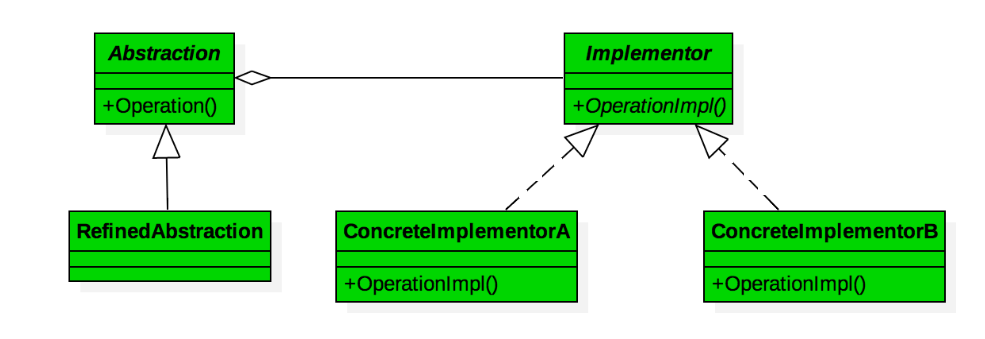
Definition. Separates an object's interface from its implementation

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



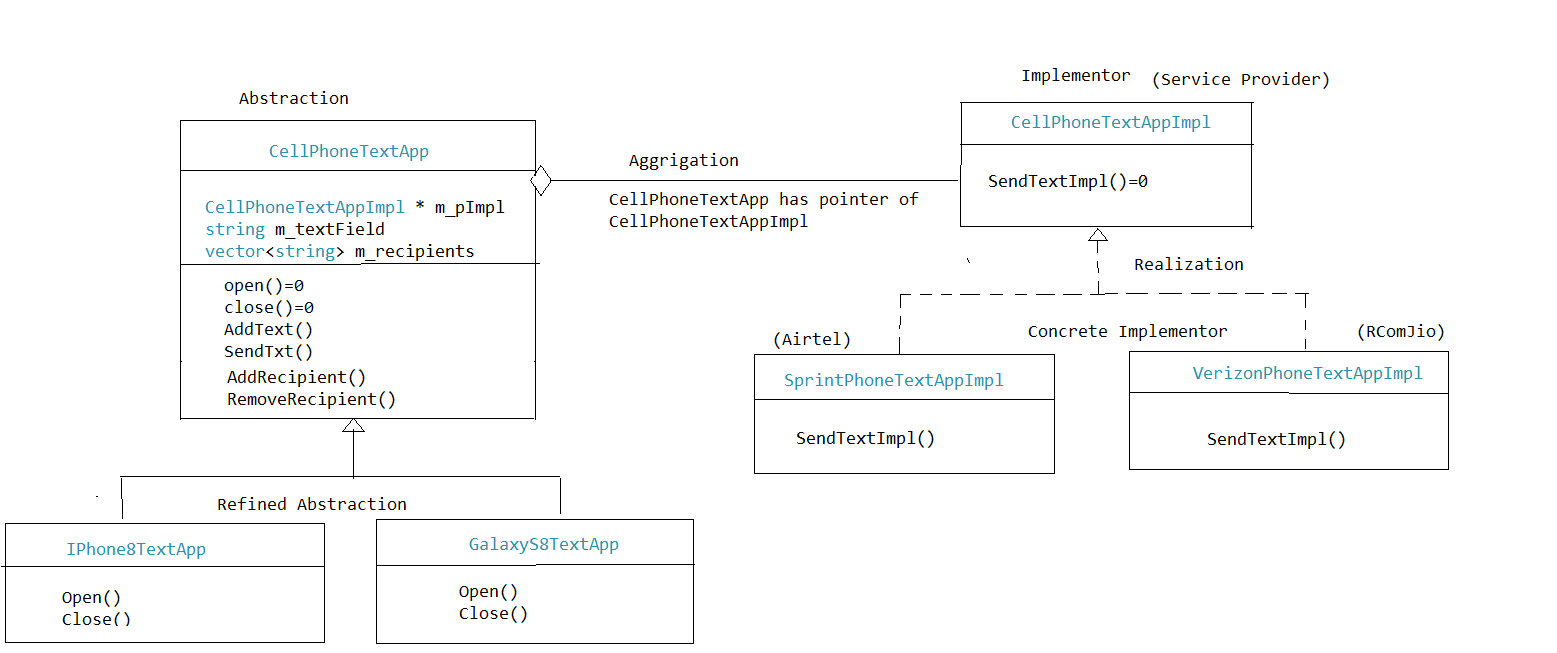
This is a design mechanism that encapsulates an implementation class inside of an interface class.

* 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.

Example:



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\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\n";

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

for (auto iter = recipients.begin();

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

{

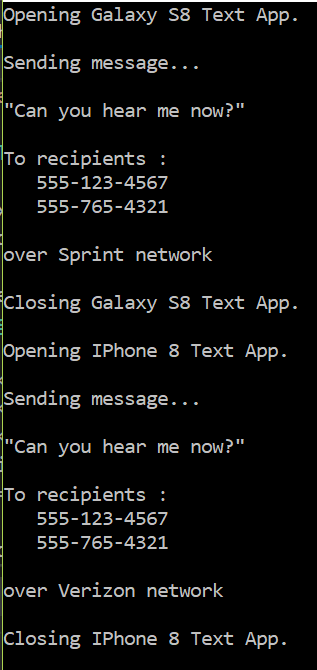
std::cout << " " << \*iter << "\n";

}

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

}

};



Benefits of the Bridge Pattern

There are a few benefits when using the Bridge Pattern. First, The abstraction and the implementation are decoupled. This adds the benefit of configuring and/or changing the implementation at runtime. In addition, 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.

Another benefit to using the Bridge Pattern is that its easier to extend the abstraction and the interface to the implementations since they are separate.

## **Applicability & Examples**

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.

### 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.

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 implementor at runtime.

## **Consequences**

Known Uses:

* 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 Implementor hierarchies can be extended independently.

### Known Uses:

* GUI frameworks as discussed previously.
* Persistence Frameworks as discussed previously.