Hexagonal Architectures

Ports and Adapters

Agenda

- Architectural Styles
  - Hierarchical
    - Layered
      - Hexagonal Architectures
- Implementing Hexagonal Architectures
  - From Application Services to Commands
  - Command Dispatcher
- Quality of Service
  - Command Processor

From Mud to Structure

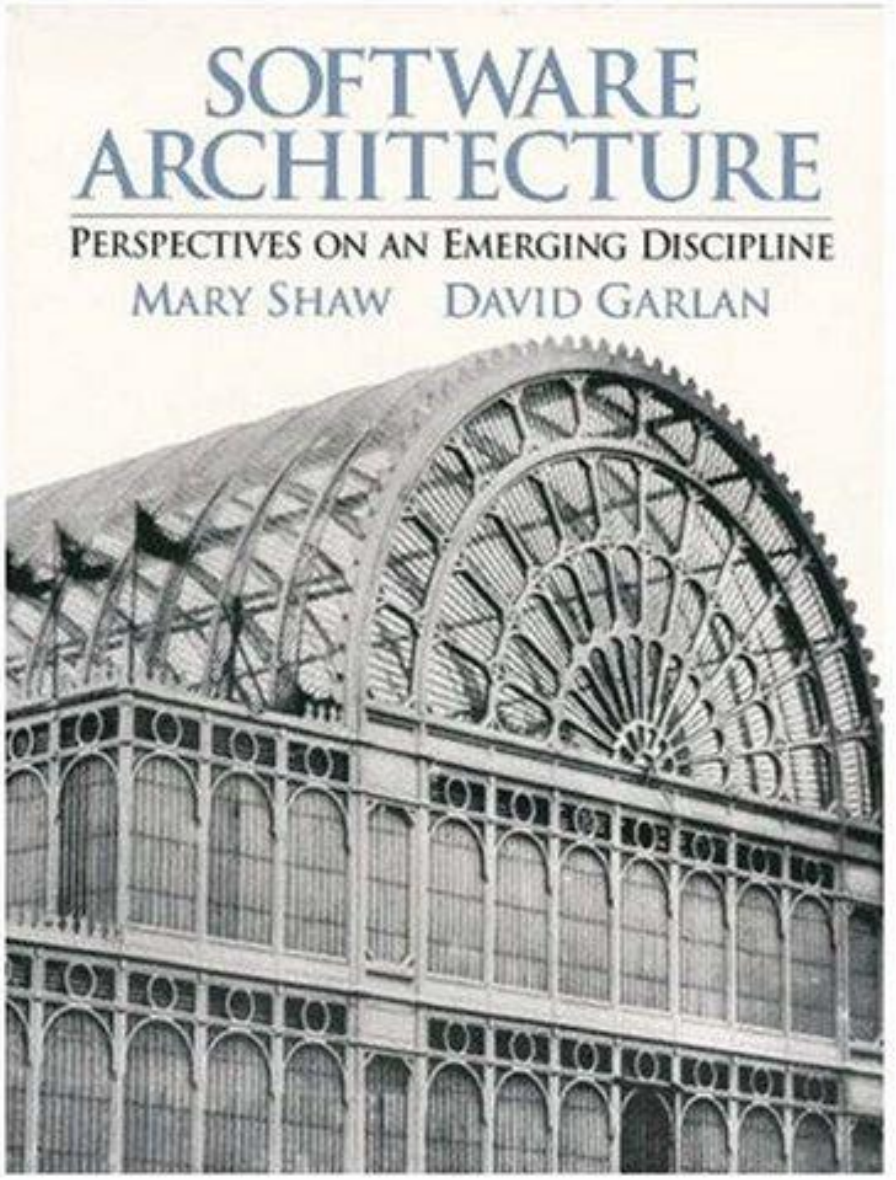
ARCHITECTURAL STYLES

Garlan and Shaw

Working at Carnegie-Mellon in the ‘90s Shaw and Garlan begin identifying *architectural styles*.

Shaw and Garlan propose that the journey from craft to engineering is the creation of a body of knowledge that allows repeatable large scale production. Science provides the insights here that drive engineering.

Similar goal to the Design Patterns community – but at a different scale, capture knowledge on patterns of component interaction.



What is an architectural style?

A *component* is a basic architectural computational component – clients, servers, filters, layers, a database etc.

A *connector* provides the interaction between components.

A style is a *vocabulary* of possible components and connector types, and *constraints* on how they can be applied.

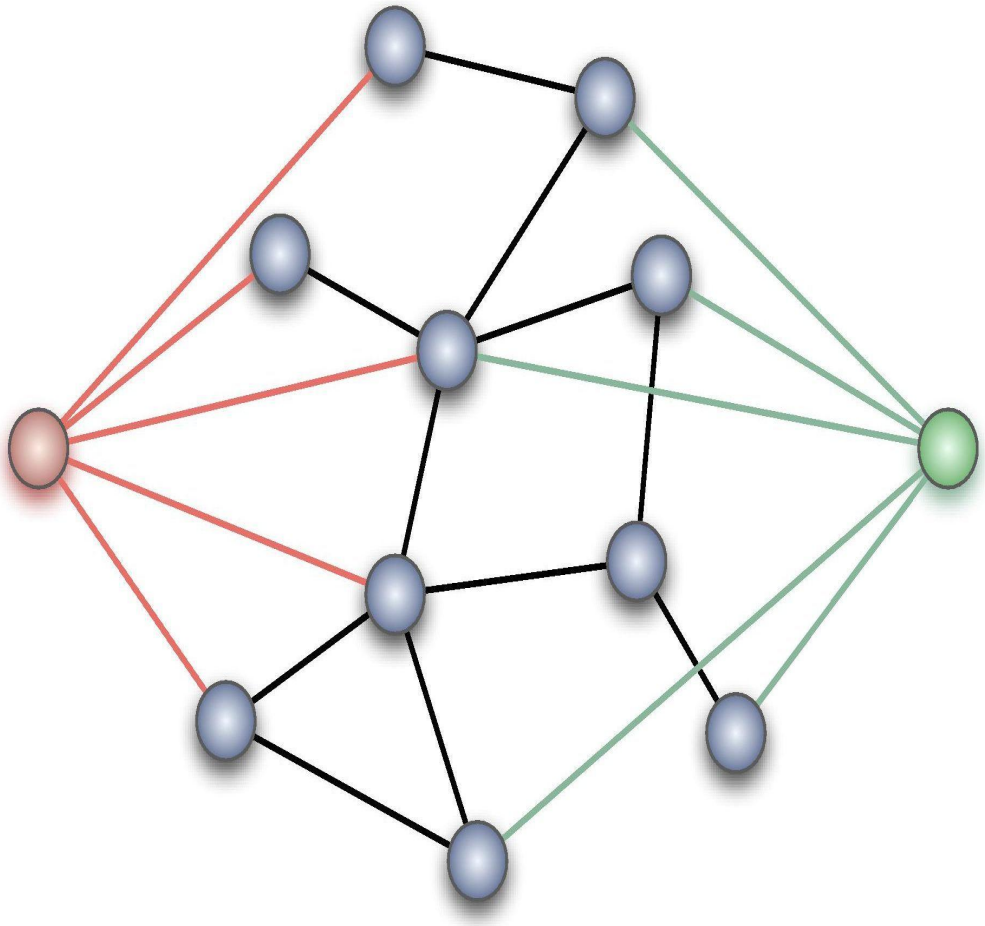
Given the above we can classify architectures:

What is the design vocabulary?

What are the allowed structural patterns?

What is the underlying computational model?

What are the essential invariants?



Boxology

Missing from this is obviously any discussion of User Interface styles. The question is whether MVC, for example, is simply a design pattern that fits into another one of these styles or has unique characteristics

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and Filters Batch Sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext Systems Compound Document
Call & Return Systems	Dominated by order of computation, usually with a single thread of control	Procedural Object Oriented Naïve Client Server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven (Implicit , Reactive, or Decoupled Invocation) Event Sourcing Broker SOA
Hierarchical Systems	Dominate by reduced coupling with the partition of the system into subsystems with limited interaction	Interpreters Rule-Based Systems Layers CQRS

Heterogeneous styles

Components

- Whilst we talk about styles in a pure form for understanding, most applications employ a combination of styles
  - The most common approach is hierarchical – a component identified in one style is itself implemented using another style
  - A single component might use different architectural connectors. A component might access a repository, but interact with other components through pipes and filters or implicit invocation
  - Elaborate a whole level of the architecture in a different style (as opposed to just a component)

Connectors

- A connector may be hierarchically decomposed as well
  - For example the pipe in a pipes and filters architecture might be implemented using messaging middleware, which is itself uses a heterogeneous architectural style such as repository and implicit invocation

Layered Systems

Drivers

The major problem in software development is the management of complexity. Separation of concerns - the decomposition of the program into small components enables breaking a large complex system into simpler smaller ones.

Complex components may in turn require further decomposition.

Interfaces should be stable – they may even be an external contract for the system.

Changes to implementation details should not ripple through the system. Components should be replaceable with alternative implementations without affecting the rest of the system.

Similar responsibilities should have high cohesion – be grouped together – to aid reasoning about the software.

