Sidecar, Ambassador & Adapter Patterns

Tejas Parikh (t.parikh@northeastern.edu)

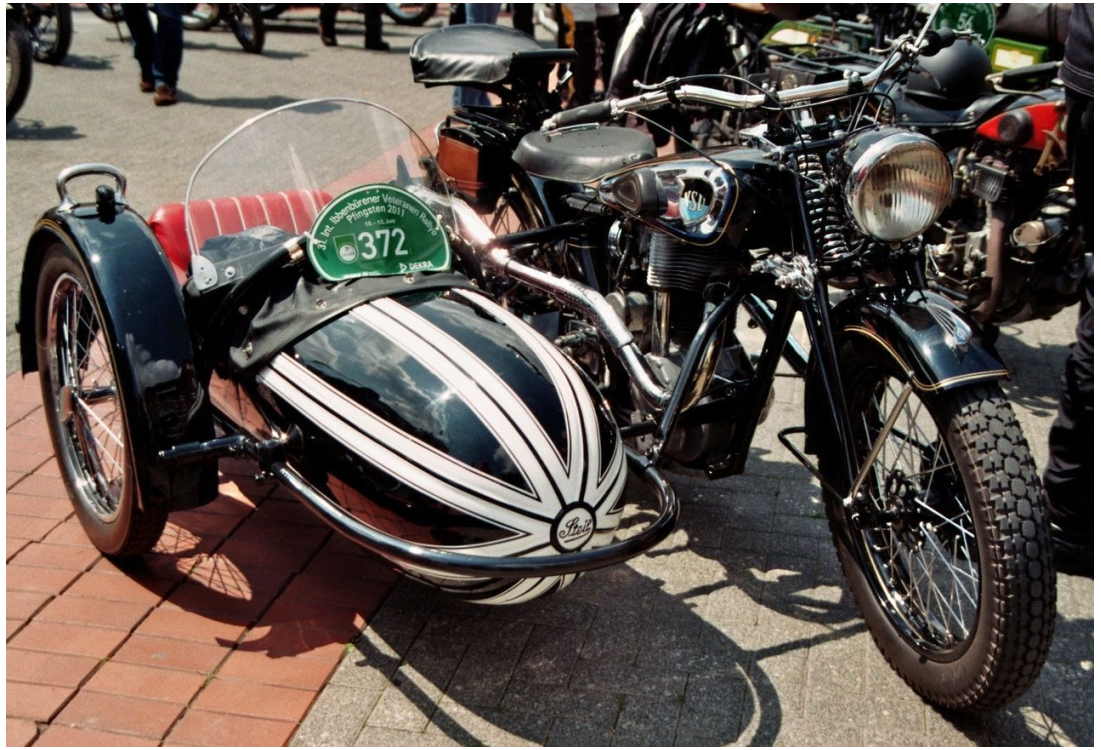
CSYE 7125

Advanced Cloud Computing
Northeastern University

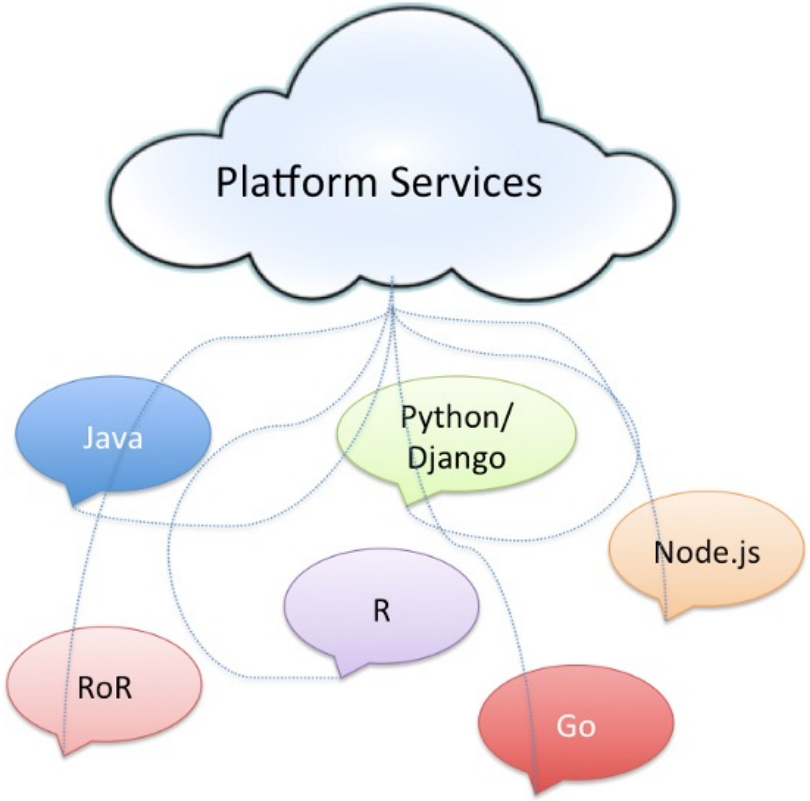
Sidecar

Sidecar

- A sidecar by definition is “A one-wheeled device attached to the side of a motorcycle, scooter, or bicycle, producing a three-wheeled vehicle.”.



