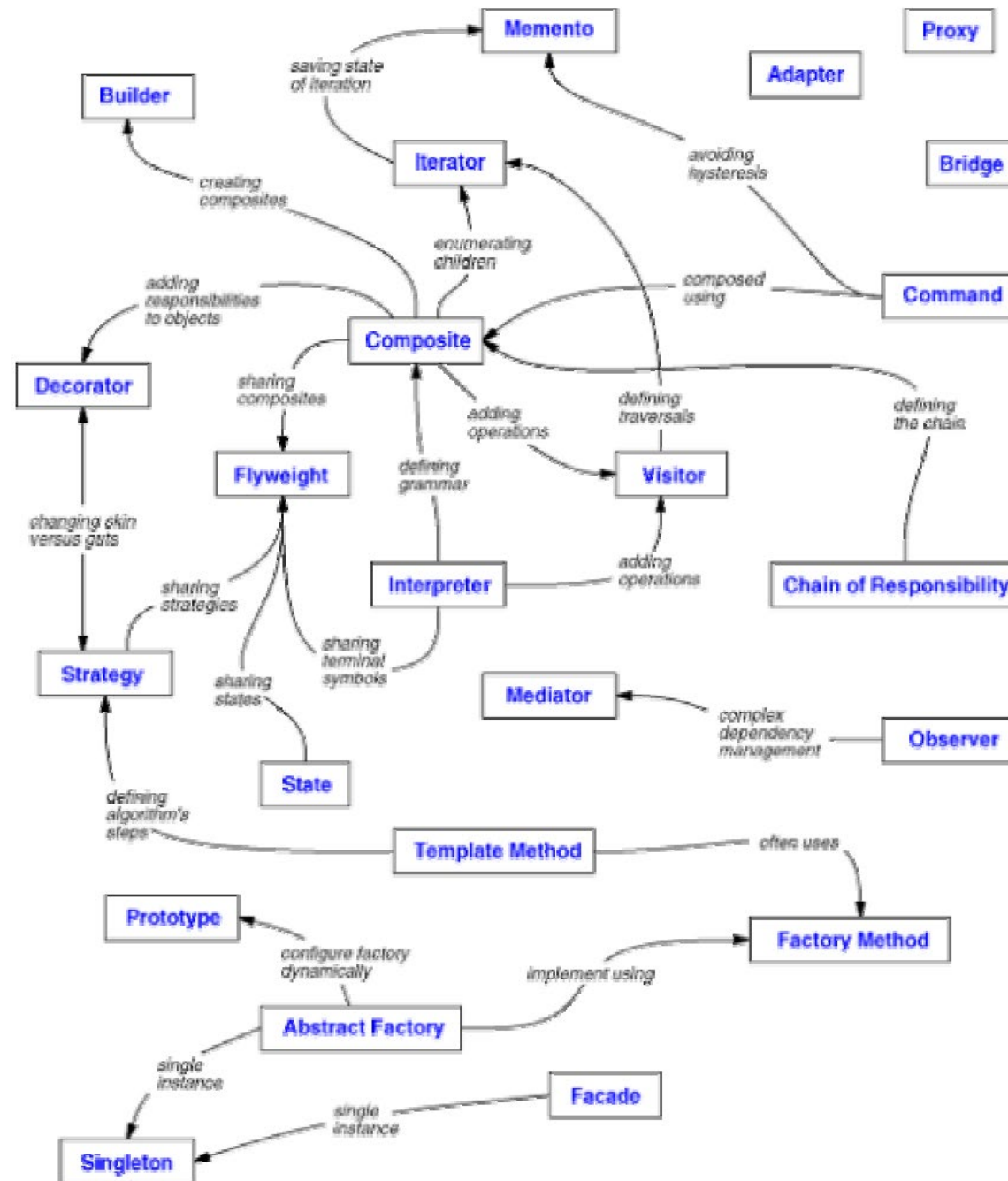
Design patterns in C++

Alert !