The dominant characteristic is a mix of higher and lower order operations, where high level operations depend on lower-level ones

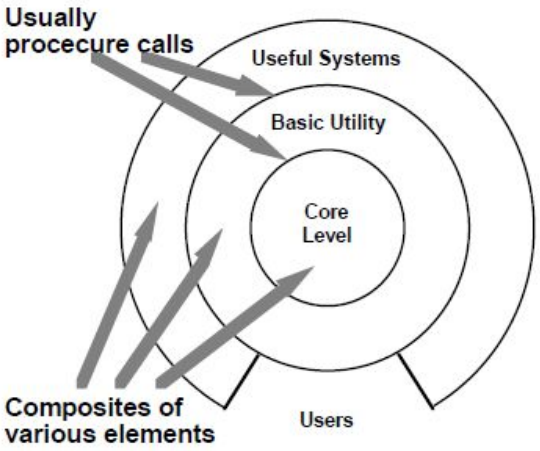


Figure 3: Layered Systems

Architectural Styles, Garlan and Shaw

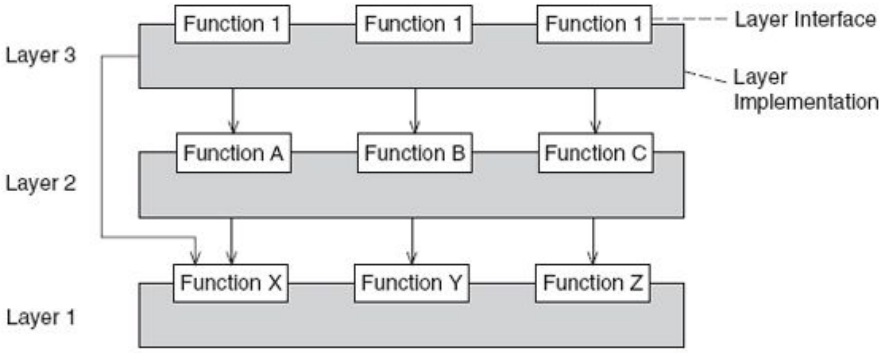
Layered Architecture

Components are the layers – a *group of subtasks* which implement an abstraction at some layer in the hierarchy.

Each layer *provides services* to the layer above it and *acts as a client* to the layer below.

The connectors are the *protocols* that define how the layers can interact.

Lower layers can pass data and service requests to higher layers via notifications (i.e. an observable)



POSA, 4th Volume

Invariants

Layers can only interact with adjacent layers.

A layer can provide services to a layer above, and consume services provided by the layer below.

A layer depends only on lower layers and has no knowledge of the higher layers.

The structure can be compared with a stack or onion.

Properties

Designs are based on increasing levels of abstraction. This allows portioning of a complex problem into a series of steps.

Changes to one layer impact at most two layers – below and above, supporting change.

Different implementations of the same layer can be reused interchangeably provided they support the same interface

Implementation

.

- Define the abstraction criteria for grouping tasks into layers
  - The criteria is often the distance from the client with higher layers providing services to a user and lower levels dealing with the hardware
- Determine the number of levels according to that criteria. Name those layers and assign tasks to them
  - Trade off between complexity of number of layers and the flexibility it provides
  - Again tasks that relate to the overall system as perceived by the client are in the highest layer, tasks that support that layer are in the next layer below and so on.
- Specify an interface for each layer
  - We want to encapsulate knowledge about the layer from callers, because it supports modifiability. So we define an interface component to the layer, allowing 'black-box' use of that layer. This component provides a façade to the services of that layer. The black box may be 'grey' if we expose more than one façade component because the tasks provided by the service are orthogonal toe each other.
- Specify the approach for communication between adjacent layers
  - Most commonly higher layers 'push' the data that lower levels need as part of the parameters to the call to that layer
  - However, sometimes a higher layer allows a lower layer to pull the information when desired, only providing a notification to the lower layer to pull on its own discretion.
  - If the higher layer provides notifications then the lower layer will need to use callbacks to call the higher layer, registering a subscriber with the higher layer to be notified when a change occurs.
- Design an error-handling strategy
  - Either handle all errors within the layer or transform and pass to the higher layer

Examples

- Ports and Adapters (see below)

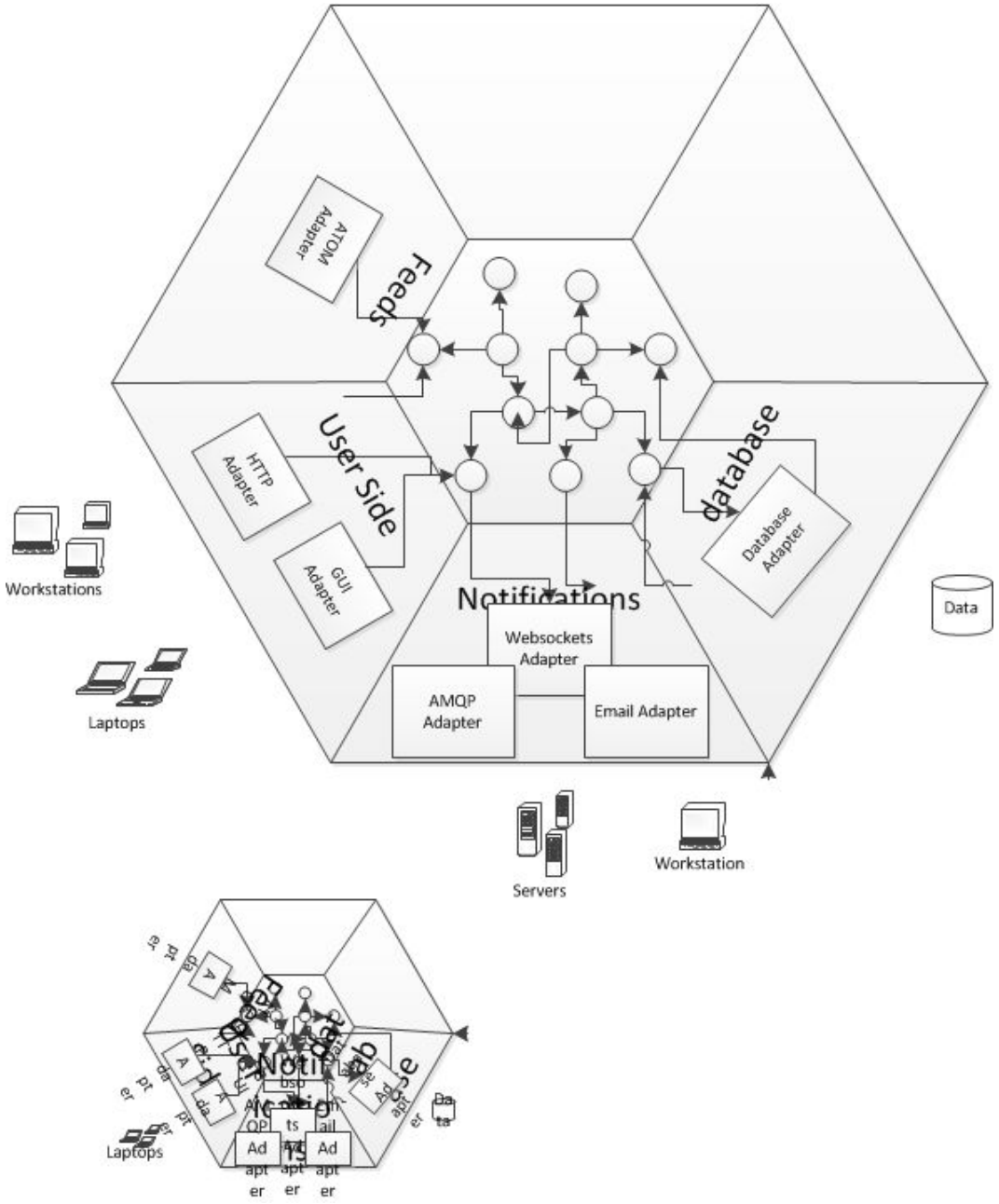
Disadvantages

Not all systems are easily structured as layers

Performance considerations may require violation of layers and a closer coupling between higher and lower layers.

