Bridge Pattern

The bridge design pattern is a structural design pattern used to decouple an abstract implementation of controller from the implementations of the object being controlled - making it possible to have multiple implementations of the controlling object as well as multiple implementations of the controlled object. This allows the two hierarchies of abstractions to evolve separately.

Defining characteristics

The bridge main defining characteristic of the bridge pattern is that at one end there is an abstract interface to the controlling abstraction and on the other end there are several abstract implementations of objects being controlled by the same controlling abstraction.

- **Decoupling of abstractions and implementations**: this is done both at the interface towards the user, and at the point where the controlling implementation interfaces with the object being controlled. Both interfaces must be abstract interfaces (ie. must use the Virtual API pattern).
- **Data flows through a 'bridge'**: Classes are extended by controlling an interface on one end of the bridge that controls an implementation on the other end of the bridge. Hence being called the 'bridge' pattern.
- **Two distinct hierarchies**: the pattern is most clear when you can see two different hierarchies that are loosely coupled over a 'bridge' through which data flows. The loosely coupled hierarchies are a positive consequence of the bridge pattern and in fact what the bridge pattern is trying to achieve in the first place.

Use cases

One of the most well known uses of the bridge pattern is in how 3D games used to implement OpenGL bindings. During the early days of 3D gaming, the OpenGL implementations varied vastly and every graphics card came with it's own OpenGL shared library that provided the functions for accessing the hardware. Unfortunately, many of these implementations were providing some methods and missing others.

The solution to all of this in the 90s and early 20s was to define a global set of OpenGL functions that the game was using and then manually resolve these symbols by loading the OpenGL shared library and resolving each symbol in sequence - while filling in the missing symbols with workarounds or simply null implementations. The resulting global OpenGL function pointers represented the functions that the game could then directly call when it needed to interface with the GPU hardware.

This example illustrates the crudest, rawest form of the bridge design pattern that allowed graphics hardware and drivers to evolve independently from the game itself and the game could stay compatible with many revisions of the OpenGL shared library shipped by the card vendor.

In practice, when we use the bridge design pattern in C, we do it in situations where one of the following is true:

- **Abstraction and implementation need to vary**: The Bridge Design Pattern is ideal when the requirements for the abstraction and implementation are likely to change independently. The beauty of a virtual method table is that we can pass it across DLL boundaries. This allows us to define the implementation in one compilation unit, export the abstraction and then have another implementation use this abstraction in another compilation unit (shared library).
- **Multiple implementations**: The bridge design pattern is also ideal when there are multiple implementations possible for a single abstraction and the implementation needs to be chosen dynamically at runtime. A good example of this is the old Quake3 engine where the engine itself could evolve separately from the game dll and the game was calling into the engine using an exported abstract interface.
- **Abstracted Access to Hardware or Platform Services**: The Bridge Design Pattern is ideal when the system needs to access hardware or platform services, as it provides a way to abstract these services from the rest of the system and allow for different implementations to be used on different platforms. An example of this use case is the Linux System Call interface and the corresponding C library that provides an abstraction that is actually seen by the application.
- **Plug-In Architecture**: The Bridge Design Pattern is ideal when the system needs to support plug-ins or add-ons, as it provides a way to create a plug-in architecture that allows for new implementations to be added to the system without affecting the rest of the system. The old Quake games were actually implemented as "plugins" to the game engine which was compiled as the main executable and the game itself was a dynamic library.

Perhaps one of the biggest differences between bridge and an abstract interface is that the bridge is a "thin pathway" between one system to another. Just like the Linux System Call interface is a very constrained pathway between the internals of the kernel and all other software running on the system - the bridge pattern also has similar properties. The implementation provides the abstract interface to be bridged (this would be the linux system call interface) and then on top of that we provide another implementation with its own abstraction that provides a rich interface to the user (this would be different libc implementations that exist on top of the system call interface - musl, glibc, newlib etc).

