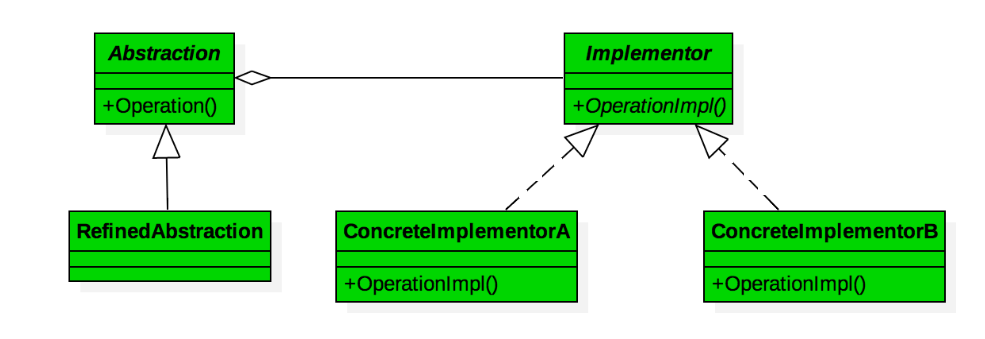
*Definition:*

***Separates an object's interface from its implementation***

*The Bridge Pattern allows you to vary the implementation and the abstraction by placing the two in separate class hierarchies.*

The Bridge design pattern allows you to separate the abstraction from the implementation. There are 2 parts in Bridge design pattern:

1. **Abstraction**
2. **Implementation**



**This is a design mechanism that encapsulates an implementation class inside of an interface class.**

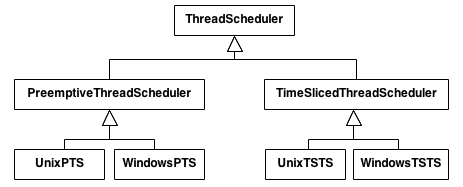
* The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
* The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.
* The abstraction contains a reference to the implementor. Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors. Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction’s implementor at run-time. Changes to the implementor do not affect client code.
* It increases the loose coupling between class abstraction and its implementation.

***Elements of Bridge Design Pattern***

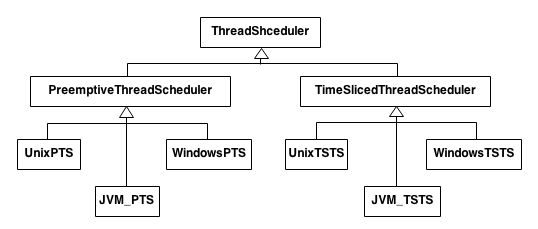
* **Abstraction** – Core of the bridge design pattern and defines the crux. Contains a reference to the implementer. It can also acts as the base class for other abstractions.
* **Refined Abstraction** - This is a class which inherits from the Abstraction class. It extends the interface defined by Abstraction class.
* **Implementer/Bridge -** It defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different. *This is an interface which acts as a bridge between the abstraction class and implementer classes* and makes the functionality of implementer class independent from the abstraction class.
* **Concrete Implementation -** Implements the above implementer by providing concrete implementation.

**Consider the domain of "thread scheduling"**

There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions.