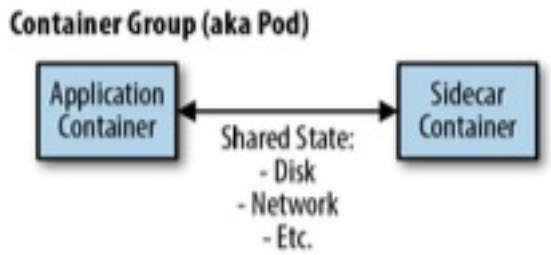
A Polyglot Cloud Ecosystem



Sidecar Pattern

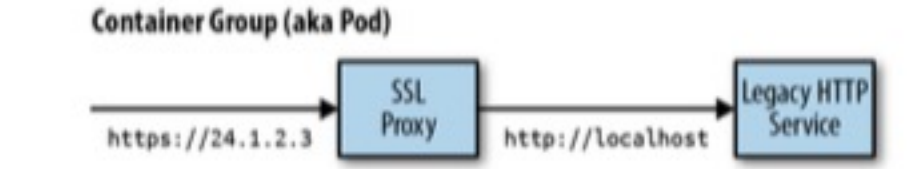
- The sidecar pattern is a single node pattern made up of two containers.
- The first is *application container*. It contains the core logic of the application.
- The second is *sidecar container*. The role of sidecar is to augment and improve the application container, often without the application container's knowledge.
- Application and Sidecar containers are deployed in same pod and share resources such as disk, network, etc.

Generic Sidecar Pattern



Example Use case

- Add HTTPS to Legacy Service

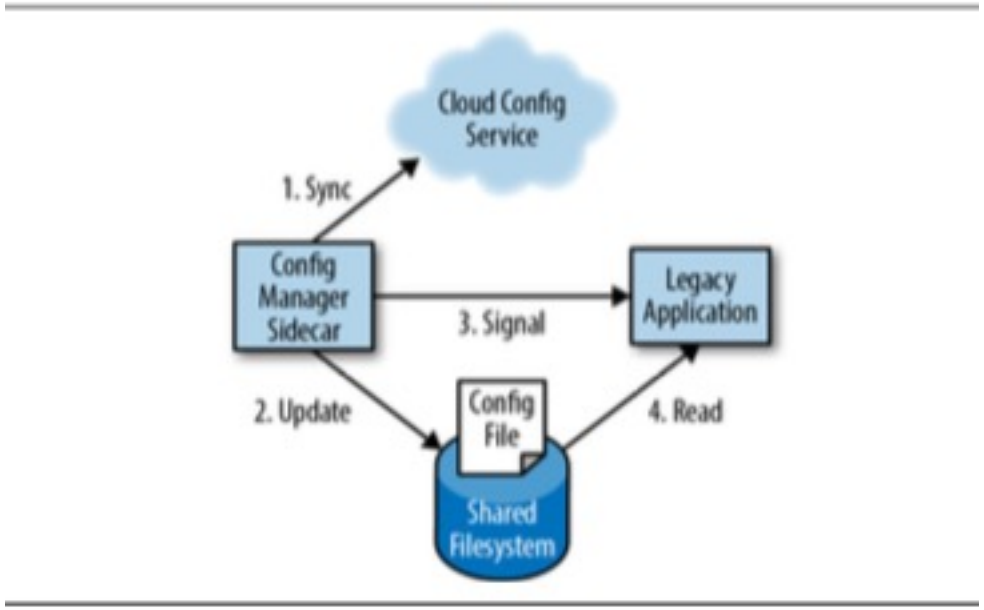


The HTTPS sidecar

Dynamic Configuration with Sidecars

- Many applications use a configuration file for parameterizing the application; this might be a raw text file or something more structured like XML, JSON, or YAML.
- Older legacy systems were written to assume that file was always there and many were designed to watch for changes to the config file and reconfigure itself if something changes.
- Sidecar can monitor configuration changes for the application in some data store or cloud service and update the legacy applications configuration file.
- Sidecar can also signal legacy application to reconfigure itself. If reconfiguration is not an option, sidecar can terminate the legacy application. Once aborted, container orchestration system will restart the legacy application, at which point it will load its new configuration.