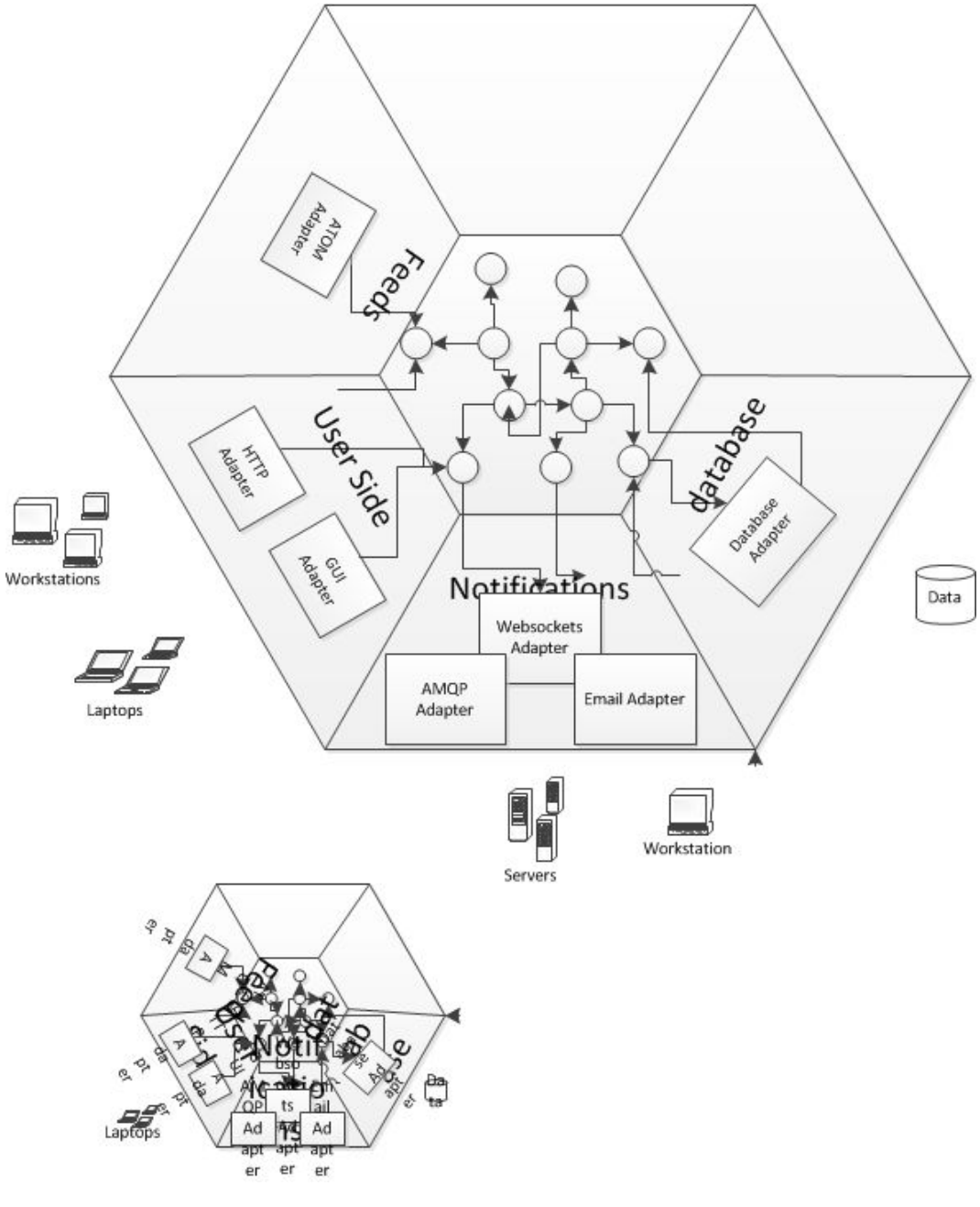
Ports and Adapters

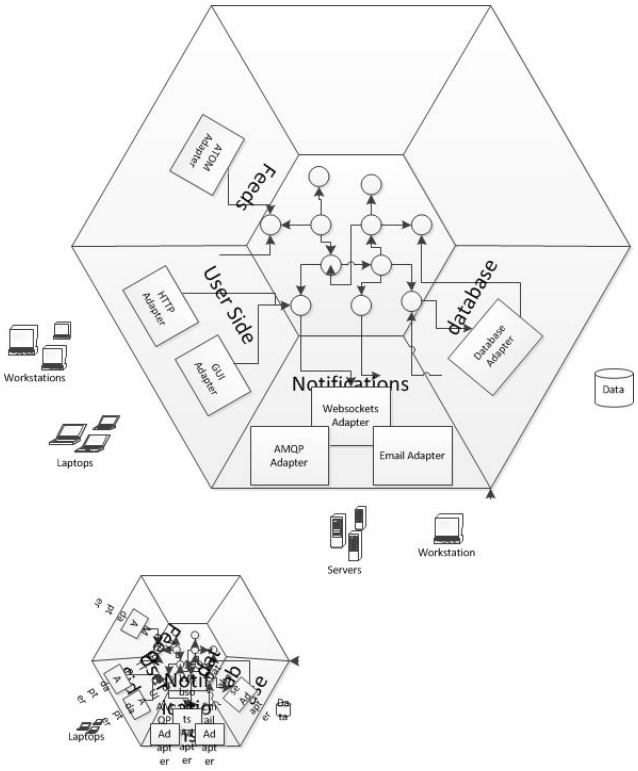
A **ports & adapters** architectural style (Hexagonal Architecture) is a variation of the layered architectural style which makes clear the separation between the **domain model** - which contains the rules of our application - and the **adapters**, which abstract the inputs to the system and our outputs. The advantage of this style is that the application is decoupled from the nature of the input or output device, and any frameworks used to implement them.



Ports and Adapters

For example when a client POSTs a request to the REST API exposed by our application the adapter receives the HTTP request, transforms it into a call onto our domain, and marshals the response back out to the client over HTTP. Similarly if our application needs to retrieve persisted entity state to initialise the domain it calls out to an adapter that wraps access to the DB.

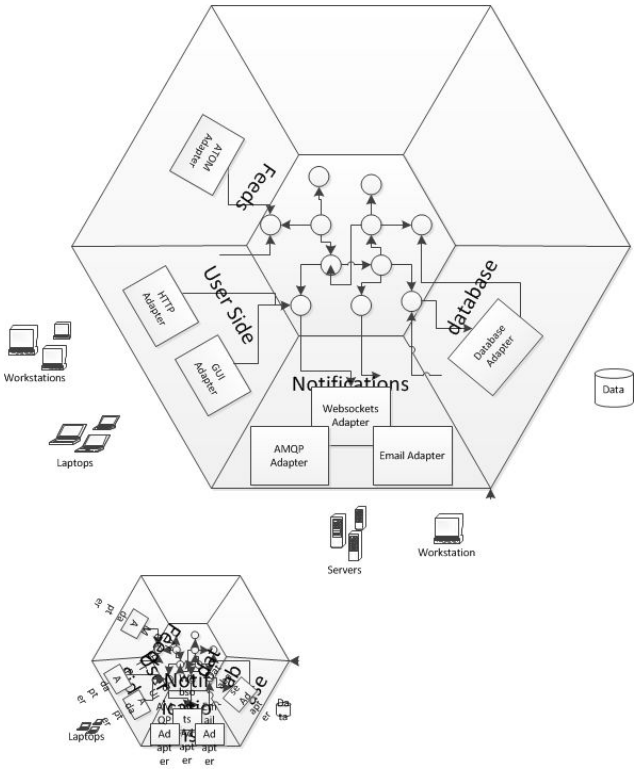




The layer between the adapter and the domain is identified as the **ports** layer. Our *domain is inside the port*, *adapters for external entities are on the outside of the port*.

The notion of a 'port' invokes the OS idea that any device that adheres to a know protocol can be plugged into a port. Similarly, many adapters may use our ports.

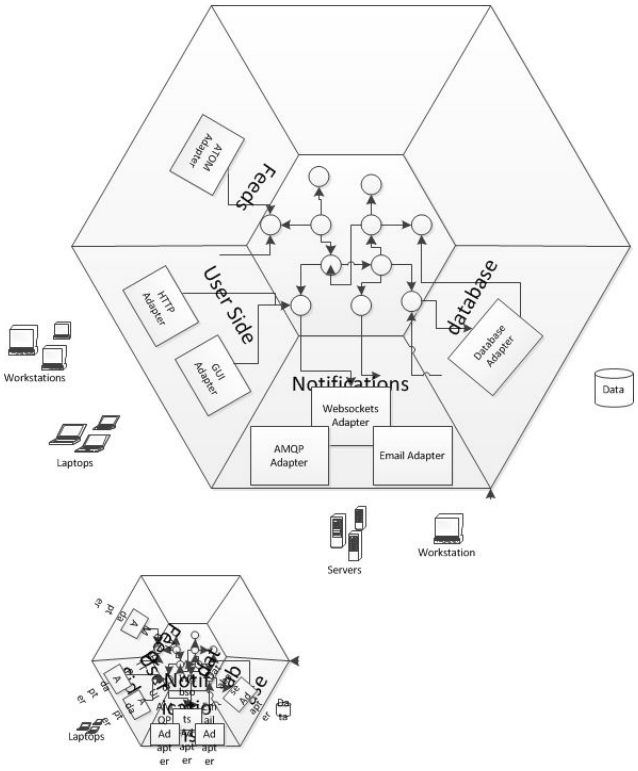
Ports and Adapters



A port is the *'use case boundary*'. Use cases become problematic when they become focused on technology concerns. Use cases written against the ports can elide those concerns and focus on the application rules, making them easier to write and maintain.

In BDD terms a Use Case is often expressed as Given-When-Then acceptance criteria for a story, but the principle is the same: the boundary for the GWT scenario is the ports layer.