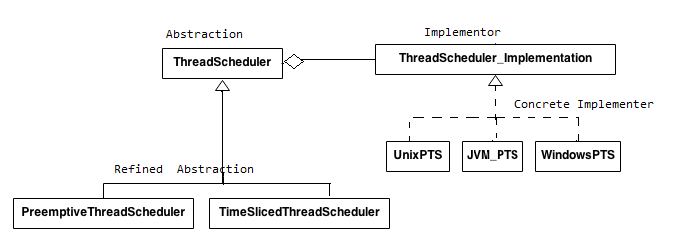


Let suppose, If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like-

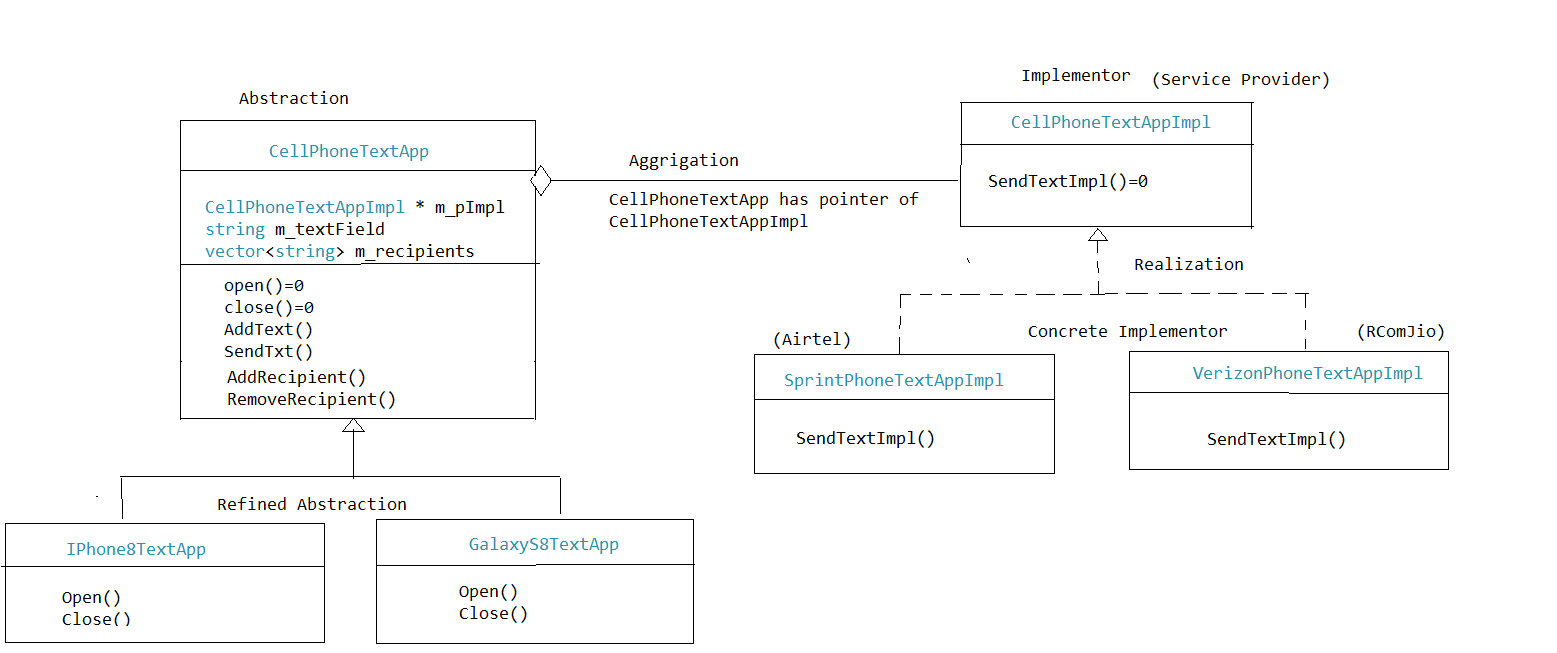


What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.

The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Example:01



Let’s begin with an abstract class ***CellPhoneTextApp*** as our ***Abstraction*** class with two ***RefinedAbstraction*** subclasses, ***GalaxyS8TextApp*** and ***IPhone8TextApp.***

Each would implement the abstract methods, *Open()* and *Close()* of their parent class for opening and closing the application, respectively.

The behavior for sending a text would be delegated to an *Implementor* interface, ***CellPhoneTextAppImpl***. Implementations of this interface, ***VerizonPhoneTextAppImpl*** and ***SprintPhoneTextAppImpl*** would vary based on service providers, Verizon and Sprint, respectively.

In this example, these concrete implementors are responsible for defining the behavior of sending a text via ***SendTextImpl()*.** The relationship between the ***CellPhoneTextApp*** and the ***CellPhoneTextAppImpl*** is the bridge part of the pattern.

class CellPhoneTextApp {

public:

CellPhoneTextApp(CellPhoneTextAppImpl \* impl) :m\_pImpl(impl) {}

virtual ~CellPhoneTextApp() {}

virtual void Open() = 0;

virtual void Close() = 0;

virtual void AddText(std::string textField) {

m\_textField = textField;

}

virtual void SendText() {

m\_pImpl->SendTextImpl(m\_recipients, m\_textField);

}

virtual void AddRecipient(std::string recipient) {

m\_recipients.push\_back(recipient);

}

virtual void RemoveRecipient(std::string recipient) {

auto iter = find(m\_recipients.begin(), m\_recipients.end(), recipient);

if (iter != m\_recipients.end())

{

m\_recipients.erase(iter);

}

}

private:

CellPhoneTextAppImpl \* m\_pImpl;

std::string m\_textField;

std::vector<std::string> m\_recipients;

};

class IPhone8TextApp : public CellPhoneTextApp {

public:

IPhone8TextApp(CellPhoneTextAppImpl \* impl) : CellPhoneTextApp(impl) {}

virtual ~IPhone8TextApp(){}

virtual void Open() {

std::cout << "Opening IPhone 8 Text App.\n\n";

}

virtual void Close() {

std::cout << "Closing IPhone 8 Text App.\n\n";

}

};

class GalaxyS8TextApp : public CellPhoneTextApp {

public:

