**Data access layer**

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/Data_access_layer#mw-head), [search](https://en.wikipedia.org/wiki/Data_access_layer#p-search)

A **data access layer** (**DAL**) in computer software, is a [layer](https://en.wikipedia.org/wiki/Layer_(object-oriented_design)) of a [computer program](https://en.wikipedia.org/wiki/Computer_program) which provides simplified access to [data](https://en.wikipedia.org/wiki/Data) stored in [persistent storage](https://en.wikipedia.org/wiki/Persistent_storage) of some kind, such as an [entity-relational](https://en.wikipedia.org/wiki/Entity_relationship) [database](https://en.wikipedia.org/wiki/Database). This [acronym](https://en.wikipedia.org/wiki/Acronym) is prevalently used in [Microsoft](https://en.wikipedia.org/wiki/Microsoft) environments.

For example, the DAL might return a reference to an [object](https://en.wikipedia.org/wiki/Object_(computer_science)) (in terms of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming)) complete with its attributes instead of a [row](https://en.wikipedia.org/wiki/Row_(database)) of [fields](https://en.wikipedia.org/wiki/Field_(computer_science)) from a database [table](https://en.wikipedia.org/wiki/Table_(database)). This allows the [client](https://en.wikipedia.org/wiki/Client_(computing)) (or user) modules to be created with a higher level of [abstraction](https://en.wikipedia.org/wiki/Abstraction). This kind of model could be implemented by creating a class of data access methods that directly reference a corresponding set of database stored procedures. Another implementation could potentially retrieve or write records to or from a file system. The DAL hides this complexity of the underlying data store from the external world.

For example, instead of using commands such as *insert*, *delete*, and *update* to access a specific table in a database, a class and a few stored procedures could be created in the database. The procedures would be called from a method inside the class, which would return an object containing the requested values. Or, the insert, delete and update commands could be executed within simple functions like *registeruser* or *loginuser* stored within the data access layer.

Also, business logic methods from an application can be mapped to the Data Access Layer. So, for example, instead of making a query into a database to fetch all users from several tables the application can call a single method from a DAL which abstracts those database calls.

Applications using a data access layer can be either database server dependent or independent. If the data access layer supports multiple database types, the application becomes able to use whatever databases the DAL can talk to. In either circumstance, having a data access layer provides a centralized location for all calls into the database, and thus makes it easier to port the application to other database systems (assuming that 100% of the database interaction is done in the DAL for a given application).

[Object-Relational Mapping](https://en.wikipedia.org/wiki/Object-Relational_Mapping) tools provide data layers in this fashion, following the [active record](https://en.wikipedia.org/wiki/Active_record_pattern) model. The ORM/active-record model is popular with web frameworks.

**Object-relational mapping** (**ORM**, **O/RM**, and **O/R mapping tool**)

in [computer science](https://en.wikipedia.org/wiki/Computer_science) is a [programming](https://en.wikipedia.org/wiki/Computer_programming) technique for converting data between incompatible [type systems](https://en.wikipedia.org/wiki/Type_system) using [object-oriented](https://en.wikipedia.org/wiki/Object-oriented) programming languages. This creates, in effect, a "virtual [object database](https://en.wikipedia.org/wiki/Object_database)" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), [data-management](https://en.wikipedia.org/wiki/Data_management) tasks act on object-oriented (OO) [objects](https://en.wikipedia.org/wiki/Object_(computer_science)) that are almost always non-[scalar](https://en.wikipedia.org/wiki/Scalar_(computing)) values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person [object](https://en.wikipedia.org/wiki/Object_(computer_science))" with [attributes/fields](https://en.wikipedia.org/wiki/Attribute_(computing)) to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address-book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

However, many popular database products such as [SQL](https://en.wikipedia.org/wiki/SQL) database management systems (DBMS) can only store and manipulate [scalar](https://en.wikipedia.org/wiki/Scalar_(computing)) values such as integers and strings organized within [tables](https://en.wikipedia.org/wiki/Table_(database)). The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping implements the first approach.[[1]](https://en.wikipedia.org/wiki/Object-relational_mapping#cite_note-hibernate-orm-overview-1)

The heart of the problem involves translating the logical representation of the objects into an atomized form that is capable of being stored in the database, while preserving the properties of the objects and their relationships so that they can be reloaded as objects when needed. If this storage and retrieval functionality is implemented, the objects are said to be [persistent](https://en.wikipedia.org/wiki/Persistence_(computer_science)).[[1]](https://en.wikipedia.org/wiki/Object-relational_mapping#cite_note-hibernate-orm-overview-1)

**Overview**

Implementation-specific details of storage drivers are generally wrapped in an API in the programming language in use, exposing methods to interact with the storage medium in a way which is simpler and more in line with the paradigms of surrounding code.

The following is a simple example. The below is [C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)) code to deliver a query, itself written in [SQL](https://en.wikipedia.org/wiki/SQL), to a database engine.

String sql = "SELECT ... FROM persons WHERE id = 10";

DbCommand cmd = new DbCommand(connection, sql);

Result res = cmd.Execute();

String name = res[0]["FIRST\_NAME"];

In contrast, the following makes use of an ORM API, allowing the writing of code which naturally makes use of the features of the language.

Person p = repository.GetPerson(10);

String name = p.getFirstName();

The case above makes use of an object representing the storage repository, and methods of that object. Other frameworks might provide code as static methods, as in the below, and yet other methods may not implement an object-oriented system at all. Often the choice of paradigm is made to fit the ORM best into the surrounding language's design principles.

Person p = Person.Get(10);

Usually, the framework will expose some filtering and querying functionality, allowing subsets of the storage base to be accessed and modified. The code below queries for people in the database whose ID value is '10'.

Person p = Person.Get(Person.Properties.Id == 10);

**Comparison with traditional data access techniques**

Compared to traditional techniques of exchange between an object-oriented language and a relational database, ORM often reduces the amount of code that needs to be written.[[2]](https://en.wikipedia.org/wiki/Object-relational_mapping#cite_note-2)

Disadvantages of ORM tools generally stem from the high level of abstraction obscuring what is actually happening in the implementation code. Also, heavy reliance on ORM software has been cited as a major factor in producing poorly designed databases.[[3]](https://en.wikipedia.org/wiki/Object-relational_mapping#cite_note-3)

**Object-oriented databases**

Another approach is to use an [object-oriented database management system](https://en.wikipedia.org/wiki/Object_database) (OODBMS) or [document-oriented databases](https://en.wikipedia.org/wiki/Document-oriented_database) such as native [XML databases](https://en.wikipedia.org/wiki/XML_database) that provide more flexibility in data modeling. OODBMSs are databases designed specifically for working with object-oriented values. Using an OODBMS eliminates the need for converting data to and from its SQL form, as the data is stored in its original object representation and relationships are directly represented, rather than requiring [join tables](https://en.wikipedia.org/wiki/Junction_table)/operations. The equivalent of ORMs for [Document-oriented databases](https://en.wikipedia.org/wiki/Document-oriented_database) are called Object-Document Mappers (ODMs).

Document oriented databases also prevent the user from having to "shred" objects into table rows. Many of these systems also support the [XQuery](https://en.wikipedia.org/wiki/XQuery) query language to retrieve datasets.

Object-oriented databases tend to be used in complex, niche applications. One of the arguments against using an OODBMS is that switching from an SQL DBMS to a purely object-oriented DBMS means that you may lose the capability to create application independent queries for retrieving ad hoc combinations of data without restriction to access path.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] For this reason, many programmers find themselves more at home with an object-SQL mapping system, even though most object-oriented databases are able to process SQL queries to a limited extent. Other OODBMS (such as RavenDB[[4]](https://en.wikipedia.org/wiki/Object-relational_mapping" \l "cite_note-ravendb-link-4)) provide replication to SQL databases, as a means of addressing the need for ad hoc queries, while preserving well-known query patterns.

**Challenges**

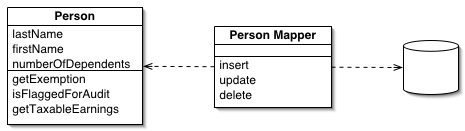
A variety of difficulties arise when considering how to match an object system to a relational database. These difficulties are referred to as the [object-relational impedance mismatch](https://en.wikipedia.org/wiki/Object-relational_impedance_mismatch).