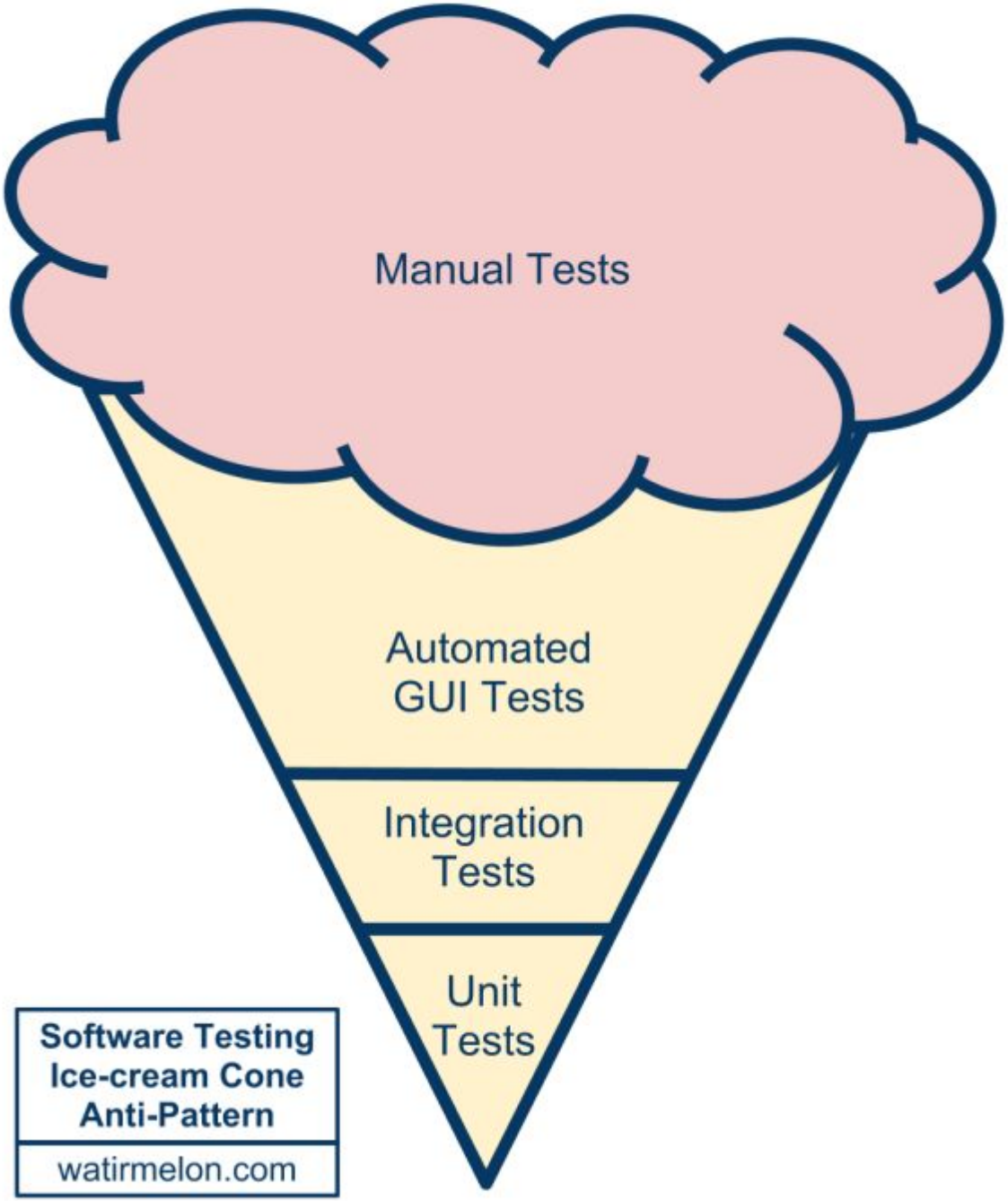
Ports and Adapters



There is a correlation here between the use case boundary and the test boundary - tests should focus on the behaviour expressed by a use case, not on a unit of code.

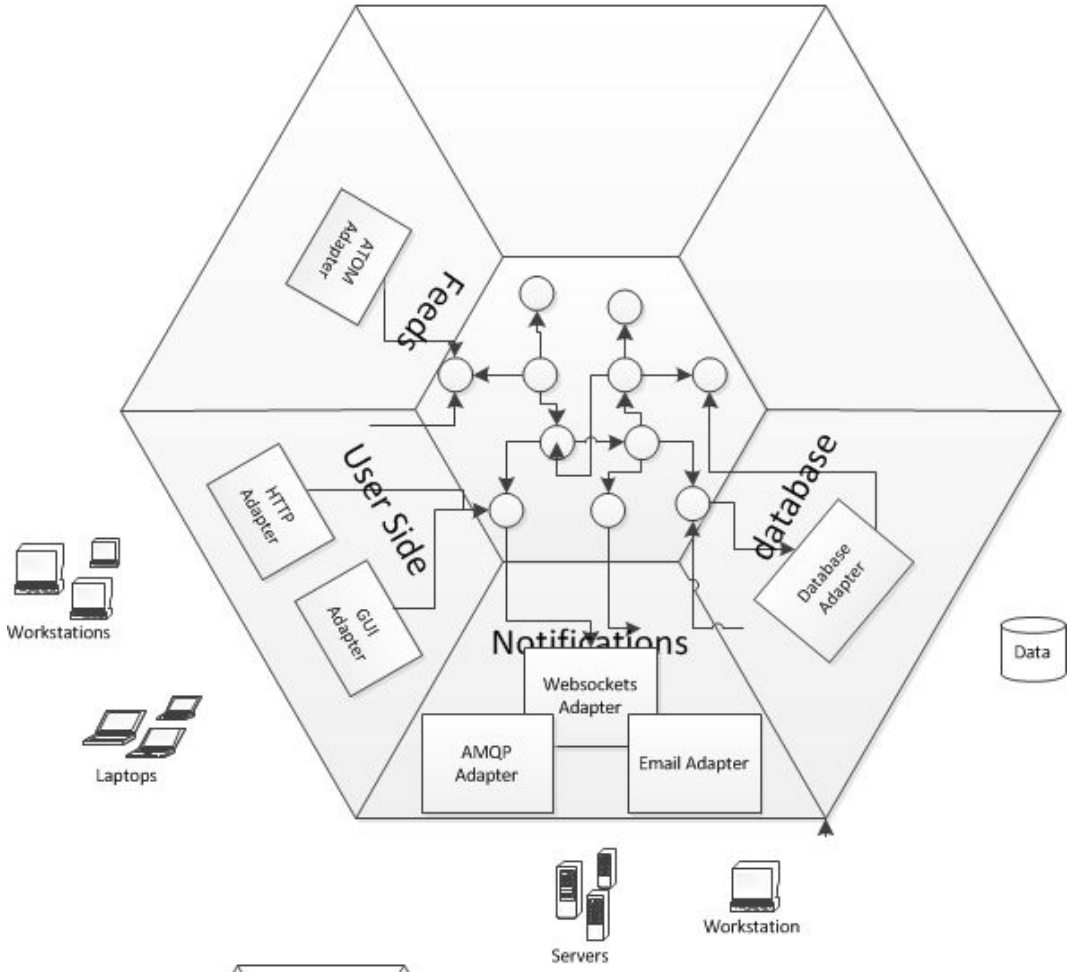
Contrary to Cockburn, I don't suggest using a ATT like FIT or SpecFlow here, preferring to use our xUnit test tool here (and not test implementation details).

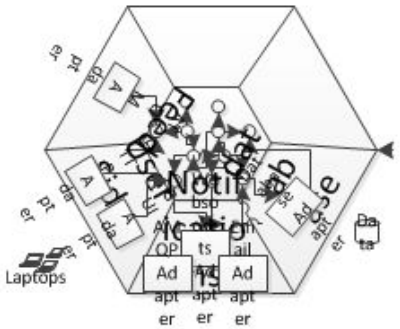
Ports and Adapters

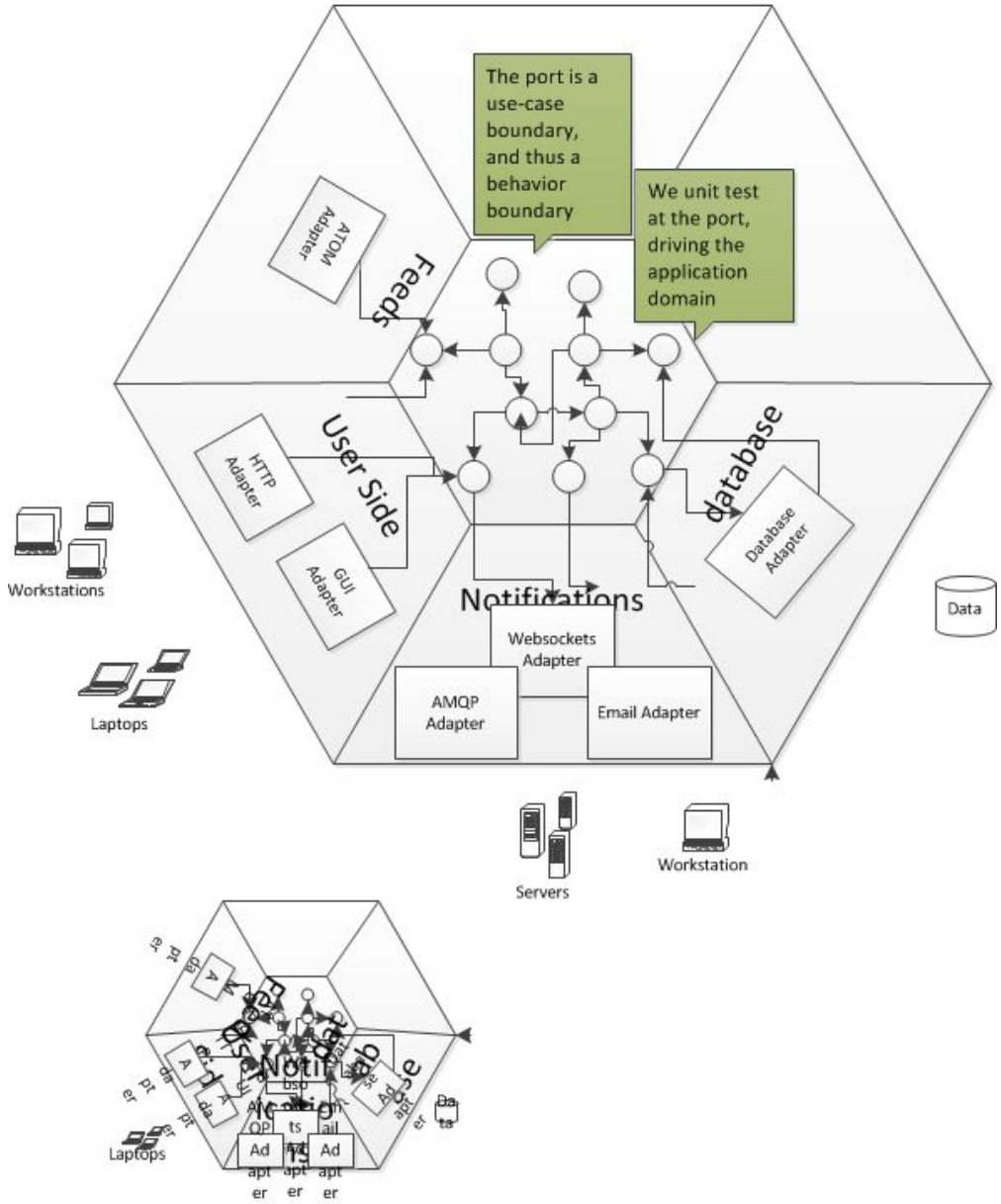

Manual Tests
Automated
GUI Tests
Integration
Tests
Unit
Tests
Software Testing
Ice-cream Cone
Anti-Pattern
watirmelon.com