GalaxyS8TextApp(CellPhoneTextAppImpl \* impl) : CellPhoneTextApp(impl) { }

virtual ~GalaxyS8TextApp(){}

virtual void Open() {

std::cout << "Opening Galaxy S8 Text App.\n\n";

}

virtual void Close() {

std::cout << "Closing Galaxy S8 Text App.\n\n";

}

};

class CellPhoneTextAppImpl

{

public:

virtual void SendTextImpl(std::vector<std::string> recipients,

std::string textField) = 0;

};

class SprintPhoneTextAppImpl : public CellPhoneTextAppImpl {

public:

SprintPhoneTextAppImpl(){}

virtual ~SprintPhoneTextAppImpl(){}

virtual void SendTextImpl(std::vector<std::string> recipients,

std::string textField) {

std::cout << "Sending message...\n\n";

std::cout << std::string("\"") << textField << std::string("\"")<<"\n";

std::cout << "To recipients : \n";

for (auto iter = recipients.begin();

iter != recipients.end(); ++iter)

{

std::cout << " " << \*iter << "\n";

}

std::cout << "\nover Sprint network\n\n";

}

};

class VerizonPhoneTextAppImpl : public CellPhoneTextAppImpl {

public:

VerizonPhoneTextAppImpl(){}

virtual ~VerizonPhoneTextAppImpl(){}

virtual void SendTextImpl(std::vector<std::string> recipients,

std::string textField) {

std::cout << "Sending message...\n\n";

std::cout << std::string("\"") << textField << std::string("\"") <<"\n";

std::cout << "To recipients : \n";

for (auto iter = recipients.begin();

iter != recipients.end(); ++iter)

{

std::cout << " " << \*iter << "\n";

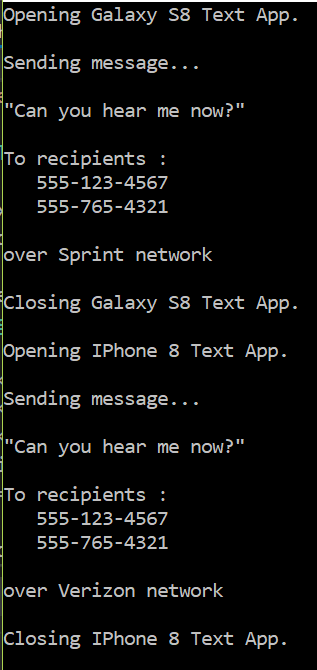
}

std::cout << "\nover Verizon network\n\n";

}

};

|  |
| --- |
| void SendText(CellPhoneTextApp \* textApp) |
|  | { |
|  | textApp->Open(); |
|  |  |
|  | textApp->AddRecipient(std::string("555-123-4567")); |
|  | textApp->AddRecipient(std::string("555-765-4321")); |
|  |  |
|  | textApp->AddText(std::string("Can you hear me now?")); |
|  |  |
|  | textApp->SendText(); |
|  |  |
|  | textApp->Close(); |
|  | } |
|  |  |
|  |  |
|  | int main() |
|  | { |
|  |  |
|  | SprintPhoneTextAppImpl sprintTextAppImpl; |
|  |  |
|  | GalaxyS8TextApp galaxyS8TextApp(&sprintTextAppImpl); |
|  |  |
|  |  |
|  | SendText(&galaxyS8TextApp); |
|  |  |
|  |  |
|  | VerizonPhoneTextAppImpl verizonTextAppImpl; |
|  |  |
|  |  |
|  | IPhone8TextApp iPhone8TextApp(&verizonTextAppImpl); |
|  |  |
|  | SendText(&iPhone8TextApp); |
|  |  |
|  |  |
|  | getchar(); |
|  | return 0; |
|  | } |



Applicability

The bridge pattern applies when there is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently. Using the bridge pattern would leave the client code unchanged with no need to recompile the code.

The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time)

Use the Bridge pattern when:

* When we want run-time binding of the implementation.
* When we have a proliferation of classes resulting from a coupled interface and numerous implementations.
* When we want to share an implementation among multiple objects,

Benefits of the Bridge Pattern

* The abstraction and the implementation are decoupled. This adds the benefit of configuring and/or changing the implementation at runtime.
* The decoupling allows the client and the abstraction to only depend on interface to the implementations, not the implementations themselves. So changing the implementations doesn’t force the client and abstraction to recompile.
* Easier to extend the abstraction and the interface to the implementations since they are separate.