An alternative to implementing ORM is use of the native procedural languages provided with every major database. These can be called from the client using SQL statements. The [Data Access Object](https://en.wikipedia.org/wiki/Data_access_object) (DAO) design pattern is used to abstract these statements and offer a lightweight object-oriented interface to the rest of the application.[[5]](https://en.wikipedia.org/wiki/Object-relational_mapping#cite_note-5)

**Data Mapper**

*A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.*

For a full description see [P of EAA](https://martinfowler.com/books/eaa.html) page **165**



Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

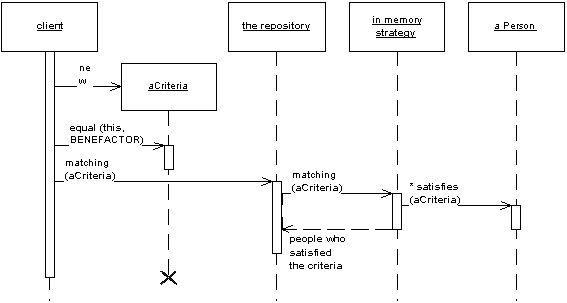
The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.) Since it's a form of Mapper (473), Data Mapper itself is even unknown to the domain layer.

**Repository**

**by Edward Hieatt and Rob Mee**

*Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.*

For a full description see [P of EAA](https://martinfowler.com/books/eaa.html) page **322**



A system with a complex domain model often benefits from a layer, such as the one provided by Data Mapper (165), that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.

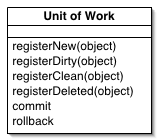
A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

You can also find a good write-up of this pattern in [Domain Driven Design](http://domaindrivendesign.org/books/#DDD).

**Unit of Work**

*Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.*

For a full description see [P of EAA](https://martinfowler.com/books/eaa.html) page **184**



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

# SOLID (object-oriented design)

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#mw-head), [search](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#p-search)

*"SOLID" redirects here. For other uses, see* [*Solid (disambiguation)*](https://en.wikipedia.org/wiki/Solid_(disambiguation))*.*

|  |
| --- |
| **SOLID** |
| **Principles** |
| * [Single Responsibility](https://en.wikipedia.org/wiki/Single_responsibility_principle) * [Open/closed](https://en.wikipedia.org/wiki/Open/closed_principle) * [Liskov substitution](https://en.wikipedia.org/wiki/Liskov_substitution_principle) * [Interface segregation](https://en.wikipedia.org/wiki/Interface_segregation_principle) * [Dependency inversion](https://en.wikipedia.org/wiki/Dependency_inversion_principle) |
|  |

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), the term **SOLID** is a [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) [acronym](https://en.wikipedia.org/wiki/Acronym) for five [design](https://en.wikipedia.org/wiki/Object-oriented_design) principles intended to make software designs more understandable, flexible and [maintainable](https://en.wikipedia.org/wiki/Software_maintenance). The principles are a subset of many principles promoted by [Robert C. Martin](https://en.wikipedia.org/wiki/Robert_C._Martin), [[1]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-ub-old-web-solid-1)[[2]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-ub-solid-2)[[3]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-metz-presentation-2009-3). Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as [agile development](https://en.wikipedia.org/wiki/Agile_software_development) or [Adaptive Software Development](https://en.wikipedia.org/wiki/Adaptive_Software_Development).[[3]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-metz-presentation-2009-3) The SOLID acronym was introduced by Michael Feathers.

## Overview

|  |  |
| --- | --- |
| **Initial** | **Concept** |
| **S** | [Single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)[[4]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-4)  a [class](https://en.wikipedia.org/wiki/Class_(computer_science)) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class) |
| **O** | [Open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle)[[5]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-5)  “software entities … should be open for extension, but closed for modification.” |
| **L** | [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle)[[6]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-6)  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also [design by contract](https://en.wikipedia.org/wiki/Design_by_contract). |
| **I** | [Interface segregation principle](https://en.wikipedia.org/wiki/Interface_segregation_principle)[[7]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-7)  “many client-specific interfaces are better than one general-purpose interface.”[[8]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-martin-design-principles-8) |
| **D** | [Dependency inversion principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle)[[9]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-9)  one should “depend upon abstractions, [not] concretions.”[[8]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-martin-design-principles-8) |

### Design and development principles[[edit](https://en.wikipedia.org/w/index.php?title=SOLID_(object-oriented_design)&action=edit&section=4)]

* [Package principles](https://en.wikipedia.org/wiki/Package_principles)
* [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)
* [GRASP (object-oriented design)](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design))
* [KISS](https://en.wikipedia.org/wiki/KISS_principle)
* [YAGNI](https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it)

# Facade pattern

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/Facade_pattern#mw-head), [search](https://en.wikipedia.org/wiki/Facade_pattern#p-search)

The **facade pattern** (also spelled *façade*) is a [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) commonly used with [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming). The name is an analogy to an architectural [façade](https://en.wikipedia.org/wiki/Facade).

A facade is an object that provides a simplified interface to a larger body of code, such as a [class library](https://en.wikipedia.org/wiki/Class_library). A facade can

* make a [software library](https://en.wikipedia.org/wiki/Software_library) easier to use, understand, and test, since the facade has convenient methods for common tasks,
* make the library more readable, for the same reason,
* reduce [dependencies](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system,
* wrap a poorly designed collection of [APIs](https://en.wikipedia.org/wiki/Application_programming_interface) with a single well-designed API.

The Facade design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by client. These members access the system on behalf of the facade client and hide the implementation details.

## Overview

The Facade [[1]](https://en.wikipedia.org/wiki/Facade_pattern#cite_note-GoF-1) design pattern is one of the twenty-three well-known [*GoF design patterns*](https://en.wikipedia.org/wiki/Design_Patterns) that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Facade design pattern solve? [[2]](https://en.wikipedia.org/wiki/Facade_pattern#cite_note-2)

* To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
* The dependencies on a subsystem should be minimized.

Clients that access a complex subsystem directly refer to (depend on) many different objects having different interfaces (tight coupling), which makes the clients hard to implement, change, test, and reuse.

What solution does the Facade design pattern describe?

Define a Facade object that

* implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and
* may perform additional functionality before/after forwarding a request.

This enables to work through a Facade object to minimize the dependencies on a subsystem.  
See also the UML class and sequence diagram below.

## Usage

A Facade is used when an easier or simpler interface to an underlying object is desired.[[3]](https://en.wikipedia.org/wiki/Facade_pattern#cite_note-3) Alternatively, an [adapter](https://en.wikipedia.org/wiki/Adapter_pattern) can be used when the wrapper must respect a particular interface and must support [polymorphic](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) behavior. A [decorator](https://en.wikipedia.org/wiki/Decorator_pattern) makes it possible to add or alter behavior of an interface at run-time.

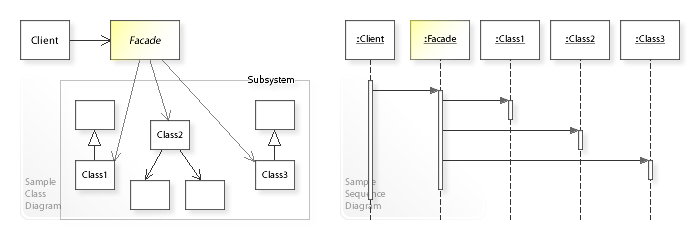
|  |  |
| --- | --- |
| **Pattern** | **Intent** |
| [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern) | Converts one interface to another so that it matches what the client is expecting |
| [Decorator](https://en.wikipedia.org/wiki/Decorator_pattern) | Dynamically adds responsibility to the interface by wrapping the original code |
| Facade | Provides a simplified interface |

The facade pattern is typically used when

* a simple interface is required to access a complex system,
* a system is very complex or difficult to understand,
* an entry point is needed to each level of layered software, or
* the abstractions and implementations of a subsystem are tightly coupled.

## Structure

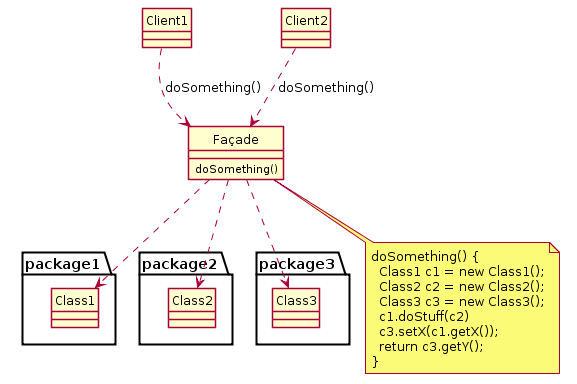
### UML class and sequence diagram

[](https://en.wikipedia.org/wiki/File:W3sDesign_Facade_Design_Pattern_UML.jpg)

A sample UML class and sequence diagram for the Facade design pattern. [[4]](https://en.wikipedia.org/wiki/Facade_pattern#cite_note-4)

In the above [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) [class diagram](https://en.wikipedia.org/wiki/Class_diagram), the Client class doesn't access the subsystem classes directly. Instead, the Client works through a Facade class that implements a simple interface in terms of (by delegating to) the subsystem classes (Class1, Class2, and Class3). The Client depends only on the simple Facade interface and is independent of the complex subsystem.  
The sequence diagram shows the run-time interactions: The Client object works through a Facade object that delegates the request to the Class1, Class2, and Class3 instances that perform the request.