Another example you may want to consider to understand this pattern better is the case where we provide a programming interface for an object that does not reside on the same machine. For example, we can have an abstract interface for a "CurrentSensor" available to core running on a Linux gateway. The concrete implementation of this CurrentSensor may involve reading a value over bluetooth using a request-response protocol. The actual sensor is in turn connected to a chip on the other end of the bluetooth connection. Our bridge here starts with the abstract interface for CurrentSensor, then the concrete implementation that makes the request over bluetooth, then the bluetooth current sensor GATT service implementation which finally communicates with the actual implementation that reads the value from the hardware current sensor.

The bridge is a transparent call path that makes these kinds of complex abstractions simple to interact with. Our local implementation can choose to cache the value or filter it or only update it at fixed intervals. To our user application it simply does not matter and most importantly, the

bridge makes sure that the interface looks very simply to the user application hiding all of the complexities involved in implementing it while still also ensuring that implementations on the other side of the bridge (in this case of the bluetooth connection) can change freely so long as the bridge protocol stays compatible.

Benefits

The most prominent benefit of this pattern is that it gives you a workflow for connecting two loosely related hierarchies of objects without mixing these objects together.

- **Bridge between two object hierarchies**: you have two representations of complex data structures with a loosely coupled interface between them.
- **Dependency separation**: the concrete implementation of one hierarchy does not need to pull in dependencies of the other hierarchy (ex. in a game where every object has a physics body and a mesh body you can separate these two hierarchies and implement bridge pattern between them to keep them separate and possibly even on different machines).
- **Separation of concerns**: enables two hierarchies to be created representing different parts of the same system while allowing each hierarchy to evolve independently in terms of implementation.

Drawbacks

- **Increased complexity**: instead of just a single hierarchy and mixed dependencies you have to maintain two separate hierarchies and links between them.
- **Additional indirection**: since each operation involves calls through multiple interfaces, there is a potential with a lot of very "light" methods which simply make another call.
- **Increased development time**: due to additional data structures and classes that have to be implemented.

Like many other design patterns, the bridge pattern has it’s own uses and where it truly makes sense to have a bridge (such as in a client/server architecture where part of the same hierarchy is handled by the server) the bridge design pattern absolutely shines. The pitfalls only emerge if you try to use this pattern in situations where it is not really needed.

Implementation

In this section we are going to implement the bridge pattern in C and in Rust. You can find a working sample under samples/design-patterns/bridge in the Swedish Embedded Platform SDK.

To illustrate the main concept we are going to have the following setup:

- **Client/Server**: we will be creating two local hierarchies of objects where one represents the client structures and the other represents server structures. We will use the example of physics and drawable objects.
- **Two hierarchies**: client hierarchy handles visual representation while server hierarchy

handles physical behaviors. We will implement both in Rust and C.

- **Client: Drawable**: the objects on client side implement **`Drawable`** interface which we will define.
- **Server: PhysicsObject**: the objects on the server side implement **`PhysicsObject`** interface.
- **User interacts with client objects**: finally we will allow the user to interact with client side instances and the bridge will connect these to the server side instance.

There are a few things that will happen in such a scenario where we call a virtual method "draw" upon an object implementing the **`Drawable`** interface. First we will have to make a call to the corresponding server object and get its position. Then we should draw our client side object.

In a real world scenario, the server would build its own hierarchy separately. And the client side server objects would in fact all have some sort of RPC implementation. We don't worry about that here so we set up our bindings between objects when creating them. To do this, we are going to provide a pointer to the server **`PhysicsObject`** interface when creating each concrete client object. We are assuming then that this interface points to a correct corresponding object that matches our client side representation.

The main addition in client code is that all of our objects are drawable - while on the server side they are just part of a simulation. Thus by using the bridge pattern we are able to separate these two potentially complex hierarchies into two different ones that can execute independently.