Dynamic Configuration with Sidecars



A sidecar example of managing a dynamic configuration

Sidecar is not just for legacy applications

- Sidecar pattern does not exist just to adapt legacy applications where you no longer wanted to make modifications to the original source code.
- Main advantage of sidecar is modularity and reuse of the components used as sidecar.

Example Sidecar for Monitoring

- All applications must provide an HTTP /topz interface that provides a readout of resource usage.
- Applications might be implemented in various languages.
- Rather than implement the interface in every language used and tightly coupling with application, it can be implemented in sidecar application.
- Sidecar application running in same shared PID namespace can now provide complete view of resource usage.
- Sidecar container can be added to applications deployed via orchestration system to ensure that there is consistent set of tools available for all applications running in your infrastructure.

Designing Sidecar for Modularity and Reusability

- Sidecar must be a modular, reusable artifact.
- Sidecar must be reusable across a wide variety of applications and deployments.
- Reusable sidecar can dramatically speed up the building of your application.
- For a successful, modular and reusable sidecar you must:
  - Parameterize your containers
  - Create API surface of your container
  - Document the operation of your container

Parameterized Containers

- Just like application containers, sidecar containers should also provide flexibility in configuration thru parameters.
- Hardcoded values on sidecar containers make them less flexible and reduce reusability.
- Think of parameters for functions in your program.

Parameterized Container - Example

- Earlier we discussed sidecar to add HTTPS support to legacy applications using sidecar.
- Where is the SSL certificate stored? In the sidecar?
- Provide path to SSL cert in parameter and legacy application HTTP endpoint in parameter to sidecar container.

Container's API

- In the world of microservices, sidecar containers rely on APIs to ensure that there is clean separation between the main application container and the sidecar.
- APIs exist to ensure that all consumers of sidecar will continue to work correctly as subsequent version are released.

Documenting Your Containers

- Parameterization made containers reusable and modular, API provided stability to sidecar consumers but to make it all work you must have documentation so people can use the sidecars.

Advantages of using a sidecar pattern

- A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language.
- The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application.
- Because of its proximity to the primary application, there’s no significant latency when communicating between them.
- Even for applications that don’t provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as own process in the same host or sub-container as the primary application.

When to use sidecar pattern?

- Your primary application uses a heterogenous set of languages and frameworks. A component located in a sidecar service can be consumed by applications written in different languages using different frameworks.
- A component is owned by a remote team or a different organization.
- A component or feature must be co-located on the same host as the application
- You need a service that shares the overall lifecycle of your main application, but can be independently updated.
- You need fine-grained control over resource limits for a particular resource or component. For example, you may want to restrict the amount of memory a specific component uses. You can deploy the component as a sidecar and manage memory usage independently of the main application.

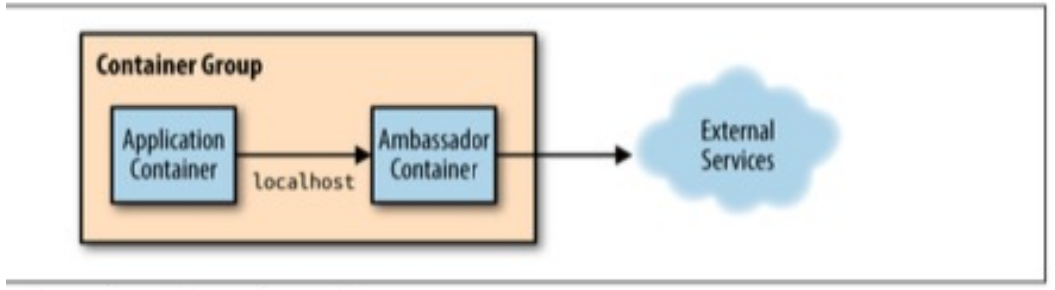
When sidecar pattern might not be good fit?

- When inter-process communication needs to be optimized. Communication between a parent application and sidecar services includes some overhead, notably latency in the calls. This may not be an acceptable trade-off for chatty interfaces.
- For small applications where the resource cost of deploying a sidecar service for each instance is not worth the advantage of isolation.
- When the service needs to scale differently than or independently from the main applications. If so, it may be better to deploy the feature as a separate service.

Ambassadors

Ambassador Pattern

- An ambassador container brokers interactions between the application containers and rest of the world.
- Application container and Ambassador container are tightly linked in the sense that they are scheduled to a single machine.