### UML class diagram[[edit](https://en.wikipedia.org/w/index.php?title=Facade_pattern&action=edit&section=5)]

[](https://en.wikipedia.org/wiki/File:Example_of_Facade_design_pattern_in_UML.png)

Facade

The facade class abstracts Packages 1, 2, and 3 from the rest of the application.

Clients

The objects are using the Facade Pattern to access resources from the Packages.

## Example

This is an abstract example of how a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive).

### C#[[edit](https://en.wikipedia.org/w/index.php?title=Facade_pattern&action=edit&section=7)]

#### Implementation[[edit](https://en.wikipedia.org/w/index.php?title=Facade_pattern&action=edit&section=8)]

namespace DesignPattern.Facade

{

class SubsystemA

{

public string OperationA1()

{

return "Subsystem A, Method A1\n";

}

public string OperationA2()

{

return "Subsystem A, Method A2\n";

}

}

class SubsystemB

{

public string OperationB1()

{

return "Subsystem B, Method B1\n";

}

public string OperationB2()

{

return "Subsystem B, Method B2\n";

}

}

class SubsystemC

{

public string OperationC1()

{

return "Subsystem C, Method C1\n";

}

public string OperationC2()

{

return "Subsystem C, Method C2\n";

}

}

public class Facade

{

private readonly SubsystemA a = new SubsystemA();

private readonly SubsystemB b = new SubsystemB();

private readonly SubsystemC c = new SubsystemC();

public void Operation1()

{

Console.WriteLine("Operation 1\n" +

a.OperationA1() +

b.OperationB1() +

c.OperationC1());

}

public void Operation2()

{

Console.WriteLine("Operation 2\n" +

a.OperationA2() +

b.OperationB2() +

c.OperationC2());

}

}

}

#### Sample code[[edit](https://en.wikipedia.org/w/index.php?title=Facade_pattern&action=edit&section=9)]

namespace DesignPattern.Facade.Sample

{

// The 'Subsystem ClassA' class

class CarModel

{

public void SetModel()

{

Console.WriteLine(" CarModel - SetModel");

}

}

/// <summary>

/// The 'Subsystem ClassB' class

/// </summary>

class CarEngine

{

public void SetEngine()

{

Console.WriteLine(" CarEngine - SetEngine");

}

}

// The 'Subsystem ClassC' class

class CarBody

{

public void SetBody()

{

Console.WriteLine(" CarBody - SetBody");

}

}

// The 'Subsystem ClassD' class

class CarAccessories

{

public void SetAccessories()

{

Console.WriteLine(" CarAccessories - SetAccessories");

}

}

// The 'Facade' class

public class CarFacade

{

private readonly CarAccessories accessories;

private readonly CarBody body;

private readonly CarEngine engine;

private readonly CarModel model;

public CarFacade()

{

accessories = new CarAccessories();

body = new CarBody();

engine = new CarEngine();

model = new CarModel();

}

public void CreateCompleteCar()

{

Console.WriteLine("\*\*\*\*\*\*\*\* Creating a Car \*\*\*\*\*\*\*\*\*\*");

model.SetModel();

engine.SetEngine();

body.SetBody();

accessories.SetAccessories();

Console.WriteLine("\*\*\*\*\*\*\*\* Car creation is completed. \*\*\*\*\*\*\*\*\*\*");

}

}

// Facade pattern demo

class Program

{

static void Main(string[] args)

{

var facade = new CarFacade();

facade.CreateCompleteCar();

Console.ReadKey();

}

}

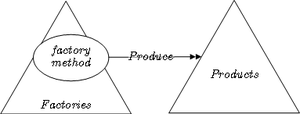
}

# Factory (object-oriented programming)

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#mw-head), [search](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#p-search)

*"Factory pattern" redirects here. For the GoF design patterns using factories, see* [*factory method pattern*](https://en.wikipedia.org/wiki/Factory_method_pattern) *and* [*abstract factory pattern*](https://en.wikipedia.org/wiki/Abstract_factory_pattern)*.*

[](https://en.wikipedia.org/wiki/File:Factory_Method_pattern_in_LePUS3.png)

Factory Method in [LePUS3](https://en.wikipedia.org/wiki/LePUS3)

In [object-oriented programming (OOP)](https://en.wikipedia.org/wiki/Object-oriented_programming), a **factory** is an [object](https://en.wikipedia.org/wiki/Object_(computer_science)) for [creating other objects](https://en.wikipedia.org/wiki/Object_creation) – formally a factory is a function or method that returns objects of a varying prototype or class[[1]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-1) from some method call, which is assumed to be "new".[[a]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-2) More broadly, a [subroutine](https://en.wikipedia.org/wiki/Subroutine) that returns a "new" object may be referred to as a "factory", as in *factory method* or *factory function*. This is a basic concept in OOP, and forms the basis for a number of related [software design patterns](https://en.wikipedia.org/wiki/Software_design_pattern).

## Motivation

In [class-based programming](https://en.wikipedia.org/wiki/Class-based_programming), a factory is an [abstraction](https://en.wikipedia.org/wiki/Abstraction_(computer_science)) of a [constructor](https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)) of a class, while in [prototype-based programming](https://en.wikipedia.org/wiki/Prototype-based_programming) a factory is an abstraction of a prototype object. A constructor is concrete in that it creates objects as instances of a single class, and by a specified process (class instantiation), while a factory can create objects by instantiating various classes, or by using other allocation schemes such as an [object pool](https://en.wikipedia.org/wiki/Object_pool). A prototype object is concrete in that it is used to create objects by being [cloned](https://en.wikipedia.org/wiki/Cloning_(programming)), while a factory can create objects by cloning various prototypes, or by other allocation schemes.

Factories may be invoked in various ways, most often a method call (a [*factory method*](https://en.wikipedia.org/wiki/Factory_method)), sometimes by being called as a function if the factory is a [function object](https://en.wikipedia.org/wiki/Function_object) (a *factory function*). In some languages factories are generalizations of constructors, meaning constructors are themselves factories and these are invoked in the same way. In other languages factories and constructors are invoked differently, for example using the keyword new to invoke constructors but an ordinary method call to invoke factories; in these languages factories are an abstraction of constructors but not strictly a generalization, as constructors are not themselves factories.

## Terminology

Terminology differs as to whether the concept of a factory is itself a design pattern – in the seminal book [*Design Patterns*](https://en.wikipedia.org/wiki/Design_Patterns) there is no "factory pattern", but instead two patterns ([factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern) and [abstract factory pattern](https://en.wikipedia.org/wiki/Abstract_factory_pattern)) that use factories. Some sources refer to the concept as the **factory pattern**,[[2]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-oodesign-3)[[3]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-4) while others consider the concept itself a [programming idiom](https://en.wikipedia.org/wiki/Programming_idiom),[[4]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-headfirst-5) reserving the term "factory pattern" or "factory patterns" to more complicated patterns that use factories, most often the factory method pattern; in this context, the concept of a factory itself may be referred to as a **simple factory.**[[4]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-headfirst-5) In other contexts, particularly the Python language, "factory" itself is used, as in this article.[[5]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-6) More broadly, "factory" may be applied not just to an object that returns objects from some method call, but to a [*subroutine*](https://en.wikipedia.org/wiki/Subroutine) that returns objects, as in a *factory function* (even if functions are not objects) or *factory method.*[[6]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-7) Because in many languages factories are invoked by calling a method, the general concept of a factory is often confused with the specific [factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern) design pattern.

## Use

OOP provides [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)) on object *use* by [method dispatch](https://en.wikipedia.org/wiki/Method_dispatch), formally [subtype polymorphism](https://en.wikipedia.org/wiki/Subtype_polymorphism) via [single dispatch](https://en.wikipedia.org/wiki/Single_dispatch) determined by the type of the object on which the method is called. However, this does not work for constructors, as constructors *create* an object of some type, rather than *use* an existing object. More concretely, when a constructor is called, there is no object yet on which to dispatch.[[b]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-8)

Using factories instead of constructors or prototypes allows one to use polymorphism for object creation, not only object use. Specifically, using factories provides [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)), and means the code is not tied to specific classes or objects, and thus the class hierarchy or prototypes can be changed or [refactored](https://en.wikipedia.org/wiki/Refactored) without needing to change code that uses them – they abstract from the class hierarchy or prototypes.

More technically, in languages where factories generalize constructors, factories can usually be used anywhere constructors can be,[[c]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-9) meaning that interfaces that accept a constructor can also in general accept a factory – usually one only need something that creates an object, rather than needing to specify a class and instantiation.