- There are other ways to organize the patterns. Some patterns are often used together. For example, Composite is often used with Iterator or Visitor.
- Some patterns are alternatives: Prototype is often an alternative to Abstract Factory.
- Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar
- Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns"



Design patterns in C++



How Design Patterns Solve Design Problems

- *Under construction ...*



How to select design patterns

It might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you

1. **Consider how design patterns solve design problems:** Find appropriate objects, determine object granularity, specify object interfaces etc.
2. **Scan Intent sections:** Read through each pattern's intent to find one or more that sound relevant to your problem
3. **Study how patterns interrelate:** Refer the big picture, which shows relationships between design patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns
4. **Study patterns of like purpose:** Understand more on creational, structural and behavioral design patterns
5. **Examine a cause of redesign:** Look at the causes of redesign starting on [page 37](#) to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign
6. **Consider what should be variable in your design:** This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns



Design patterns in C++

Continued...

Below are the lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

Purpose	Design Pattern	Aspect(s) that can vary
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

Purpose	Design Pattern	Aspect(s) that can vary
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
	Facade	Interface to a subsystem
	Flyweight	Storage costs of objects
	Proxy	How an object is accessed; its location



Design patterns in C++

Continued...

Purpose	Design Pattern	Aspect(s) that can vary
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 information is stored outside an object, and 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)

E - G R A S P

Mastering Embedded Application Development



How to use a Design Patterns

- *Page 44*

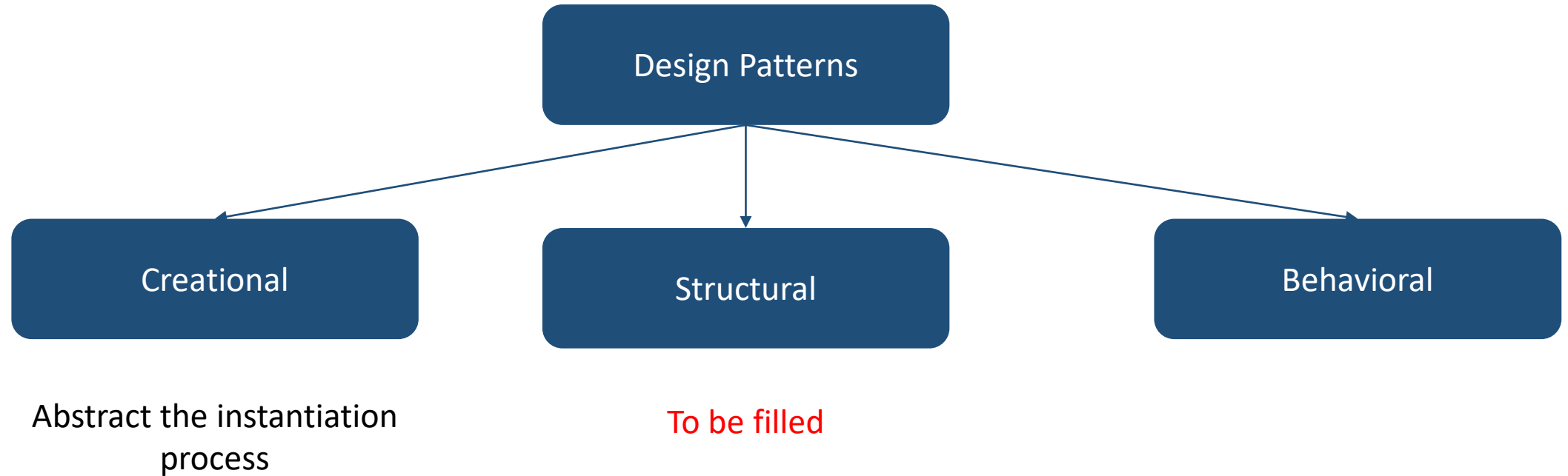


How to use a Design Patterns

- *Page 44*



Design patterns in C++



Design patterns in C++

Creational

Abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented

Class creational pattern

-> Uses inheritance to vary the class that is instantiated

Object creational pattern

-> Delegate instantiation to another object



- They all encapsulate knowledge about which concrete classes the system uses
- They hide how instances of these classes are created and put together



Singleton Design Pattern - Creational

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Motivation

- Although there can be many printers in a system, there should be only one printer spooler
- There should be only one file system and one window manager
- Digital filter will have one A/D converter (ADC)
- An accounting system will be dedicated to serving one company



Singleton Design Pattern – Creational (Continued...)

Applicability

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.



Singleton Design Pattern – Creational (Continued...)

Applicability

Use when

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