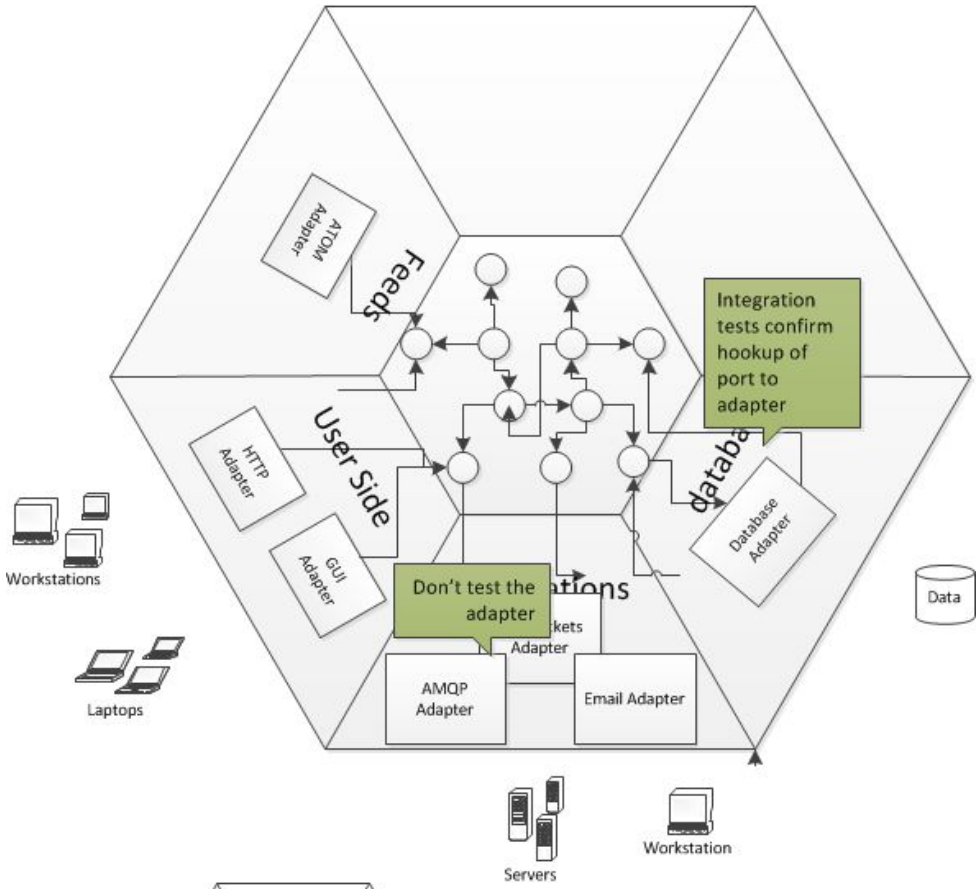
- UI
- Integration
- Unit Tests

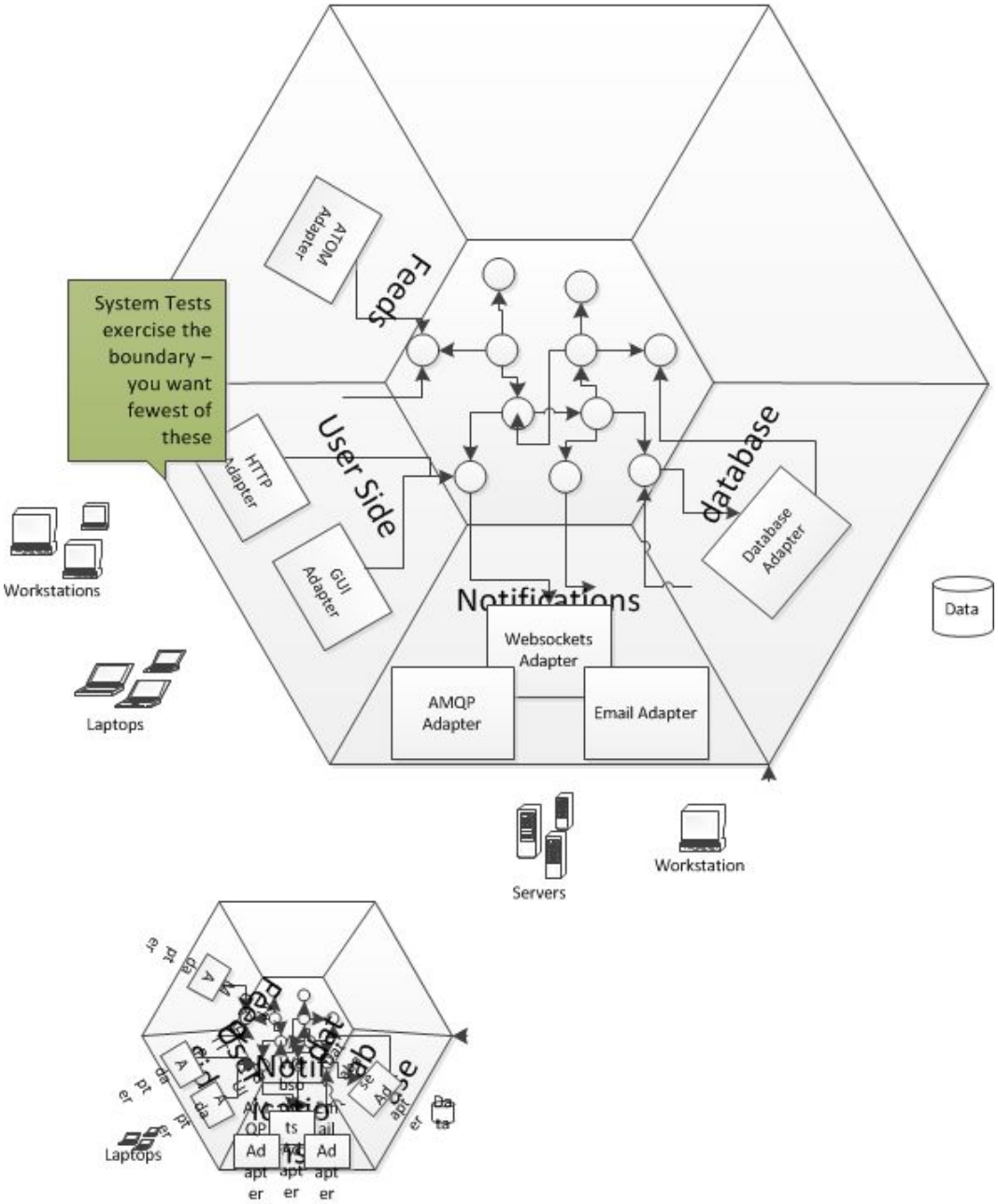
After Mike Cohn, Martin Fowler et al.

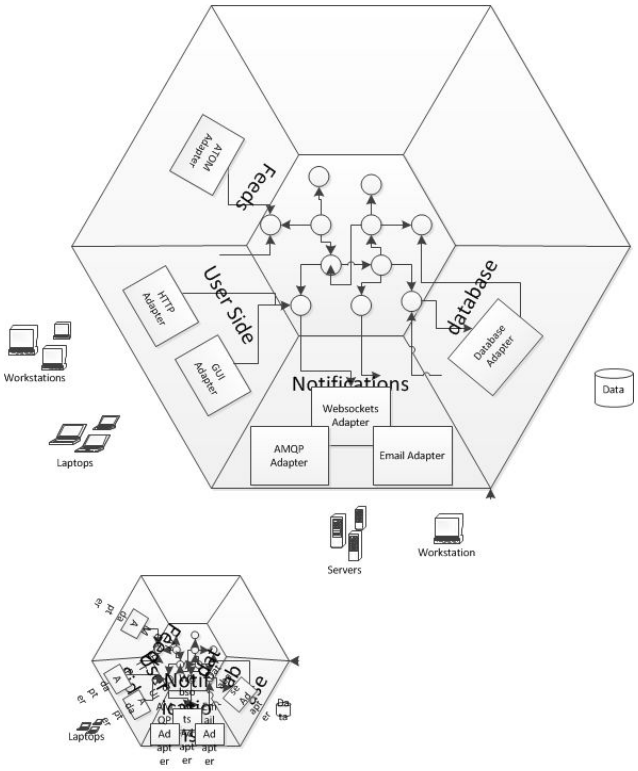

ATOM Adapter
Feeds
HTTP Adapter
GUI Adapter
User Side
database
Database Adapter
Data
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Servers
Workstation




The port is a use-case boundary, and thus a behavior boundary
We unit test at the port, driving the application domain
ATOM Adapter
Feeds
HTTP Adapter
User Side
GUI Adapter
database
Database Adapter
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Data
Servers
Workstation


Feeds
ATOM Adapter
User Side
HTTP Adapter
GUI Adapter
Integration tests confirm hookup of port to adapter
Database Adapter
Data
Don't test the adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Servers
Workstation


System Tests exercise the boundary – you want fewest of these
ATOM Adapter
Feeds
HTTP Adapter
User Side
GUI Adapter
database
Database Adapter
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Data
Workstations
Laptops
Servers
Workstation



We have a notion *of primary and secondary actors* in use cases which map to the adapter and port layer.

Primary actors exercise our application, they are inputs into our application. *A primary actor uses a primary adapter, which calls a primary port - the chain is one of inputs into our application*. So our REST API is a primary adapter, so are our tests.

*A **secondary actor** is one that our application exercises as part of its work, they are outputs from out application*. So our DB is a secondary adapter, as our mocks and we talk to them over a secondary port.

Many applications seem to consist only of one primary and one secondary, which leads to the n-tier style, but once we factor in tests we may begin to observe that we have more, and by building for multiple ports we make our application more modifiable to new ports in future.