Server side objects

Server side objects

```
pub struct Circle {
}

pub struct Square {
}

pub struct Point2D(f32, f32);

impl Circle {
    pub fn new() -> Self {
        return Self { };
    }
}

impl Square {
    pub fn new() -> Self {
        return Self { };
    }
}
```

These are concrete objects on the server. Now we want to create a generic interface through which we want our client to access these objects. In such a scenario the client might want to

get position, rotation and other variables of the object but without necessarily knowing what object it is dealing with. Thus our interface would provide such functionality.

Client Interface

On the client, for simplicity's sake we will simply implement a method to get position.

Physics object interface

```
pub trait PhysicsObject {
    fn position(&self) -> Point2D;
}

impl PhysicsObject for Circle {
    fn position(&self) -> Point2D {
        printk("server: get position for circle\n");
        return Point2D(1.0, 1.0);
    }
}

impl PhysicsObject for Square {
    fn position(&self) -> Point2D {
        printk("server: get position for square\n");
        return Point2D(1.0, 1.0);
    }
}
```

It is through the **`PhysicsObject`** interface that our client objects would now be getting their physics state. The client side objects would in turn implement all logic and functionality for visually showing the state of the physics object. This will naturally involve functionality that is extending the simple physics object.

Client side objects

```
pub struct Circle<P: server::PhysicsObject> {
    physics: P,
}

pub struct Square<P: server::PhysicsObject> {
    physics: P,
}

impl<P: server::PhysicsObject> Circle<P> {
    pub fn new(p: P) -> Self {
        return Self { physics: p };
    }
}

impl<P: server::PhysicsObject> Square<P> {
    pub fn new(p: P) -> Self {
        return Self { physics: p };
    }
}
```

Notice how this separates physics from drawing. We have simplified the access to physics hierarchy to simply pass through the **`PhysicsObject`** interface. This is our bridge. In practice this can have a local implementation that could potentially have its own levels of complexity. For example, we may want to interpolate the local position of the physics object so that its position looks smooth to the client even when network updates lag behind. We can now easily do this as part of our bridge path and entirely without modifying either one of our main hierarchies.

Client side drawable interface

Moreover we can also modify the visual representation without modifying the physical representation and we can do the same in reverse. The bridge pattern is exactly about this style of separation.

```
pub trait Drawable {
    fn draw(&self);
}

impl<P: server::PhysicsObject> Drawable for Circle<P> {
    fn draw(&self){
        let _p = self.physics.position();
        printk("client: drawing circle at position\n");
    }
}

impl<P: server::PhysicsObject> Drawable for Square<P> {
    fn draw(&self){
        let _p = self.physics.position();
        printk("client: drawing square at position\n");
    }
}
```

Our client code now implements the corresponding **`Drawable`** interface for our client side hierarchy and we can now construct and use our objects easily.

Usage

```
use client::{Drawable};
let circle = client::Circle::new(server::Circle::new());
let square = client::Square::new(server::Square::new());
circle.draw();
square.draw();
```

The sample can be built for simulated cortex M3 platform and run using:

```
west build -b qemu_cortex_m3 samples/design-patterns/bridge/ -t run
```

The output will print:

```
server: get position for circle
client: drawing circle at position
server: get position for square
```

```
client: drawing square at position
```

Implementation In C

We can also implement the same sample in C and here is how we would do it.

First we are going to define our point2d:

```
struct point2d {
    uint32_t x, y;
};
```

Then let's define the physics object interface:

```
struct physics_object_ops {
    void (*position)(const struct physics_object_ops **ops, struct point2d *out);
};

typedef const struct physics_object_ops ** physics_object_t;

static inline void physics_object_position(const struct physics_object_ops **ops,
struct point2d *out){
    (*ops)->position(ops, out);
}
```

We can see already just how much more verbose C is in its implementation.