For example, in Python, the collections.defaultdict class[[7]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-10) has a constructor which creates an object of type defaultdict[[d]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)" \l "cite_note-11) whose default values are produced by invoking a factory. The factory is passed as an argument to the constructor, and can itself be a constructor, or any thing that behaves like a constructor – a callable object that returns an object, i.e., a factory. For example, using the list constructor for lists:

# collections.defaultdict([default\_factory[, ...]])

d = defaultdict(list)

### Object creation

Factory objects are used in situations where getting hold of an object of a particular kind is a more complex process than simply creating a new object, notably if complex allocation or initialization is desired. Some of the processes required in the creation of an object include determining which object to create, managing the lifetime of the object, and managing specialized build-up and tear-down concerns of the object. The factory object might decide to create the object's [class](https://en.wikipedia.org/wiki/Class_(computer_science)) (if applicable) dynamically, return it from an [object pool](https://en.wikipedia.org/wiki/Object_pool), do complex configuration on the object, or other things. Similarly, using this definition, a [singleton](https://en.wikipedia.org/wiki/Singleton_(mathematics)) implemented by the [singleton pattern](https://en.wikipedia.org/wiki/Singleton_pattern) is a formal factory – it returns an object, but does not create new objects beyond the single instance.

## Examples

The simplest example of a factory is a simple factory function, which just invokes a constructor and returns the result. In Python, a factory function f that instantiates a class A can be implemented as:

def f():

return A()

A simple factory function implementing the singleton pattern is:

def f():

if f.obj is None:

f.obj = A()

return f.obj

f.obj = None

This will create an object when first called, and always return the same object thereafter.

## Syntax

Factories may be invoked in various ways, most often a method call (a *factory method*), sometimes by being called as a function if the factory is a callable object (a *factory function*). In some languages constructors and factories have identical syntax, while in others constructors have special syntax. In languages where constructors and factories have identical syntax, like Python, Perl, Ruby, Object Pascal, and F#,[[e]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-12) constructors can be transparently replaced by factories. In languages where they differ, one must distinguish them in interfaces, and switching between constructors and factories requires changing the calls.

## Semantics

In languages where objects are [dynamically allocated](https://en.wikipedia.org/wiki/Dynamic_memory_allocation), as in Java or Python, factories are semantically equivalent to constructors. However, in languages such as C++ that allow some objects to be statically allocated, factories are different from constructors for statically allocated classes, as the latter can have memory allocation determined at compile time, while allocation of the return values of factories must be determined at run time. If a constructor can be passed as an argument to a function, then invocation of the constructor and allocation of the return value must be done dynamically at run time, and thus have similar or identical semantics to invoking a factory.

## Design patterns

*Main article:* [*Creational pattern*](https://en.wikipedia.org/wiki/Creational_pattern)

Factories are used in various [design patterns](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)), specifically in [creational patterns](https://en.wikipedia.org/wiki/Creational_pattern) such as the [Design pattern object library](https://en.wikipedia.org/w/index.php?title=Design_pattern_object_library&action=edit&redlink=1). Specific recipes have been developed to implement them in many languages. For example, several "[GoF patterns](https://en.wikipedia.org/wiki/Design_Patterns_(book)" \o "Design Patterns (book))", like the "[Factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern)", the "[Builder](https://en.wikipedia.org/wiki/Builder_pattern)" or even the "[Singleton](https://en.wikipedia.org/wiki/Singleton_pattern)" are implementations of this concept. The "[Abstract factory pattern](https://en.wikipedia.org/wiki/Abstract_factory_pattern)" instead is a method to build collections of factories.

In some design patterns, a factory object has a [method](https://en.wikipedia.org/wiki/Method_(computer_science)) for every kind of object it is capable of creating. These methods optionally accept [parameters](https://en.wikipedia.org/wiki/Parameter) defining how the object is created, and then return the created object.

## Applications

Factory objects are common in [toolkits](https://en.wikipedia.org/wiki/Toolkit) and [frameworks](https://en.wikipedia.org/wiki/Software_framework) where library code needs to create objects of types which may be subclassed by applications using the framework. They are also used in [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) to allow classes to be put under test.[[8]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-13)

Factories determine the actual *concrete* type of [object](https://en.wikipedia.org/wiki/Object_(computer_science)) to be created, and it is here that the object is actually created. As the factory only returns an abstract interface to the object, the client code does not know – and is not burdened by – the actual concrete type of the object which was just created. However, the type of a concrete object is known by the abstract factory. In particular, this means:

* The client code has no knowledge whatsoever of the concrete [type](https://en.wikipedia.org/wiki/Data_type), not needing to include any [header files](https://en.wikipedia.org/wiki/Header_file) or [class](https://en.wikipedia.org/wiki/Class_(computer_science)) [declarations](https://en.wikipedia.org/wiki/Declaration_(computer_science)) relating to the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their [abstract interface](https://en.wikipedia.org/wiki/Abstract_interface).
* Adding new concrete types is done by modifying the client code to use a different factory, a modification which is typically one line in one file. This is significantly easier than modifying the client code to instantiate a new type, which would require changing *every* location in the code where a new object is created.

## Applicability

Factories can be used when:

1. The creation of an object makes reuse impossible without significant duplication of code.
2. The creation of an object requires access to information or resources that should not be contained within the composing class.
3. The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.

Factories, specifically factory methods, are common in [toolkits](https://en.wikipedia.org/wiki/Toolkit) and [frameworks](https://en.wikipedia.org/wiki/Software_framework), where library code needs to create objects of types that may be subclassed by applications using the framework.

Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from another.

Factory methods are used in [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) to allow classes to be put under test.[[9]](https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)#cite_note-14) If such a class Foo creates another object Dangerous that can't be put under automated [unit tests](https://en.wikipedia.org/wiki/Unit_test) (perhaps it communicates with a production database that isn't always available), then the creation of Dangerous objects is placed in the [virtual](https://en.wikipedia.org/wiki/Virtual_function) factory method createDangerous in class Foo. For testing, TestFoo (a subclass of Foo) is then created, with the virtual factory method createDangerous overridden to create and return FakeDangerous, a [fake object](https://en.wikipedia.org/wiki/Fake_object). Unit tests then use TestFoo to test the functionality of Foo without incurring the side effect of using a real Dangerous object.

## Benefits and variants

Besides use in design patterns, factories, especially factory methods, have various benefits and variations.

### Descriptive names[[edit](https://en.wikipedia.org/w/index.php?title=Factory_(object-oriented_programming)&action=edit&section=12)]

A factory method has a distinct name. In many object-oriented languages, constructors must have the same name as the class they are in, which can lead to ambiguity if there is more than one way to create an object (see [overloading](https://en.wikipedia.org/wiki/Polymorphism_(computer_science)#Overloading)). Factory methods have no such constraint and can have descriptive names; these are sometimes known as **alternative constructors**. As an example, when [complex numbers](https://en.wikipedia.org/wiki/Complex_number) are created from two real numbers the real numbers can be interpreted as Cartesian or polar coordinates, but using factory methods, the meaning is clear, as illustrated by the following example in C#.

public class Complex

{

public double real;

public double imaginary;

public static Complex FromCartesian(double real, double imaginary)

{

return new Complex(real, imaginary);

}

public static Complex FromPolar(double modulus, double angle)

{

return new Complex(modulus \* Math.Cos(angle), modulus \* Math.Sin(angle));

}

private Complex(double real, double imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

}

Complex product = Complex.FromPolar(1, Math.PI);

When factory methods are used for disambiguation like this, the raw constructors are often made private to force clients to use the factory methods.

### Encapsulation[[edit](https://en.wikipedia.org/w/index.php?title=Factory_(object-oriented_programming)&action=edit&section=13)]

Factory methods encapsulate the creation of objects. This can be useful if the creation process is very complex; for example, if it depends on settings in configuration files or on user input.

Consider as an example a program that reads [image files](https://en.wikipedia.org/wiki/Image_file). The program supports different image formats, represented by a reader class for each format.

Each time the program reads an image, it needs to create a reader of the appropriate type based on some information in the file. This logic can be encapsulated in a factory method. This approach has also been referred to as the [Simple Factory](http://corey.quickshiftconsulting.com/1/post/2009/5/first-post.html)

# Bridge pattern

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/Bridge_pattern#mw-head), [search](https://en.wikipedia.org/wiki/Bridge_pattern#p-search)

The **bridge pattern** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) used in [software engineering](https://en.wikipedia.org/wiki/Software_engineering) that is meant to *"decouple an* [*abstraction*](https://en.wikipedia.org/wiki/Abstraction_(computer_science)) *from its* [*implementation*](https://en.wikipedia.org/wiki/Implementation) *so that the two can vary independently"*, introduced by the [Gang of Four](https://en.wikipedia.org/wiki/Design_Patterns).[[1]](https://en.wikipedia.org/wiki/Bridge_pattern#cite_note-1) The *bridge* uses [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_science)), [aggregation](https://en.wikipedia.org/wiki/Object_composition#Aggregation), and can use [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)) to separate responsibilities into different [classes](https://en.wikipedia.org/wiki/Class_(computer_science)).