. Generic ambassador pattern

Ambassador Pattern

- An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.
- Ambassador pattern has inherent value in building modular and reusable containers.
- Ambassador container can be reused with a number of different application containers.
- Ambassadors provide consistent implementation because it is built once and used in many different contexts.

Ambassador Pattern

- This pattern can be useful for offloading common client connectivity tasks such as monitoring, logging, routing, security (such as TLS), and resiliency patterns in a language agnostic way.
- It is often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities.
- It can also enable a specialized team to implement those features.

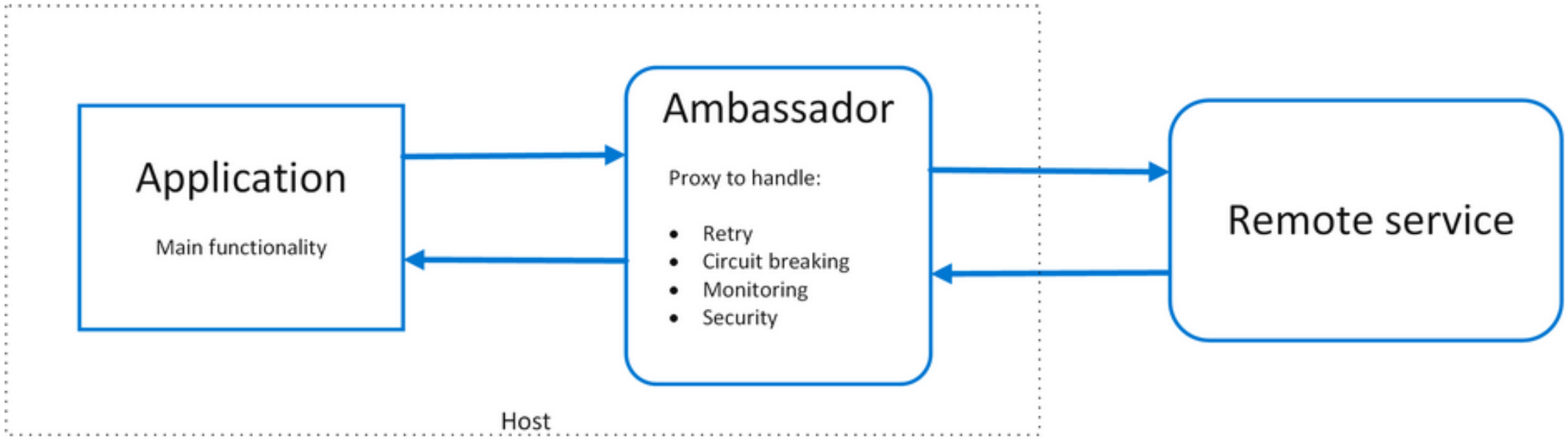
Ambassador Pattern Use case

- Resilient cloud-based applications require features such as circuit breaking, routing, metering, and monitoring, and the ability to make network-related configuration updates. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.
- Network calls may also require substantial configuration for connection, authentication, and authorization. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

Ambassador Pattern

- Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions. You can also use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.
- Features that are offloaded to the ambassador can be managed independently of the application. You can update and modify the ambassador without disturbing the application's legacy functionality. It also allows for separate, specialized teams to implement and maintain security, networking, or authentication features that have been moved to the ambassador.
- Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or service. Alternatively, if an ambassador is shared by multiple separate processes on a common host, it can be deployed as a daemon or Windows service. If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate link configured for communication.

Ambassador Pattern



Issues and Considerations

- The proxy adds some latency overhead. Consider whether a client library, invoked directly by the application, is a better approach.
- Consider the possible impact of including generalized features in the proxy. For example, the ambassador could handle retries, but that might not be safe unless all operations are idempotent.
- Consider a mechanism to allow the client to pass some context to the proxy, as well as back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
- Consider how you will package and deploy the proxy.
- Consider whether to use a single shared instance for all clients or an instance for each client.

Use Ambassador Pattern When You

- Need to build a common set of client connectivity features for multiple languages or frameworks.
- Need to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
- Need to support cloud or cluster connectivity requirements in a legacy application or an application that is difficult to modify.

Don't use Ambassador Pattern

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases, this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.