Server side objects

Let's continue to implement the server side objects:

```
struct server_circle {
    struct point2d pos;
    const struct physics_object_ops *ops;
};

static void server_circle_position(const struct physics_object_ops **ops, struct
point2d *out){
    struct server_circle *self = CONTAINER_OF(ops, struct server_circle, ops);
    printk("server: get position for circle (%d, %d)\n", self->pos.x, self->pos.y);
}

static const struct physics_object_ops circle_ops = {
    .position = server_circle_position
};

void server_circle_init(struct server_circle *self){
    memset(self, 0, sizeof(*self));
    self->ops = &circle_ops;
}
```

```
struct server_square {
    struct point2d pos;
    const struct physics_object_ops *ops;
```

```
};

static void server_square_position(const struct physics_object_ops **ops, struct
point2d *out){
    struct server_circle *self = CONTAINER_OF(ops, struct server_circle, ops);
    printk("server: get position for square (%d, %d)\n", self->pos.x, self->pos.y);
}

static const struct physics_object_ops square_ops = {
    .position = &server_square_position
};

void server_square_init(struct server_square *self){
    memset(self, 0, sizeof(*self));
    self->ops = &square_ops;
}
```

This gives us the server side hierarchy. One thing to note about C is that it does not have name spaces, so what we do instead is we prefix all our methods with what normally would be a name space name. This helps us keep our methods separate.

Drawable interface

Next let's implement our drawable interface:

```
struct drawable_ops {
    void (*draw)(const struct drawable_ops **ops);
};

typedef const struct drawable_ops ** drawable_t;

static inline void drawable_draw(const struct drawable_ops **ops){
    (*ops)->draw(ops);
}
```

Client side objects

And finally our client side objects:

```
struct client_circle {
    physics_object_t server_object;
    const struct drawable_ops *drawable;
};

void client_circle_draw(drawable_t ops){
    struct client_circle *self = CONTAINER_OF(ops, struct client_circle, drawable);
    struct point2d pos = {0};
    physics_object_position(self->server_object, &pos);
    printk("client: drawing circle at position\n");
}

static const struct drawable_ops circle_draw_ops = {
    .draw = client_circle_draw
};
```

```
void client_circle_init(struct client_circle *self, physics_object_t server_object){
    memset(self, 0, sizeof(*self));
    self->server_object = server_object;
    self->drawable = &circle_draw_ops;
}
```

```
struct client_square {
    physics_object_t server_object;
    const struct drawable_ops *drawable;
};

void client_square_draw(drawable_t ops){
    struct client_square *self = CONTAINER_OF(ops, struct client_square, drawable);
    struct point2d pos = {0};
    physics_object_position(self->server_object, &pos);
    printk("client: drawing square at position\n");
}

static const struct drawable_ops square_draw_ops = {
    .draw = client_square_draw
};

void client_square_init(struct client_square *self, physics_object_t server_object){
    memset(self, 0, sizeof(*self));
    self->server_object = server_object;
    self->drawable = &square_draw_ops;
}
```

Usage

We can now use this in much the same way as our original implementation in Rust: it's just a little more verbose as always.

```
struct server_circle s_circle;
struct server_square s_square;
struct client_circle circle;
struct client_square square;

server_circle_init(&s_circle);
server_square_init(&s_square);
client_circle_init(&circle, &s_circle.ops);
client_square_init(&square, &s_square.ops);
drawable_draw(&circle.drawable);
drawable_draw(&square.drawable);
```

This gives us two separate hierarchies of objects in C that can evolve separately and we can even maintain them inside two different libraries. The fact that they communicate over virtual interfaces makes sure that we can separate our communication from our implementation - meaning that whoever is using the abstract interface in their code never needs to even include the header files that define implementation details.

Best practices

Use the bridge pattern when it really makes sense to make a large scale separation of two hierarchies. The shapes example above that has both a visual draw-able part and a physics part is a good example for when this really makes sense. However, as you can see, it adds more code to your implementation so avoid using this pattern when it doesn't pay off in simplified maintenance.