When a class varies often, the features of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) become very useful because changes to a [program](https://en.wikipedia.org/wiki/Computer_program)'s [code](https://en.wikipedia.org/wiki/Source_code) can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the *abstraction* and what the class can do as the *implementation*. The bridge pattern can also be thought of as two layers of abstraction.

When there is only one fixed implementation, this pattern is known as the [Pimpl](https://en.wikipedia.org/wiki/Pimpl) idiom in the [C++](https://en.wikipedia.org/wiki/C%2B%2B) world.

The **bridge pattern** is often confused with the [adapter pattern](https://en.wikipedia.org/wiki/Adapter_pattern). In fact, the **bridge pattern** is often implemented using the [object **adapter pattern**](https://en.wikipedia.org/wiki/Adapter_pattern#Object_Adapter_pattern), e.g. in the Java code below.

**Variant**: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.

## Overview

The Bridge [[2]](https://en.wikipedia.org/wiki/Bridge_pattern#cite_note-GoF-2) design pattern is one of the twenty-three well-known [*GoF design patterns*](https://en.wikipedia.org/wiki/Design_Patterns) that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Bridge design pattern solve? [[3]](https://en.wikipedia.org/wiki/Bridge_pattern#cite_note-3)

* An abstraction and its implementation should be defined and extended independently from each other.
* A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

When using subclassing, different subclasses implement an abstract class in different ways. But an implementation is bound to the abstraction at compile-time and can't be changed at run-time.

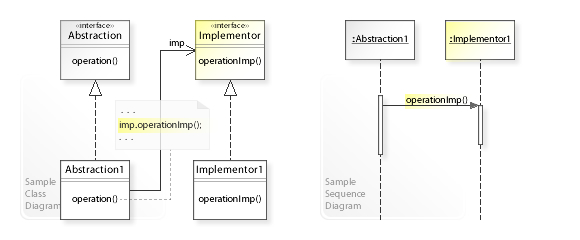
What solution does the Bridge design pattern describe?

* Separate an abstraction (Abstraction) from its implementation (Implementor) by putting them in separate class hierarchies.
* Implement the Abstraction in terms of (by delegating to) an Implementor object.

This enables to configure an Abstraction with an Implementor object at run-time.  
See also the UML class and sequence diagram below.

## Structure

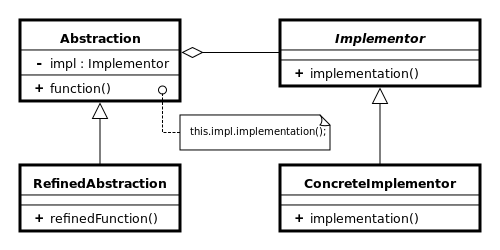
### UML class and sequence diagram[[edit](https://en.wikipedia.org/w/index.php?title=Bridge_pattern&action=edit&section=3)]

[](https://en.wikipedia.org/wiki/File:W3sDesign_Bridge_Design_Pattern_UML.jpg)

A sample UML class and sequence diagram for the Bridge design pattern. [[4]](https://en.wikipedia.org/wiki/Bridge_pattern#cite_note-4)

In the above [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) [class diagram](https://en.wikipedia.org/wiki/Class_diagram), an abstraction (Abstraction) isn't implemeted as usual in a single inheritance hierarchy. Instead, there is one hierarchy for an abstraction (Abstraction) and a separate hierarchy for its implementation (Implementor), which makes the two independent from each other. The Abstraction interface (operation()) is implemented in terms of (by delegating to) the Implementor interface (imp.operationImp()).  
The [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) [sequence diagram](https://en.wikipedia.org/wiki/Sequence_diagram) shows the run-time interactions: The Abstraction1 object delegates implementation to the Implementor1 object (by calling operationImp() on Implementor1), which performs the operation and returns to Abstraction1.

### Class diagram[[edit](https://en.wikipedia.org/w/index.php?title=Bridge_pattern&action=edit&section=4)]

[](https://en.wikipedia.org/wiki/File:Bridge_UML_class_diagram.svg)

Abstraction (abstract class)

defines the abstract interface

maintains the Implementor reference.

RefinedAbstraction (normal class)

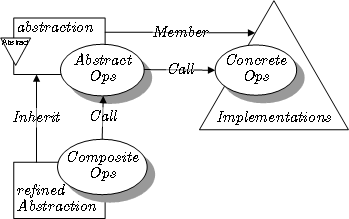
extends the interface defined by Abstraction

Implementor (interface)

defines the interface for implementation classes

ConcreteImplementor (normal class)

implements the Implementor interface

[](https://en.wikipedia.org/wiki/File:Bridge_pattern_in_LePUS3.1.gif)

Bridge in [LePUS3](https://en.wikipedia.org/wiki/Lepus3) ([legend](http://lepus.org.uk/ref/legend/legend.xml))

## Example

### C#[[edit](https://en.wikipedia.org/w/index.php?title=Bridge_pattern&action=edit&section=6)]

Bridge pattern compose objects in tree structure. It decouples abstraction from implementation. Here abstraction represents the client from which the objects will be called. An example implemented in C# is given below

// Helps in providing truly decoupled architecture

public interface IBridge

{

void Function1();

void Function2();

}

public class Bridge1 : IBridge

{

public void Function1()

{

Console.WriteLine("Bridge1.Function1");

}

public void Function2()

{

Console.WriteLine("Bridge1.Function2");

}

}

public class Bridge2 : IBridge

{

public void Function1()

{

Console.WriteLine("Bridge2.Function1");

}

public void Function2()

{

Console.WriteLine("Bridge2.Function2");

}

}

public interface IAbstractBridge

{

void CallMethod1();

void CallMethod2();

}

public class AbstractBridge : IAbstractBridge

{

public IBridge bridge;

public AbstractBridge(IBridge bridge)

{

this.bridge = bridge;

}

public void CallMethod1()

{

this.bridge.Function1();

}

public void CallMethod2()

{

this.bridge.Function2();

}

}

As you can see, the Bridge classes are the Implementation that uses the same interface-oriented architecture to create objects. On the other hand, the abstraction takes an object of the implementation phase and runs its method. Thus, it makes completely decoupled from one another.

# Create, read, update and delete

From Wikipedia, the free encyclopedia

Jump to: [navigation](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#mw-head), [search](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#p-search)

*"CRUD" redirects here. For other uses, see* [*Crud*](https://en.wikipedia.org/wiki/Crud_(disambiguation))*.*

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), **create, read, update, and delete**[[1]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-james-martin-1) (as an [acronym](https://en.wikipedia.org/wiki/Acronym_and_initialism) **CRUD**) are the four basic functions of [persistent storage](https://en.wikipedia.org/wiki/Persistent_storage).[[2]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-2) Alternate words are sometimes used when defining the four basic functions of *CRUD*, such as *retrieve* instead of *read*, *modify* instead of *update*, or *destroy* instead of *delete*. *CRUD* is also sometimes used to describe [user interface](https://en.wikipedia.org/wiki/User_interface) conventions that facilitate viewing, searching, and changing [information](https://en.wikipedia.org/wiki/Information); often using computer-based [forms](https://en.wikipedia.org/wiki/Form_(document)) and [reports](https://en.wikipedia.org/wiki/Report). The term was likely first popularized by [James Martin](https://en.wikipedia.org/wiki/James_Martin_(author)) in his 1983 book *Managing the Data-base Environment*.[[1]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-james-martin-1)[[3]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-3) The acronym may be extended to CRUDL to cover *listing* of large data sets which bring additional complexity such as [pagination](https://en.wikipedia.org/wiki/Pagination) when the data sets are too large to hold easily in memory.