We often show *primary ports on the left* and *secondary ports on the right*.

Ports and Adapters

Domain and adapter are obvious, but what are ports?

IMPLEMENTING PORTS

Implementing Ports

One question is, how is our API provided? We are inside an adapter, so we want to use a Plain Old C# Object (POCO) approach to avoid framework dependencies. A common route is to use the *Facade* pattern, providing a class that holds the API to lower level components. Indeed this is the classical implementation strategy for the application service layer in an layered architecture. The Facade design pattern provides a high-level interface to a sub-system, shielding the caller from the implementation details of the sub-system. This both reduces complexity for the caller, and decouples the caller from the implementation allowing us to vary it without amending the caller. To this extent it meets the needs of our port, providing a clear notion of outside - the caller and inside - the subsystem.

Service Facade

A Facade used in this context is being used in the Service role, where the Service stereotype describes an object that provides co-ordination and control logic. It provides a *contract* to clients for lower layers.

```
public class MyFatService
{
   public void CreateMyThing(/* .. parameters ...*/)
   {
      /*Stuff*/
   }

   public void UpdateMyThingForFoo(/* .. parameters ...*/)
   {
     /*Other Stuff*/
   }

    public void UpdateMyThingForBar(/* .. parameters ...*/)
    {
      /*Other Stuff*/
    }

    /*Loads more of these*/
}
<code/>
```

Interface Segregation Principle

The Interface Segregation Principle states that clients should not be forced to depend on methods on an interface that they do not use. This is because we do not want to update the client because the interface changes to service other clients in a way that the client itself does not care about. An Transaction Script style port forces consumers (for example MVC controllers) to become dependent on methods on the domain service class that they do not consume.

Now this can be obviated by having the domain service implement a number of interfaces, and hand to its clients interfaces that only cover the concerns they have. With application service layers this naturally tends towards one method per interface.

```
public interface ICreateMyThingService
{
   void CreateMyThing(/* .. parameters ...*/);
}

public interface IUpdateMyThingForFooService
{
   void UpdateMyThingForFoo(/* .. parameters ...*/);
}

public interface IUpdateMyThingForBarService
{
  void UpdateMyThingForBar(/* .. parameters ...*/);
}

public class MyFatService : ICreateMyThingService, IUpdateMyThingForFooService,
IUpdateMyThingForBarService
{
  public void CreateMyThing(/* .. parameters ...*/)
 {
    /*Stuff*/
  }

  public void UpdateMyThingForFoo(/* .. parameters ...*/)
 {
    /*Other Stuff*/
  }

  public void UpdateMyThingForBar(/* .. parameters ...*/)
  {
    /*Other Stuff*/
   }

/*Loads more of these*/

}
```

Single Responsibility Principle

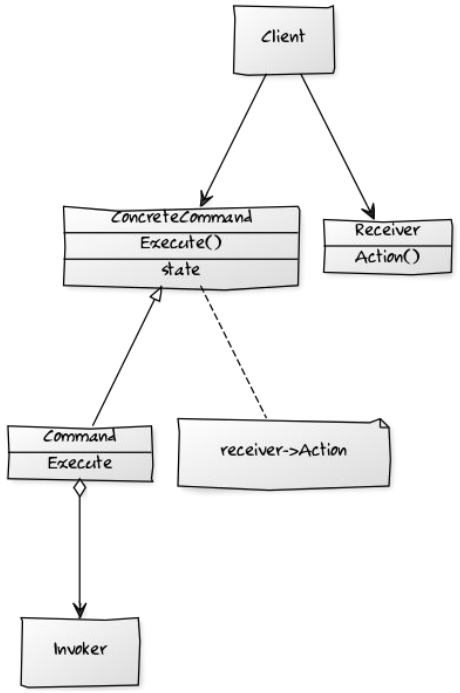
Now the Single Responsibility Principle suggests that a class should have one and only one reason to change. All these separate interfaces begin to suggest that a separate class might be better for each interface, to avoid updating a class for concerns that it does not have. So we tend to move toward one class per API endpoint.

```
public interface ICreateMyThingService
{
   void CreateMyThing(/* .. parameters ...*/);
}

public class CreateMyThingService : ICreateMyThingService
{
   public void CreateMyThing(/* .. parameters ...*/)
   {
     /*Stuff */
   }
}

public interface IUpdateMyThingForFooService
{
   void UpdateMyThingForFoo(/* .. parameters ...*/);
}

public class UpdateMyThingForFooService : IUpdateMyThingForBarService
{
   public void UpdateMyThingForBar(/* .. parameters ...*/)
  {
     /*Other Stuff*/
  }
}

public interface IUpdateMyThingForFooService
{
  void UpdateMyThingForBar(/* .. parameters ...*/);
}

public class UpdateMyThingForFooService : IUpdateMyThingForFooService
{
  public void UpdateMyThingForFoo(/* .. parameters ...*/)
 {
   /*Other Stuff*/
 }
}
```

Command Design Pattern

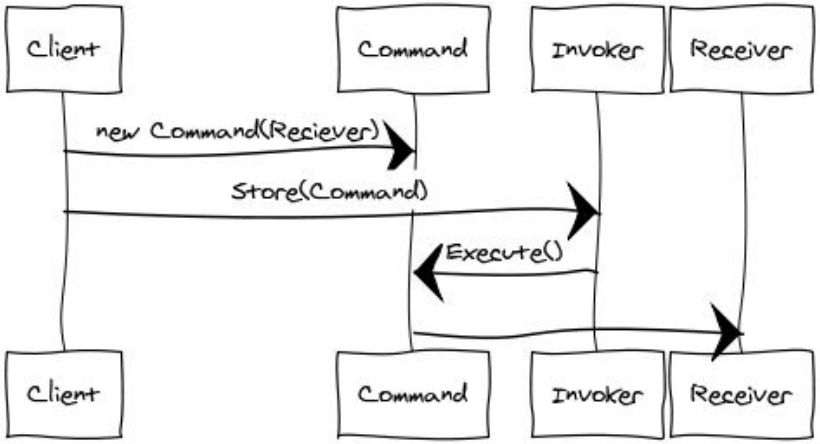
The command design pattern **encapsulates a request as an object**, allowing **reuse**, **queuing** or **logging** of requests, or **undo**able operations.

It also serves to decouple the implementation of the request from the requestor. The caller of a command object does not need to understand how the command is actioned, only that the command exists. When the caller and the implementer are decoupled it becomes easy to replace or refactor the implementation of the request, without impacting the caller - our system is more modifiable. Our ability to test the command in isolation of the caller - allows us to implement the ports and adapters model easily - we can instantiate the command, provide 'fake' parameters to it and confirm the results. We can also use the command from multiple callers, although this is not a differentiator from the service class approach.



Command - Declares an interface for executing an operation.
ConcreteCommand –Defines a binding between a **Receiver** object and an action. Implements Execute by invoking the corresponding operation(s) on Receiver.
Client – creates a ConcreteCommand object and sets its receiver.
Invoker - asks the command to carry out the request.

Command Sequence Diagram



An **Invoker** object knows about the concrete Command object.

The **Invoker** issues a request by calling *Execute* on the **Command**.

When commands are un-doable, the Command stores state for undoing the command prior to invoking Execute.

The Command object invokes operations on its **Receiver** to carry out the request

Command Design Pattern

In addition we can structure a system transaction ally using commands.

A command is essentially a **transactional boundary**.

Because a command is a transactional boundary, when using the Domain Driven Development technique of an aggregate there is a natural affinity between the command, which operates on a transactional boundary and the aggregate which is a transactional boundary within the domain model.

The aggregate is the Receiver stereotype within the Command Design pattern.

Because we want to separate use of outgoing adapters via a secondary port, such as a Repository in the DDD case, this can lead to a pattern for implementation of a command:

1: Begin Transaction
2: Load from Repository
3: Operate on Aggregate
4: Flush to Repository
5: Commit Transaction

In the Repository pattern we may need to notify other aggregates that can be eventually consistent of the change within the transaction ally consistent boundary. The pattern suggested there is a notification. Because the handling of that notification is in itself likely to be a transactional boundary for a different aggregate we can encapsulate this domain event with the Command design pattern as well, which gives rise to the following additional step to the sequence, outside the original transactional boundary:

6: Invoke Command Encapsulating Notification

Note that this is, in essence, the actor model.

Implementing a Command

Implementing the pattern is simple, we implement a command interface, that has a method to execute the command. We then create a concrete instance of the command that derives from this interface. The invoking class executes the command without being directly coupled to the receiver which the command uses to implement the action requested.

From our perspective, the single method API endpoints are in fact commands. We could rewrite

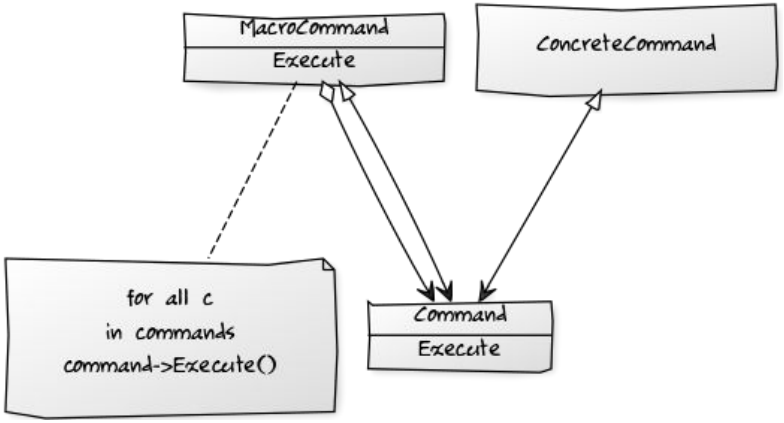
```
public class UpdateMyThingForFooinService : IUpdateMyThingForFooinService
{
  public void UpdateMyThingForFoo(/* .. parameters ...*/)
 {
   /*Other Stuff*/
 }
}
```

using an interface to represent a command as

```
public interface IAmACommand
{
   void Execute();
}

public class UpdateMyThingForFooFooCommand
{
   public UpdateMyThingForFooCommand(/* .. parameters ...*/)
   {
      /* Initialize state of command for members */
   }

  public void Execute()
  {
      /*Other Stuff*/
  }
}
```

Macro Command

A Macro command is a command that consists of other commands. At its simplest form this is just a class that contains a list of commands to execute in sequence. A Macro Command has no Receiver, because each Command in the list has its own Receiver



Command Dispatcher

To decouple a higher and lower layers we want to be able alter the implementation of the commands that we call on the lower layer without altering the calling layer.