Related Patterns

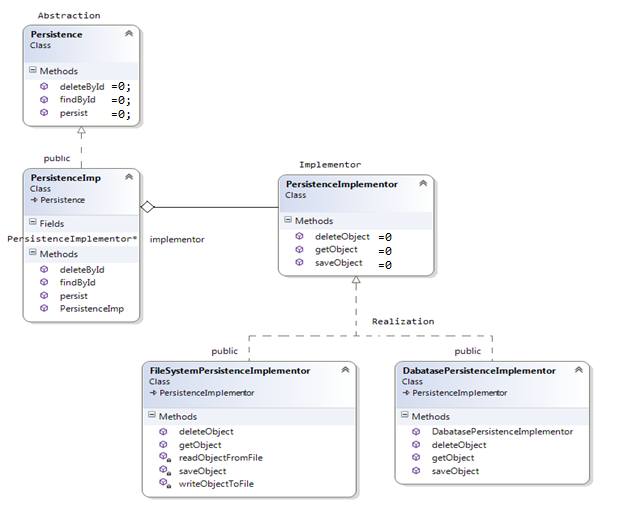
* **Abstract Factory Pattern** - An Abstract Factory pattern can be used create and configure a particular Bridge, for example a factory can choose the suitable concrete implementer at runtime.
* **Adapter** makes things work after they're designed; Bridge makes them work before they are.
* **Bridge** is designed up-front to let the abstraction and the implementation vary independently. **Adapter** is retrofitted to make unrelated classes work together.
* **State Strategy, Bridge** (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.
* If interface classes delegate the creation of their implementation classes (instead of creating/coupling them directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

## Consequences

* Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured and even switched at run-time.
* Abstraction and Implementer hierarchies can be extended independently.

Example-02

This Example has minor change in relationship in between Abstraction and Refined Abstraction. Here Abstraction class is abstract class (like interface). So I had my pointer of Implementer instance in refined abstraction class.



class Employee {};

class Persistence {

public:

virtual void persist(Employee object) = 0;

virtual Employee findById(long objectId) = 0;

virtual void deleteById(long id) = 0;

};

class PersistenceImplementor {

public:

virtual void saveObject(Employee object) = 0;

virtual void deleteObject(long objectId) = 0;

virtual Employee getObject(long objectId) = 0;

};

class PersistenceImp :public Persistence {

private :

PersistenceImplementor \*implementor;

public:

PersistenceImp(PersistenceImplementor \*imp) {

this->implementor = imp;

}

void deleteById(long id) {

implementor->deleteObject(id);

}

Employee findById(long objectId) {

return implementor->getObject(objectId);

}

void persist(Employee object) {

implementor->saveObject(object);

}

};

class FileSystemPersistenceImplementor:public PersistenceImplementor {

public:

void deleteObject(long objectId) {

//File \*fp = fopen("/persistence/txtFile.dat","a+");

//Find the store object in file based on input parameter objectId;

//delete that object;

}

Employee getObject(long objectId) {

//File \*fp = fopen("/persistence/txtFile.dat","r");

//return readObjectFromFile(fp, objectId);

}

void saveObject(Employee object) {

// open file in append mode

FILE \*fp = fopen("/persistence/txtFile.dat","a+");

// write file to Stream

writeObjectToFile(fp, object);

}

private:

Employee readObjectFromFile(FILE\* fp, objectId) {

// open file

// and load object

// return the object

}

void writeObjectToFile(FILE \*fp, Employee object) {

// serialize object and write it to file

}

};

class DabatasePersistenceImplementor:public PersistenceImplementor {

public:

DabatasePersistenceImplementor() {

// load database driver

}

void deleteObject(long objectId) {

// open database connection

// remove record

}

Employee getObject(long objectId) {

// open database connection

// read records

// create object from record and return.

}

void saveObject(Employee object) {

// open database connection

// create records for fields inside the object

}

};

bool databaseDriverExists() {

//if exist return true.

return false;

}

The code below illustrates a persistence API driver, note how the choice of concrete implementor does not affect the client code , note also that extending persistence does not affect the implementor and extending the implementor does not extend the persistence.

int main() {

// this program needs a persistence framework

// at runtime an implementor is chosen between file system implementation and

//database implememtor , depending on existence of databse drivers

PersistenceImplementor \*implementor;

if (databaseDriverExists())

implementor = new DabatasePersistenceImplementor();

else

implementor = new FileSystemPersistenceImplementor();

Persistence \*persistenceAPI = new PersistenceImp(implementor);

Employee o = persistenceAPI->findById(12343755);

// do changes to the object

// then persist

persistenceAPI->persist(o);

// can also change implementor

persistenceAPI = new PersistenceImp(new DabatasePersistenceImplementor());

persistenceAPI->deleteById(2323);

}

### Graphical User Interface Frameworks

Graphical User Interface Frameworks use the bridge pattern to separate abstractions from platform specific implementation. For example GUI frameworks separate a Window abstraction from a Window implementation for Linux or Mac OS using the bridge pattern.