## Other variations

Other variations of CRUD include:

* BREAD (Browse, Read, Edit, Add, Delete) [[4]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-4)
* MADS (Modify, Add, Delete, Show)[*[citation needed](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed" \o "Wikipedia:Citation needed)*]
* DAVE (Delete, Add, View, Edit)[[5]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-5)
* CRAP (Create, Retrieve, Alter, Purge)
* VEND (View, Edit, New, Delete)

## Database applications

The acronym CRUD refers to all of the major functions that are implemented in [relational database](https://en.wikipedia.org/wiki/Relational_database) [applications](https://en.wikipedia.org/wiki/Application_software). Each letter in the acronym can map to a standard [SQL](https://en.wikipedia.org/wiki/SQL) statement, [HTTP method](https://en.wikipedia.org/wiki/HTTP_method) (this is typically used to build [RESTful APIs](https://en.wikipedia.org/wiki/RESTful_API)[[6]](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete#cite_note-6)) or [DDS](https://en.wikipedia.org/wiki/Data_Distribution_Service) operation:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL** | **HTTP** | **DDS** |
| Create | [INSERT](https://en.wikipedia.org/wiki/Insert_(SQL)) | [PUT](https://en.wikipedia.org/wiki/PUT_(HTTP)) / [POST](https://en.wikipedia.org/wiki/POST_(HTTP)) | write |
| Read (Retrieve) | [SELECT](https://en.wikipedia.org/wiki/Select_(SQL)) | [GET](https://en.wikipedia.org/wiki/GET_(HTTP)) | read / take |
| Update (Modify) | [UPDATE](https://en.wikipedia.org/wiki/Update_(SQL)) | [PUT](https://en.wikipedia.org/wiki/PUT_(HTTP)) / [POST](https://en.wikipedia.org/wiki/POST_(HTTP)) / [PATCH](https://en.wikipedia.org/wiki/PATCH_(HTTP)) | write |
| Delete (Destroy) | [DELETE](https://en.wikipedia.org/wiki/Delete_(SQL)) | [DELETE](https://en.wikipedia.org/wiki/DELETE_(HTTP)) | dispose |

The comparison of the database oriented CRUD operations to HTTP methods has some flaws. Strictly speaking, both PUT and POST can create resources; the key difference is that POST leaves it for the server to decide at what URI to make the new resource available, whilst PUT dictates what URI to use; URIs are of course a concept that doesn't really line up with CRUD. The significant point about PUT is that it will replace whatever resource the URI was previously referring to with a brand new version, hence the PUT method being listed for Update as well. PUT is a 'replace' operation, which one could argue is not 'update'.

Although a relational database provides a common [persistence layer](https://en.wikipedia.org/wiki/Persistence_layer) in software applications, numerous other persistence layers exist. CRUD functionality can be implemented with an [object database](https://en.wikipedia.org/wiki/Object_database), an [XML database](https://en.wikipedia.org/wiki/XML_database), [flat text files](https://en.wikipedia.org/wiki/Flat_file_database), custom file formats, tape, or card, for example.

## User interface

CRUD is also relevant at the user interface level of most applications. For example, in [address book](https://en.wikipedia.org/wiki/Address_book) software, the basic storage unit is an individual [*contact*](https://en.wikipedia.org/wiki/Contact_(social)) *entry*. As a bare minimum, the software must allow the user to

* Create or add new entries
* Read, retrieve, search, or view existing entries
* Update or edit existing entries
* Delete/deactivate/remove existing entries

Without at least these four operations, the software cannot be considered complete. Because these operations are so fundamental, they are often documented and described under one comprehensive heading, such as "contact management", "content management" or "contact maintenance" (or "document management" in general, depending on the basic storage unit for the particular application).

**Design Patterns: Elements of Reusable Object-Oriented Software**

is a [software engineering](https://en.wikipedia.org/wiki/Software_engineering) book describing [software design patterns](https://en.wikipedia.org/wiki/Software_design_pattern). The book's authors are [Erich Gamma](https://en.wikipedia.org/wiki/Erich_Gamma), Richard Helm, [Ralph Johnson](https://en.wikipedia.org/wiki/Ralph_Johnson_(computer_scientist)) and [John Vlissides](https://en.wikipedia.org/wiki/John_Vlissides) with a foreword by [Grady Booch](https://en.wikipedia.org/wiki/Grady_Booch). The book is divided into two parts, with the first two chapters exploring the capabilities and pitfalls of object-oriented programming, and the remaining chapters describing 23 classic [software design patterns](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)). The book includes examples in [C++](https://en.wikipedia.org/wiki/C%2B%2B) and [Smalltalk](https://en.wikipedia.org/wiki/Smalltalk).

It has been influential to the field of software engineering and is regarded as an important source for object-oriented design theory and practice. More than 500,000 copies have been sold in English and in 13 other languages. The authors are often referred to as the **Gang of Four** (**GoF**).[[1]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-1)

## History

The book started at a [birds of a feather](https://en.wikipedia.org/wiki/Birds_of_a_feather_(computing)) (BoF) session at [OOPSLA](https://en.wikipedia.org/wiki/OOPSLA) '90, "Towards an Architecture Handbook", run by Bruce Anderson, where Erich Gamma and Richard Helm met and discovered their common interest. They were later joined by Ralph Johnson and John Vlissides.[[2]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-2) The original publication date of the book was October 21, 1994 with a 1995 copyright, hence it is often cited with a 1995-year, despite being published in 1994. The book was first made available to the public at the OOPSLA meeting held in Portland, Oregon, in October 1994. In 2005 the ACM [SIGPLAN](https://en.wikipedia.org/wiki/SIGPLAN) awarded that year's Programming Languages Achievement Award to the authors, in recognition of the impact of their work "on programming practice and programming language design".[[3]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-3) As of March 2012, the book was in its 40th printing.

## Introduction, Chapter 1[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=2)]

Chapter 1 is a discussion of [object-oriented](https://en.wikipedia.org/wiki/Object-oriented) design techniques, based on the authors' experience, which they believe would lead to good object-oriented software design, including:

* "Program to an 'interface', not an '**implementation'**." (Gang of Four 1995:18)
* [Composition over inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance): "Favor '[object composition](https://en.wikipedia.org/wiki/Object_composition)' over '[class inheritance](https://en.wikipedia.org/wiki/Inheritance_(computer_science))'." (Gang of Four 1995:20)

The authors claim the following as advantages of [interfaces](https://en.wikipedia.org/wiki/Interface_(computer_science)) over implementation:

* clients remain unaware of the specific types of objects they use, as long as the object adheres to the interface
* clients remain unaware of the classes that implement these objects; clients only know about the abstract class(es) defining the interface

Use of an interface also leads to [dynamic binding](https://en.wikipedia.org/wiki/Dynamic_dispatch) and [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming), which are central features of object-oriented programming.

The authors refer to [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)) as [*white-box*](https://en.wikipedia.org/wiki/White_box_(software_engineering)) *reuse*, with white-box referring to visibility, because the internals of parent classes are often visible to [subclasses](https://en.wikipedia.org/wiki/Subclass_(computer_science)). In contrast, the authors refer to [object composition](https://en.wikipedia.org/wiki/Object_composition) (in which objects with well-defined interfaces are used dynamically at runtime by objects obtaining references to other objects) as [*black-box*](https://en.wikipedia.org/wiki/Black_box) *reuse* because no internal details of composed objects need be visible in the code using them.

The authors discuss the tension between inheritance and encapsulation at length and state that in their experience, designers overuse inheritance (Gang of Four 1995:20). The danger is stated as follows:

"Because inheritance exposes a [subclass](https://en.wikipedia.org/wiki/Subclass_(computer_science)) to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation'". (Gang of Four 1995:19)

They warn that the implementation of a subclass can become so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change. Furthermore, they claim that a way to avoid this is to inherit only from abstract classes—but then, they point out that there is minimal code reuse.

Using inheritance is recommended mainly when adding to the functionality of existing components, reusing most of the old code and adding relatively small amounts of new code.

To the authors, 'delegation' is an extreme form of object composition that can always be used to replace inheritance. Delegation involves two objects: a 'sender' passes itself to a 'delegate' to let the delegate refer to the sender. Thus the link between two parts of a system are established only at runtime, not at compile-time. The [Callback](https://en.wikipedia.org/wiki/Callback_(computer_science)) article has more information about delegation.

The authors also discuss so-called **parameterized types**, which are also known as [generics](https://en.wikipedia.org/wiki/Generic_programming) (Ada, Eiffel, [Java](https://en.wikipedia.org/wiki/Generics_in_Java), C#, VB.NET, and Delphi) or templates (C++). These allow any type to be defined without specifying all the other types it uses—the unspecified types are supplied as 'parameters' at the point of use.

The authors admit that delegation and parameterization are very powerful but add a warning:

"Dynamic, highly parameterized software is harder to understand and build than more static software." (Gang of Four 1995:21)

The authors further distinguish between '[Aggregation](https://en.wikipedia.org/wiki/Object_composition#Aggregation)', where one object 'has' or 'is part of' another object (implying that an aggregate object and its owner have identical lifetimes) and **acquaintance**, where one object merely 'knows of' another object. Sometimes acquaintance is called 'association' or the 'using' relationship. Acquaintance objects may request operations of each other, but they aren't responsible for each other. Acquaintance is a weaker relationship than aggregation and suggests much [looser coupling](https://en.wikipedia.org/wiki/Loose_coupling) between objects, which can often be desirable for maximum maintainability in a design.