We introduce a **Command Handler** for each **Command** that can be invoked, that all implement a specific interface. Command-handlers are registered with a **Command Dispatcher** (and can also be removed) at run-time.

When the application issues a command, it notifies the **Command Dispatcher** which dispatches to the command handler(s) associated with the command.

This decouples dispatch from processing.

The **Command Dispatcher** allows dynamic registration and removal of **Command Handlers**, it is an administrative entity that manages linking of commands to the appropriate command handlers.

It relates to the Observer pattern in that hooks together publishers and subscribers.

Command Dispatcher registration requires a key – provided by the Command Dispatcher for the **Commands** it can service, using getKey(). [In practice we often use RTTI for this].

The Command Handler is fired, when a command with the same name (key) is sent to the Command Dispatcher.

The Command Dispatcher is a repository of key-value pairs (key., Command Handler) and when the Command Dispatcher is called it looks up the command's key in the repository. If there is a match it calls the appropriate method(s) on the handler to process the Command.

Command Dispatcher

Invoker - has a lit of *Commands* that are to be executed

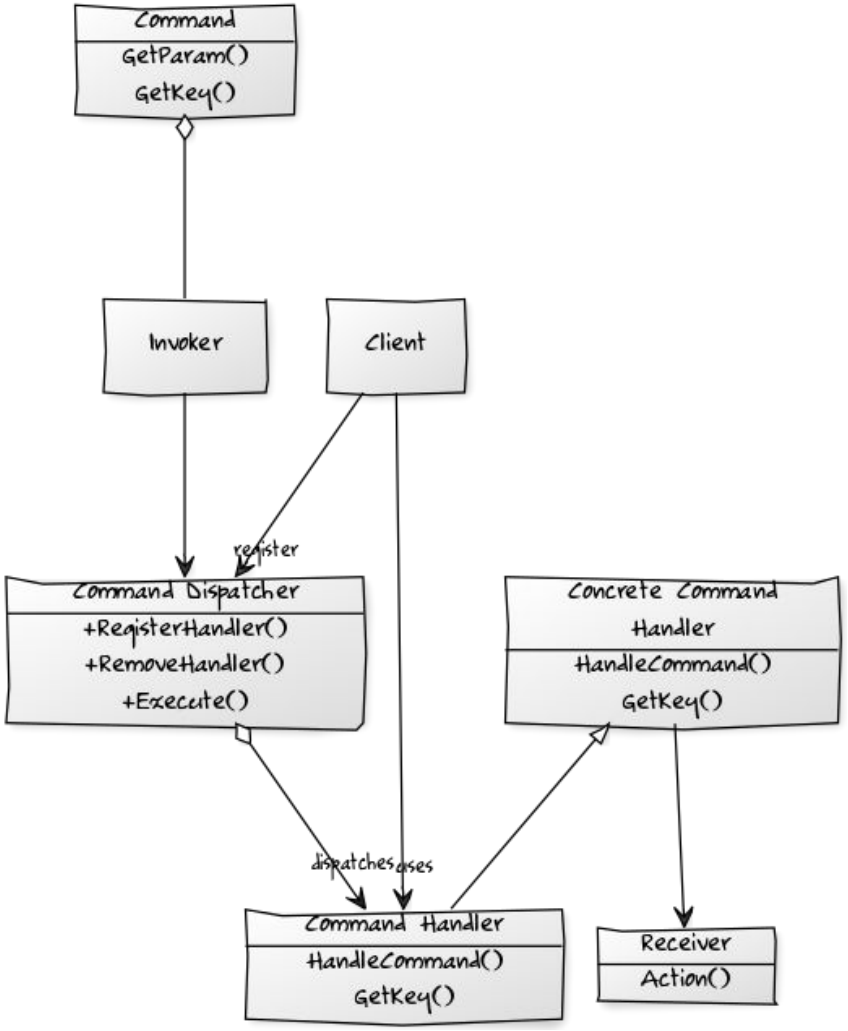
Command - represents the request to be processed, encapsulating the parameters to be passed to the command-handler to perform the request

Command Handler - specifies the interface that any command handler must implement

Concrete Command Handler – implements the request

Command Dispatcher – Allows dynamic registration of *Command Handlers* and looks up handlers for commands, by matching command and handler key.

Client – registers Commands with the Command Dispatcher.

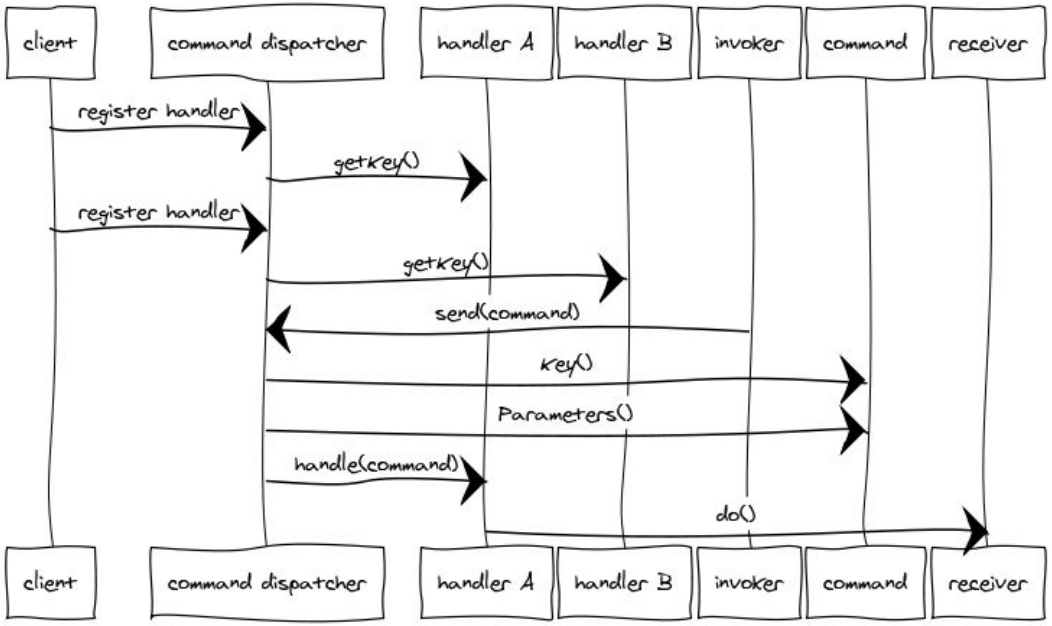


Command Dispatcher

We want to separate an **Action-Request** object that contains the identity of the action we want to perform, and the parameter for that action from the **Action-Handler** which knows how to perform that action.

A Command Dispatcher is an object that links the **Action-Request** with the appropriate **Action-Handler**.

We may distinguish between a **Command** Action-Request that has one Action Handler and an **Event** Action-Request that has many



Controlling Service Quality at Ports

QUALITY OF SERVICE

The Fallacies of Distributed Computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Calls to remote components, such as a database, broker, or http service can fail. Code that makes assumption that they will succeed is subject to the *fallacies of distributed computing*, in particular that the network is reliable.

For this reason we need to set a limit on the time we wait for a response from the a server (or other remote resource).

Otherwise we risk tying up a thread waiting for a response that will never come, potentially causing a cascade failure where once all the resources on the machine are tied up processing requests to which no response will ever come, we cannot process new requests.

If, in turn, those requests do not have a timeout, that initial failure may now bubble to our caller, and so on…

The Timeout Pattern

Provide a timeout on any inter-process operation.

The timeout pattern is usable with any client-server request.

The timeout pattern is also usable with thread resource pools.

Any operation that blocks a thread, should have a timeout

Usually a timeout is a parameter on an API call that indicates how long to wait [in milliseconds] for the operation to complete before returning control to the caller

If you are using a third-party API that has no timeout, you may need to make the request on a thread that does have a timeout, and when that thread times out assume that the operation failed.

Timeouts have a strong correlation with the retry pattern. It may seem sensible to retry, but beware that if a resource is failing under load further retries may simply overload that resource. Consider both exponential back-off and circuit breaker strategies.

Timeouts are a scaling issue. Because an API call without a timeout consumes resources, a stalled network call, or thread, prevents other requests from being executed. You might not notice the failure to use timeouts on a system with abundant resources, but as your resources dwindle failure to use timeouts to free resources will cause pain.

Assume that a timeout *will* occur at some point

Asynchronous operations still need to time out. All the caller may not be blocked, resources are still being consumed to manage the asynchronous operation that will not be released until the operation aborts.

The Retry Pattern

In the presence of unreliable calls between two components our first approach to achieving a high quality of service is to recognize that many such failures are transient: a timeout because a resource is busy, temporary loss of connectivity, the loss of one node that will be replaced by failover to another.

In this case the fault is self-correcting, the node comes up, the load on the database or server declines and their is capacity for our call, or network connectivity is restored. This means that our call will succeed if we retry after a delay to allow the transient fault to resolve.