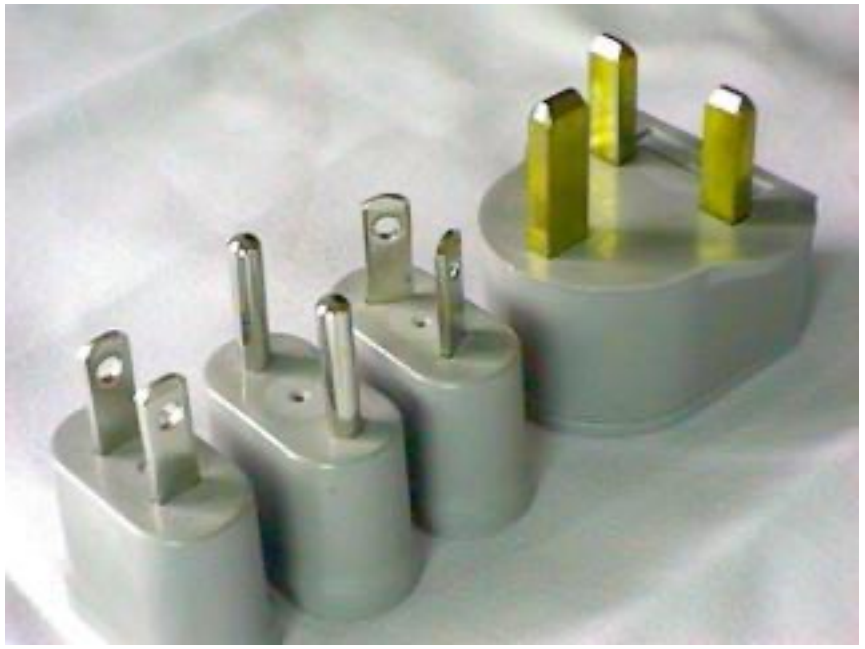
Adapters

So far we have learned that

- Sidecar patter can extend and augment existing application containers.
- Ambassadors can alter and broker how an application container communicates with the external world.

Adapters in real-world

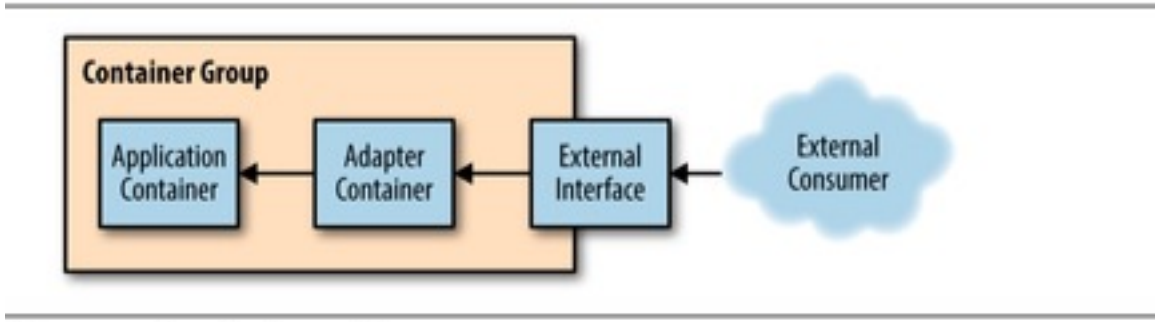
- The general idea of an adapter in software development is identical to the one in the physical world.
- If you have been to different countries, you probably recognized that a lot of them are using differently shaped power sockets. Quite often, they are shaped in a way that the plug of your electrical device doesn't fit. So, how do you connect the charger of your mobile phone or laptop to these power sockets?
- The answer is simple. You get an adapter which you can put into the power socket and then you put your plug into the other end of the adapter. The adapter changes the form of your plug so that you can use it with the power socket.
- **The adapter doesn't provide any additional functionality.** It just enables you to connect your plug to the power socket.



Adapters

- In Adapter pattern, the adapter container is used to modify the interface of the application container so that it conforms to some predefined interface that is expected of the application container so that it conforms to some predefined interface that is expected of all applications.

Adapter Pattern



The generic adapter pattern

Why do we need Adapters?

- Real-world application development is heterogeneous, hybrid exercise.
- Parts are written by your team, some parts are supplied by third party and others might be off-the-shelf opensource software.
- The net effect of this heterogeneity is that real-world application you deploy will have been written in a variety of languages, and with a variety of conventions for logging, monitoring, and other common services.

When should you use Adapters?

- When you want to use an existing class, and its interface does not match the one you need
- When you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
- When you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.
- When most of the applications using third party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library. If another library has to be used only an adapter for the new library is required without having to change the application code.

Adapter Pattern Example

- Different systems/applications/programming languages log data format varies widely. You want to ensure that despite different structures of the log message, every log message ends up with appropriate timestamp.
- Adapter pattern can help by transforming log data from different application into a single structured representation that can be consumed by the log aggregator.
- Adapter is taking a heterogenous world of applications and creating a homogenous world of common interfaces.

Additional Resources

See Lecture Page