The authors employ the term 'toolkit' where others might today use 'class library', as in C# or Java. In their parlance, toolkits are the object-oriented equivalent of subroutine libraries, whereas a '[framework](https://en.wikipedia.org/wiki/Software_framework)' is a set of cooperating classes that make up a reusable design for a specific class of software. They state that **applications** are hard to design, **toolkits** are harder, and **frameworks** are the hardest to design.

## Case study, Chapter 2

Chapter 2 is a step-by-step case study on "the design of a '[What-You-See-Is-What-You-Get](https://en.wikipedia.org/wiki/WYSIWYG)' (or 'WYSIWYG') document editor called Lexi." (pp33)

The chapter goes through seven problems that must be addressed in order to properly design Lexi, including any constraints that must be followed. Each problem is analyzed in depth, and solutions are proposed. Each solution is explained in full, including [pseudo-code](https://en.wikipedia.org/wiki/Pseudo-code) and a slightly modified version of [Object Modeling Technique](https://en.wikipedia.org/wiki/Object_Modeling_Technique) where appropriate.

Finally, each solution is associated directly with one or more design patterns. It is shown how the solution is a direct implementation of that design pattern.

The seven problems (including their constraints) and their solutions (including the pattern(s) referenced), are as follows:

### Document Structure[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=4)]

The document is "an arrangement of basic graphical elements" such as characters, lines, other shapes, etc., that "capture the total information content of the document"(pp35). The structure of the document contains a collection of these elements, and each element can in turn be a substructure of other elements.

**Problems and Constraints**

1. Text and graphics should be treated the same way (that is, graphics aren't a derived instance of text, nor vice versa)
2. The implementation should treat complex and simple structures the same way. It should not have to know the difference between the two.
3. Specific derivatives of abstract elements should have specialized analytical elements.

**Solution and Pattern**

A *recursive composition* is a hierarchical structure of elements, that builds "increasingly complex elements out of simpler ones" (pp36). Each node in the structure knows of its own children and its parent. If an operation is to be performed on the whole structure, each node calls the operation on its children (recursively).

This is an implementation of the [composite pattern](https://en.wikipedia.org/wiki/Composite_pattern), which is a collection of nodes. The node is an [abstract base class](https://en.wikipedia.org/wiki/Abstract_type), and derivatives can either be leaves (singular), or collections of other nodes (which in turn can contain leaves or collection-nodes). When an operation is performed on the parent, that operation is recursively passed down the hierarchy.

### Formatting[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=5)]

Formatting differs from structure. Formatting is a method of constructing a particular instance of the document's physical structure. This includes breaking text into lines, using hyphens, adjusting for margin widths, etc.

**Problems and Constraints**

1. Balance between (formatting) quality, speed and storage space
2. Keep formatting independent (uncoupled) from the document structure.

**Solution and Pattern**

A *Compositor* class will encapsulate the algorithm used to format a composition. Compositor is a subclass of the primitive object of the document's structure. A Compositor has an associated instance of a Composition object. When a Compositor runs its Compose(), it iterates through each element of its associated Composition, and rearranges the structure by inserting Row and Column objects as needed.

The Compositor itself is an abstract class, allowing for derivative classes to use different formatting algorithms (such as double-spacing, wider margins, etc.)

The [Strategy Pattern](https://en.wikipedia.org/wiki/Strategy_pattern) is used to accomplish this goal. A Strategy is a method of encapsulating multiple algorithms to be used based on a changing context. In this case, formatting should be different, depending on whether text, graphics, simple elements, etc., are being formatted.

### Embellishing the User Interface[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=6)]

The ability to change the graphical interface that the user uses to interact with the document.

**Problems and Constraints**

1. Demarcate a page of text with a border around the editing area
2. Scroll bars that let the user view different parts of the page
3. User interface objects should not know about the embellishments
4. Avoid an "explosion of classes" that would be caused by subclassing for "every possible combination of embellishments" and elements (p44)

**Solution and Pattern**

The use of a *transparent enclosure* allows elements that augment the behaviour of composition to be added to a composition. These elements, such as Border and Scroller, are special subclasses of the singular element itself. This allows the composition to be augmented, effectively adding state-like elements. Since these augmentations are part of the structure, their appropriate Operation() will be called when the structure's Operation() is called. This means that the client does not need any special knowledge or interface with the structure in order to use the embellishments.

This is a [Decorator pattern](https://en.wikipedia.org/wiki/Decorator_pattern), one that adds responsibilities to an object without modifying the object itself.

### Supporting Multiple Look-And-Feel Standards[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=7)]

[Look-and-feel](https://en.wikipedia.org/wiki/Look_and_feel) refers to [platform](https://en.wikipedia.org/wiki/Platform_(computing))-specific UI standards. These standards "define guidelines for how applications appear and react to the user" (pp47).

**Problems and Constraints**

1. The editor must implement standards of multiple platforms so that it is [portable](https://en.wikipedia.org/wiki/Porting)
2. Easily adapt to new and emergent standards
3. Allow for run-time changing of look-and-feel (i.e.: No [hard-coding](https://en.wikipedia.org/wiki/Hard_coded))
4. Have a set of abstract elemental subclasses for each category of elements (ScrollBar, Buttons, etc.)
5. Have a set of concrete subclasses for each abstract subclass that can have a different look-and-feel standard. (ScrollBar having MotifScrollBar and PresentationScrollBar for Motif and Presentation look-and-feels)

**Solution and Pattern**

Since object creation of different concrete objects cannot be done at runtime, the object creation process must be abstracted. This is done with an abstract guiFactory, which takes on the responsibility of creating UI elements. The abstract guiFactory has concrete implementations, such as MotifFactory, which creates concrete elements of the appropriate type (MotifScrollBar). In this way, the program need only ask for a ScrollBar and, at run-time, it will be given the correct concrete element.

This is an [Abstract Factory](https://en.wikipedia.org/wiki/Abstract_factory_pattern). A regular factory creates concrete objects of one type. An abstract factory creates concrete objects of varying types, depending on the concrete implementation of the factory itself. Its ability to focus on not just concrete objects, but entire *families* of concrete objects "distinguishes it from other creational patterns, which involve only one kind of product object" (pp51).

### Supporting Multiple Window Systems[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=8)]

Just as look-and-feel is different across platforms, so is the method of handling [windows](https://en.wikipedia.org/wiki/Window_(computing)). Each platform displays, lays out, handles input to and output from, and layers windows differently.

**Problems and Constraints**

1. The document editor must run on many of the "important and largely incompatible window systems" that exist (p. 52)
2. An Abstract Factory cannot be used. Due to differing standards, there will not be a common abstract class for each type of widget.
3. Do not create a new, nonstandard windowing system

**Solution and Pattern**

It is possible to develop "our own abstract and concrete product classes", because "all window systems do generally the same thing" (p. 52). Each window system provides operations for drawing primitive shapes, iconifying/de-iconifying, resizing, and refreshing window contents.

An abstract base Window class can be derived to the different types of existing windows, such as application, [iconified](https://en.wikipedia.org/w/index.php?title=Iconification&action=edit&redlink=1), dialog. These classes will contain operations that are associated with windows, such as reshaping, graphically refreshing, etc. Each window contains elements, whose Draw() functions are called upon by the Window's own draw-related functions.

In order to avoid having to create platform-specific Window subclasses for every possible platform, an interface will be used. The Window class will implement a Window implementation (WindowImp) abstract class. This class will then in turn be derived into multiple platform-specific implementations, each with platform-specific operations. Hence, only one set of Window classes are needed for each type of Window, and only one set of WindowImp classes are needed for each platform (rather than the [Cartesian product](https://en.wikipedia.org/wiki/Cartesian_product) of all available types and platforms). In addition, adding a new window type does not require any modification of platform implementation, or vice versa.

This is a [Bridge pattern](https://en.wikipedia.org/wiki/Bridge_pattern). Window and WindowImp are different, but related. Window deals with windowing in the program, and WindowImp deals with windowing on a platform. One of them can change without ever having to modify the other. The Bridge pattern allows these two "separate class hierarchies to work together even as they evolve independently" (p. 54).

### User Operations[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=9)]

All actions the user can take with the document, ranging from entering text, changing formatting, quitting, saving, etc.

**Problems and Constraints**

1. Operations must be accessed through different inputs, such as a menu option and a keyboard shortcut for the same command
2. Each option has an interface, which should be modifiable
3. Operations are implemented in several different classes
4. In order to avoid coupling, there must not be a lot of dependencies between implementation and user interface classes.
5. Undo and redo commands must be supported on most document changing operations, with [no arbitrary limit](https://en.wikipedia.org/wiki/Zero_one_infinity_rule) on the number of levels of undo
6. Functions are not viable, since they don't undo/redo easily, are not easily associated with a state, and are hard to extend or reuse.
7. Menus should be treated like hierarchical composite structures. Hence, a menu is a menu item that contains menu items which may contain other menu items, etc.