- **Identify component hierarchies that need separation**: these should be largely independent hierarchies where the bridge pattern can truly simplify the interactions with objects on one side of the bridge.
- **Define clear interfaces**: since every interaction will pass through the abstract interfaces, it is important to have a well defined set of interfaces that will be implemented on the other side.
- **Thorough testing**: encode all requirements into integration and system tests and make sure that you verify that the whole chain of command works as expected from one side of the bridge to the other.

Pitfalls

I have shown you a C implementation in this module that maps directly onto an implementation in Rust. This C implementation is very powerful. If you try to change it in any way, you will easily confuse yourself. For example, a simple change such as changing the "pointer-to-pointer" to a simple pointer will make it impossible for you to use CONTAINER_OF on a handle.

The C implementation has been used, tested and fine tuned over many years. What I'm trying to do here is make you completely independent upon what language you use. You should easily be able to implement a pattern in any language. Patterns are like aggregations of meaning. In a spoken language it works the same way: you still have the same meaning - it's just that in different language you express it differently.

If you try to simplify the C implementation you will also make it less optimal because in that case you will instead likely remove the type safety that is inherent in this particular structure of C code. You are of course more than welcome to fiddle with it and try improving it, but I think it's pretty solid at the moment and the only next step I would personally take is simply switch to using Rust instead of C.

Additional pitfalls include:

- **Over-engineering**: the bridge pattern should be used sparingly and only where it is truly necessary to achieve complete separation of concerns, such as in the client server architecture scenario.
- **Complex interfaces**: the interfaces should be designed with the idea of providing "means, not policy". The interface should be simple and generic and provide means of communication with implementations on the other side of the bridge without enforcing too many rules.

Alternatives

- **Adapter Pattern**: This pattern is a lot more lightweight and it's primary use is to alter the interface of a class to suit user's needs. It's not a complete bridge thought and so lacks the ability to fully separate two hierarchies of objects. It can be considered as an alternative to the Bridge pattern when the conversion of data is involved.
- **Strategy Pattern**: This pattern focuses on breaking out behaviors of objects and making them interchangeable. It is useful when you need to only have interchangeable behaviors where an object can be configured to use one behavior strategy or another. It can be considered as an alternative to the Bridge pattern when the implementation of an algorithm is the primary concern.

Remember that the primary use of a bridge is to bridge together two pieces of land. The land in your code is represented by hierarchies of objects. If this is not what you are trying to do then it may be a good indication to consider alternative design patterns instead.

Conclusion

In this module we have covered the bridge design pattern and it's uses. It should be clear to you by now what defines the bridge pattern, when you should use it and when you may need to consider alternatives.

I have also presented you with two different implementations that compile and run in simulation using QEMU simulator and can be built and run using Swedish Embedded Platform SDK which you should already be very familiar with if you have followed the infrastructure series.

Quiz

- What is the main defining characteristic of the bridge pattern?
    1. That it defines a bridge between two hierarchies of objects both of which can evolve independently.
    2. That it bridges two APIs together
    3. That it defines a protocol of communication between two complex objects.
- How does the bridge pattern decouple abstractions and implementations?
    1. It is not used for decoupling.
    2. It uses a client side interface with which the application interacts and then another server side object interface which is connected to the server side object over the bridge. Virtual API pattern is used for the interfaces.
    3. It uses a network connection to decouple all abstractions abstractions.
- What is the role of abstraction in the bridge pattern?
    1. It is used for simplifying the design pattern.
    2. It is used to keep the interfaces simple
    3. It is used to separate the two hierarchies of implementations as much as possible.
- What is the difference between bridge and abstract interface?
    1. The bridge pattern has the purpose of separating hierarchies while abstract interface is a way of separating interface from implementation.

2. Abstract interface pattern builds on the bridge pattern.
3. There is no difference.

- How can you use the bridge pattern to simplify the software architecture of your current project?