The Retry pattern is simply that if the call fails, we can try again. It is important to have an upper bound on retries in case a fault that appears transient is not. See Circuit Breaker as well.

If the fault is permanent, and unlikely to succeed, for example a login failure due to invalid credentials – don't retry.

If the fault is caused by a rare event, for example packet loss or other corruption, consider an immediate retry as the server may be able to respond.

If the fault is caused by load, such as SQL Timeouts, or 429 Too Many Requests, then back-off for a period before retrying. Failure to observe this can lead to an Application Denial of Service Attack i.e. we overload a struggling server.

In the case of a 429 for example, the Retry-After header will tell you how long to back off for.

The Circuit Breaker Pattern

The Circuit Breaker pattern prevents an application from executing an operation that is likely to fail, thus freeing up resources that would otherwise be consumed waiting for a timeout and retry cycle to occur.

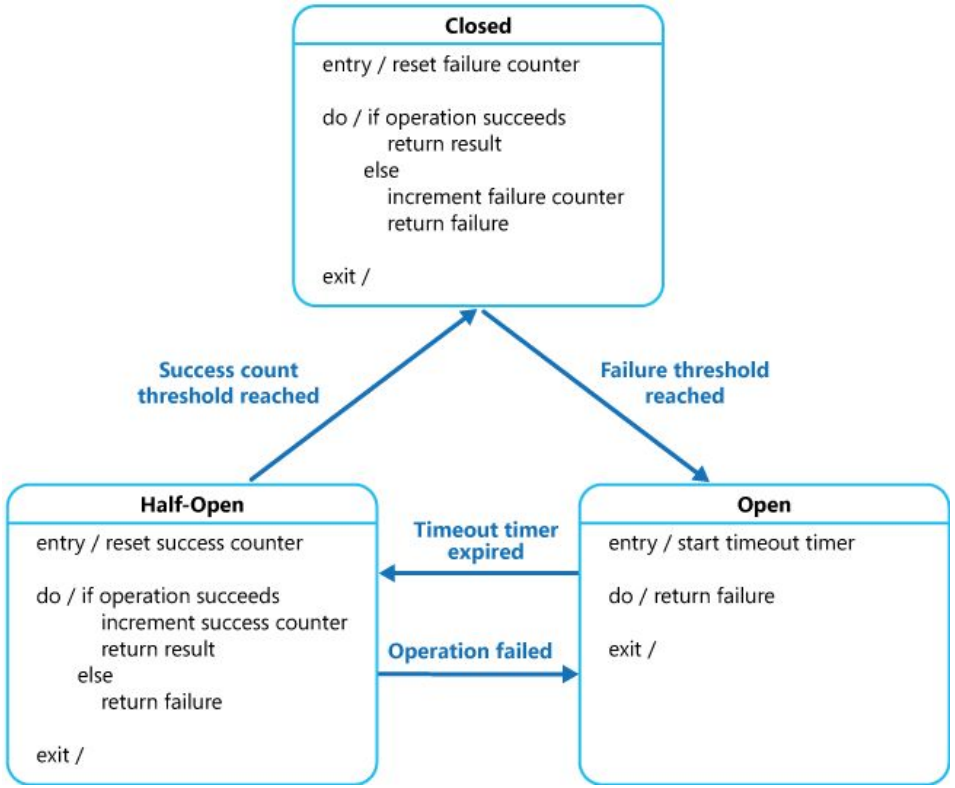
Once normal service has been restored to the server or resource pool the circuit breaker pattern allows detection of the resumption of service.

A Circuit Breaker acts as a proxy to operations that can fail. It has one of three states:

Closed: Requests are routed as normal, on a failure a counter is incremented and if the threshold is exceeded within a time limit, the circuit breaker opens.

Open: No calls are allowed, and are failed automatically by the proxy. After a specified time interval the circuit breaker is moved to half-open

Half-Open: A call is allowed. On a failure the breaker moves to Open, on a success it moves to Closed



From MSDN: http://msdn.microsoft.com/en-us/library/dn589784.aspx

Command Processor

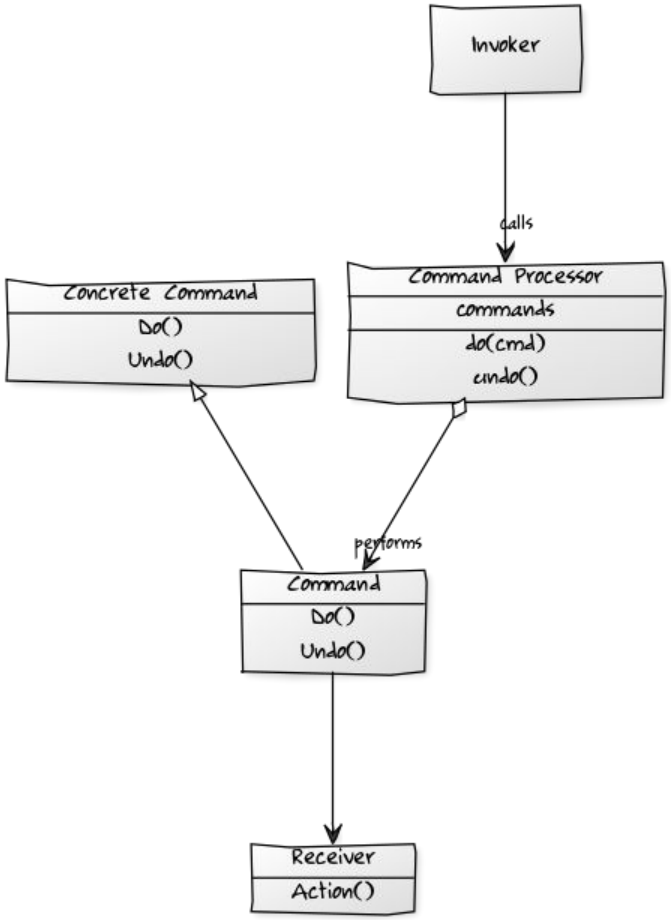
The Command Processor pattern separates the request for a service from its execution.

A Command Processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

A Command Dispatcher and a Command Processor are similar in that both divorce the caller of a Command from invoker of that Command.

However, the motivation is different. A Dispatcher seeks to decouple the caller from the invoker to allow us to easily extend the system without modification to the caller.

Conversely the motivation behind a Command Processor is to allows us to implement orthogonal operations such as logging, or scheduling without forcing the sender or receiver to be aware of them. It does this by giving those responsibilities to the invoker.



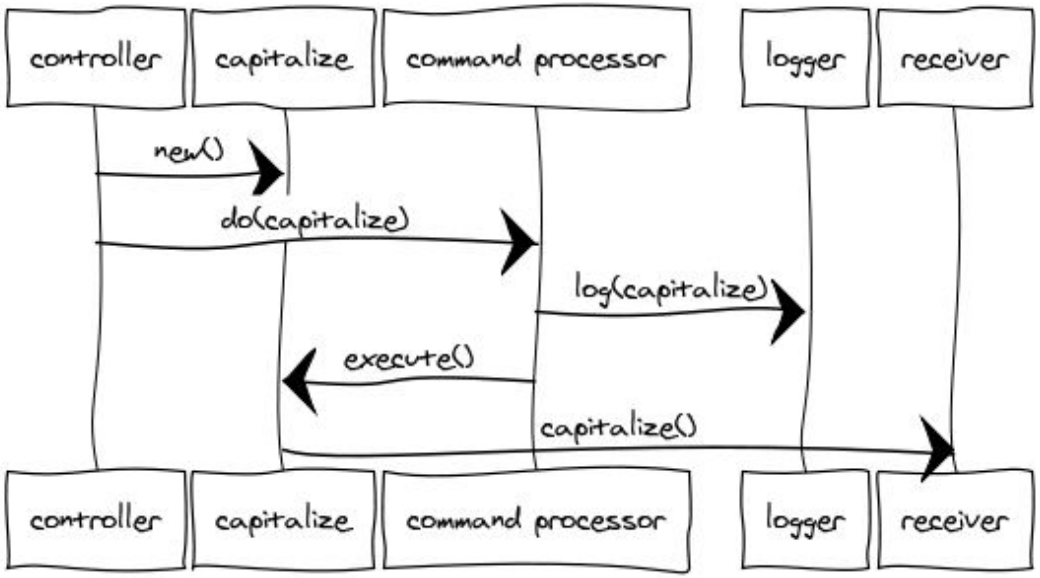
Command Processor

The Command Processor pattern provides details of how to manage a set of commands used to separate the request for service from its execution.

The Command Processor accepts requests for Commands, schedules and executes them, manages their history (such as storing for later undo or replay) and performs any other orthogonal housekeeping operations. The client is decoupled from the responsibilities of managing and scheduling commands.

A Command Processor enforces quality of service and maximizes throughput.

A Command Processor forms a juncture at which concerns like: *retry*, *timeout* and *circuit breaker* can be implemented for all commands.



Quality of Service

By Quality of Service we mean those aspects of an application that relate to meeting the quality attributes.

Quality attributes include: maintainability, usability, security, testability, availability, performance, interoperability

Using a command processor can help us meet QoS requirements that are orthogonal to the domain, e.g.:

- Logging
- Retry
- Timeout
- Circuit Breaker
- Authorization

One option is just to bake these into the Command Processor as steps that are always performed.

Another option is to use the Open-Closed Principle to provide an interface on commands that supports these concerns that the command processor calls i.e. undo

Perhaps a better option is to merge a command processor with a command dispatcher and recognize that the Command Handler can be a batch-sequence instead of a single handler, with a sequence of steps

This sequence can be configured to run orthogonal steps by configuration, or by convention.

Classic approaches are fluent declaration and attributes. Both have trade-offs, and adherents.

Example

HYSTRIX
DEFEND YOUR APP