**Solution and Pattern**

Each menu item, rather than being instantiated with a list of parameters, is instead done with a *Command* object.

Command is an abstract object that only has a single abstract Execute() method. Derivative objects extend the Execute() method appropriately (i.e., the PasteCommand.Execute() would utilize the content's clipboard buffer). These objects can be used by widgets or buttons just as easily as they can be used by menu items.

To support undo and redo, Command is also given Unexecute() and Reversible(). In derivative classes, the former contains code that will undo that command, and the latter returns a boolean value that defines if the command is undoable. Reversible() allows some commands to be non-undoable, such as a Save command.

All executed Commands are kept in a list with a method of keeping a "present" marker directly after the most recently executed command. A request to undo will call the Command.Unexecute() directly before "present", then move "present" back one command. Conversely, a Redo request will call Command.Execute() after "present", and move "present" forward one.

This Command approach is an implementation of the [Command pattern](https://en.wikipedia.org/wiki/Command_pattern). It encapsulates requests in objects, and uses a common interface to access those requests. Thus, the client can handle different requests, and commands can be scattered throughout the application.

### Spell Check and Hyphenation[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=10)]

This is the document editor's ability to textually analyze the contents of a document. Although there are many analyses that can be performed, spell check and hyphenation-formatting are the focus.

**Problems and Constraints**

1. Allow for multiple ways to check spelling and identify places for hyphenation
2. Allow for expansion for future analysis (e.g., word count, grammar check)
3. Be able to iterate through a text's contents without access to the text's actual structure (e.g., array, linked list, string)
4. Allow for any manner of traversal of document (beginning to end, end to beginning, alphabetical order, etc.)

**Solution and Pattern**

Removing the integer-based index from the basic element allows for a different iteration interface to be implemented. This will require extra methods for traversal and object retrieval. These methods are put into an abstract Iterator interface. Each element then implements a derivation of the Iterator, depending on how that element keeps its list (ArrayIterator, LinkListIterator, etc.).

Functions for traversal and retrieval are put into the abstract Iterator interface. Future Iterators can be derived based on the type of list they will be iterating through, such as Arrays or Linked Lists. Thus, no matter what type of indexing method any implementation of the element uses, it will have the appropriate Iterator.

This is an implementation of the [Iterator pattern](https://en.wikipedia.org/wiki/Iterator_pattern). It allows the client to traverse through any object collection, without needing to access the contents of the collection directly, or be concerned about the type of list the collection's structure uses.

Now that traversal has been handled, it is possible to analyze the elements of a structure. It is not feasible to build each type of analysis into the element structure themselves; every element would need to be coded, and much of the code would be the same for similar elements.

Instead, a generic CheckMe() method is built into the element's abstract class. Each Iterator is given a reference to a specific algorithm (such as spell check, grammar check, etc.). When that Iterator iterates through its collection, it calls each element's CheckMe, passing the specified algorithm. CheckMe then passes a reference to its element back to said algorithm for analysis.

Thus, to perform a spell check, a front-to-end iterator would be given a reference to a SpellCheck object. The iterator would then access each element, executing its CheckMe() method with the SpellCheck parameter. Each CheckMe would then call the SpellCheck, passing a reference to the appropriate element.

In this manner, any algorithm can be used with any traversal method, without hard-code coupling one with the other. For example, Find can be used as "find next" or "find previous", depending on if a "forward" iterator was used, or a "backwards" iterator.

In addition, the algorithms themselves can be responsible for dealing with different elements. For example, a SpellCheck algorithm would ignore a Graphic element, rather than having to program every Graphic-derived element to not send themselves to a SpellCheck.

## Patterns by Type

### Creational[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=12)]

*Main article:* [*Creational pattern*](https://en.wikipedia.org/wiki/Creational_pattern)

[Creational patterns](https://en.wikipedia.org/wiki/Creational_pattern) are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

* [Abstract factory pattern](https://en.wikipedia.org/wiki/Abstract_factory_pattern) groups object factories that have a common theme.
* [Builder pattern](https://en.wikipedia.org/wiki/Builder_pattern) constructs complex objects by separating construction and representation.
* [Factory method pattern](https://en.wikipedia.org/wiki/Factory_method_pattern) creates objects without specifying the exact class to create.
* [Prototype pattern](https://en.wikipedia.org/wiki/Prototype_pattern) creates objects by cloning an existing object.
* [Singleton pattern](https://en.wikipedia.org/wiki/Singleton_pattern) restricts object creation for a class to only one instance.

### Structural[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=13)]

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

* [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern) allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
* [Bridge](https://en.wikipedia.org/wiki/Bridge_pattern) decouples an abstraction from its implementation so that the two can vary independently.
* [Composite](https://en.wikipedia.org/wiki/Composite_pattern) composes zero-or-more similar objects so that they can be manipulated as one object.
* [Decorator](https://en.wikipedia.org/wiki/Decorator_pattern) dynamically adds/overrides behaviour in an existing method of an object.
* [Facade](https://en.wikipedia.org/wiki/Facade_pattern) provides a simplified interface to a large body of code.
* [Flyweight](https://en.wikipedia.org/wiki/Flyweight_pattern) reduces the cost of creating and manipulating a large number of similar objects.
* [Proxy](https://en.wikipedia.org/wiki/Proxy_pattern) provides a placeholder for another object to control access, reduce cost, and reduce complexity.

### Behavioral[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=14)]

Most of these design patterns are specifically concerned with communication between **objects**.

* [Chain of responsibility](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern) delegates commands to a chain of processing objects.
* [Command](https://en.wikipedia.org/wiki/Command_pattern) creates objects which encapsulate actions and parameters.
* [Interpreter](https://en.wikipedia.org/wiki/Interpreter_pattern) implements a specialized language.
* [Iterator](https://en.wikipedia.org/wiki/Iterator_pattern) accesses the elements of an object sequentially without exposing its underlying representation.
* [Mediator](https://en.wikipedia.org/wiki/Mediator_pattern) allows [loose coupling](https://en.wikipedia.org/wiki/Loose_coupling) between classes by being the only class that has detailed knowledge of their methods.
* [Memento](https://en.wikipedia.org/wiki/Memento_pattern) provides the ability to restore an object to its previous state (undo).
* [Observer](https://en.wikipedia.org/wiki/Observer_pattern) is a publish/subscribe pattern which allows a number of observer objects to see an event.
* [State](https://en.wikipedia.org/wiki/State_pattern) allows an object to alter its behavior when its internal state changes.
* [Strategy](https://en.wikipedia.org/wiki/Strategy_pattern) allows one of a family of algorithms to be selected on-the-fly at runtime.
* [Template method](https://en.wikipedia.org/wiki/Template_method_pattern) defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
* [Visitor](https://en.wikipedia.org/wiki/Visitor_pattern) separates an algorithm from an object structure by moving the hierarchy of methods into one object.

## Criticism

Significant criticism has been directed at the concept of [software design patterns](https://en.wikipedia.org/wiki/Software_design_pattern) generally, and at *Design Patterns* specifically.

A primary criticism of *Design Patterns* is that its patterns are simply workarounds for missing features in C++, replacing elegant abstract features with lengthy concrete patterns, essentially becoming a "human compiler" or "generating by hand the expansions of some macro".[[4]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-Graham2002-4) [Peter Norvig](https://en.wikipedia.org/wiki/Peter_Norvig) demonstrates that 16 out of the 23 patterns in *Design Patterns* are simplified or eliminated (via direct language support) in [Lisp](https://en.wikipedia.org/wiki/Lisp_(programming_language)) or [Dylan](https://en.wikipedia.org/wiki/Dylan_(programming_language)).[[5]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-Norvig1998-5) Related observations were made by Hannemann and [Kiczales](https://en.wikipedia.org/wiki/Gregor_Kiczales) who implemented several of the 23 design patterns using an [aspect-oriented programming language](https://en.wikipedia.org/wiki/Aspect-oriented_programming) (AspectJ) and showed that code-level dependencies were removed from the implementations of 17 of the 23 design patterns and that aspect-oriented programming could simplify the implementations of design patterns.[[6]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-Hannemann_2002-6)

Paul Graham wrote:[[4]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-Graham2002-4)

When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough-- often that I'm generating by hand the expansions of some macro that I need to write.

There has also been humorous criticism, such as a show trial at OOPSLA '99 on 3 November 1999,[[7]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-7)[[8]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-8)[[a]](https://en.wikipedia.org/wiki/Design_Patterns#cite_note-9) and a parody of the format, by [Jim Coplien](https://en.wikipedia.org/wiki/Jim_Coplien), entitled "[Kansas City Air Conditioner](http://c2.com/cgi/wiki?KansasCityAirConditioner)".