Hexagonal architecture

Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, and link applications together without any user involvement.

(*Japanese translation of this article at* http://blog.tai2.net/hexagonal_architexture.html)
(*Spanish translation of this article at* http://academyfor.us/posts/arquitectura-hexagonal *courtesy of Arthur Mauricio Delgadillo*)
(*original explanation w updates at* http://wiki.c2.com/?HexagonalArchitecture *and* http://wiki.c2.com/?PortsAndAdaptersArchitecture)

See also Hexagonal Architecture FAQ

Hexagonal architecture pic 1-to-4 socket.jpg

The Pattern: Ports and Adapters (''Object Structural'')

Alternative name: ''Ports & Adapters''

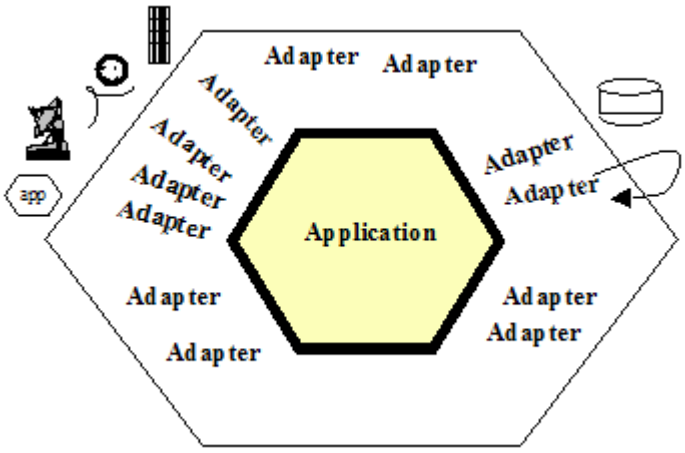
Alternative name: ''Hexagonal Architecture''

Intent

Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

As events arrive from the outside world at a port, a technology-specific adapter converts it into a usable procedure call or message and passes it to the application. The application is blissfully ignorant of the nature of the input device. When the application has something to send out, it sends it out through a port to an adapter, which creates the appropriate signals needed by the receiving technology (human or automated). The application has a semantically sound interaction with the adapters on all sides of it, without actually knowing the nature of the things on the other side of the adapters.

Figure 1 : Hexagonal architecture basic.gif



Motivation

One of the great bugaboos of software applications over the years has been infiltration of business logic into the user interface code. The problem this causes is threefold:

- First, the system can't neatly be tested with automated test suites because part of the logic needing to be tested is dependent on oft-changing visual details such as field size and button placement;
- For the exact same reason, it becomes impossible to shift from a human-driven use of the system to a batch-run system;
- For still the same reason, it becomes difficult or impossible to allow the program to be driven by another program when that becomes attractive.

The attempted solution, repeated in many organizations, is to create a new layer in the architecture, with the promise that this time, really and truly, no business logic will be put into the new layer. However, having no mechanism to detect when a violation of that promise occurs, the organization finds a few years later that the new layer is cluttered with business logic and the old problem has reappeared.

Imagine now that ''every'' piece of functionality the application offers were available through an API (application programmed interface) or function call. In this situation, the test or QA department can run automated test scripts against the application to detect when any new coding breaks a previously working function. The business experts can create automated test cases, before the GUI details are finalized, that tells the programmers when they have done their work correctly (and these tests become the ones run by the test department). The application can be deployed in ''headless'' mode, so only the API is available, and other programs can make use of its functionality — this simplifies the overall design of complex application suites and also permits business-to-business service applications to use each other without human intervention over the web. Finally, the automated function regression

Software engineering

Hexagonal

Other

What Links Here

tests detect any violation of the promise to keep business logic out of the presentation layer. The organization can detect, and then correct, the logic leak.

An interesting similar problem exists on what is normally considered “the other side” of the application, where the application logic gets tied to an external database or other service. When the database server goes down or undergoes significant rework or replacement, the programmers can’t work because their work is tied to the presence of the database. This causes delay costs and often bad feelings between the people.

It is not obvious that the two problems are related, but there is a symmetry between them that shows up in the nature of the solution.

Nature of the Solution

Both the user-side and the server-side problems actually are caused by the same error in design and programming — the entanglement between the business logic and the interaction with external entities. The asymmetry to exploit is not that between ‘’left’’ and ‘’right’’ sides of the application but between ‘’inside’’ and ‘’outside’’ of the application. The rule to obey is that code pertaining to the ‘’inside’’ part should not leak into the ‘’outside’’ part.

Removing any left-right or up-down asymmetry for a moment, we see that the application communicates over ‘’ports’’ to external agencies. The word “port” is supposed to evoke thoughts of ‘’ports’’ in an operating system, where any device that adheres to the protocols of a port can be plugged into it; and ‘’ports’’ on electronics gadgets, where again, any device that fits the mechanical and electrical protocols can be plugged in.

- The protocol for a port is given by the *purpose of the conversation* between the two devices.

The protocol takes the form of an application program interface (API).

For each external device there is an ‘’adapter’’ that converts the API definition to the signals needed by that device and vice versa. A graphical user interface or GUI is an example of an adapter that maps the movements of a person to the API of the port. Other adapters that fit the same port are automated test harnesses such as FIT or Fitnesse, batch drivers, and any code needed for communication between applications across the enterprise or net.

On another side of the application, the application communicates with an external entity to get data. The protocol is typically a database protocol. From the application’s perspective, if the database is moved from a SQL database to a flat file or any other kind of database, the conversation across the API should not change. Additional adapters for the same port thus include an SQL adapter, a flat file adapter, and most importantly, an adapter to a “mock” database, one that sits in memory and doesn’t depend on the presence of the real database at all.

Many applications have only two ports: the user-side dialog and the database-side dialog. This gives them an asymmetric appearance, which makes it seem natural to build the application in a one-dimensional, three-, four-, or five-layer stacked architecture.

There are two problems with these drawings. First and worst, people tend not to take the “lines” in the layered drawing seriously. They let the application logic leak across the layer boundaries, causing the problems mentioned above. Secondly, there may be more than two ports to the application, so that the architecture does not fit into the one-dimensional layer drawing.

The hexagonal, or ports and adapters, architecture solves these problems by noting the symmetry in the situation: there is an application on the inside communicating over some

number of ports with things on the outside. The items outside the application can be dealt with symmetrically.

The hexagon is intended to visually highlight

(a) the inside-outside asymmetry and the similar nature of ports, to get away from the one-dimensional layered picture and all that evokes, and

(b) the presence of a defined number of different ports – two, three, or four (four is most I have encountered to date).

The hexagon is not a hexagon because the number six is important, but rather to allow the people doing the drawing to have room to insert ports and adapters as they need, not being constrained by a one-dimensional layered drawing. The term ''hexagonal architecture'' comes from this visual effect.

The term "port and adapters" picks up the ''purposes'' of the parts of the drawing. A port identifies a purposeful conversation. There will typically be multiple adapters for any one port, for various technologies that may plug into that port. Typically, these might include a phone answering machine, a human voice, a touch-tone phone, a graphical human interface, a test harness, a batch driver, an http interface, a direct program-to-program interface, a mock (in-memory) database, a real database (perhaps different databases for development, test, and real use).

In the Application Notes, the left-right asymmetry will be brought up again. However, the primary purpose of this pattern is to focus on the inside-outside asymmetry, pretending briefly that all external items are identical from the perspective of the application.

Structure

Figure 2 : Hexagonal architecture with adapters.gif

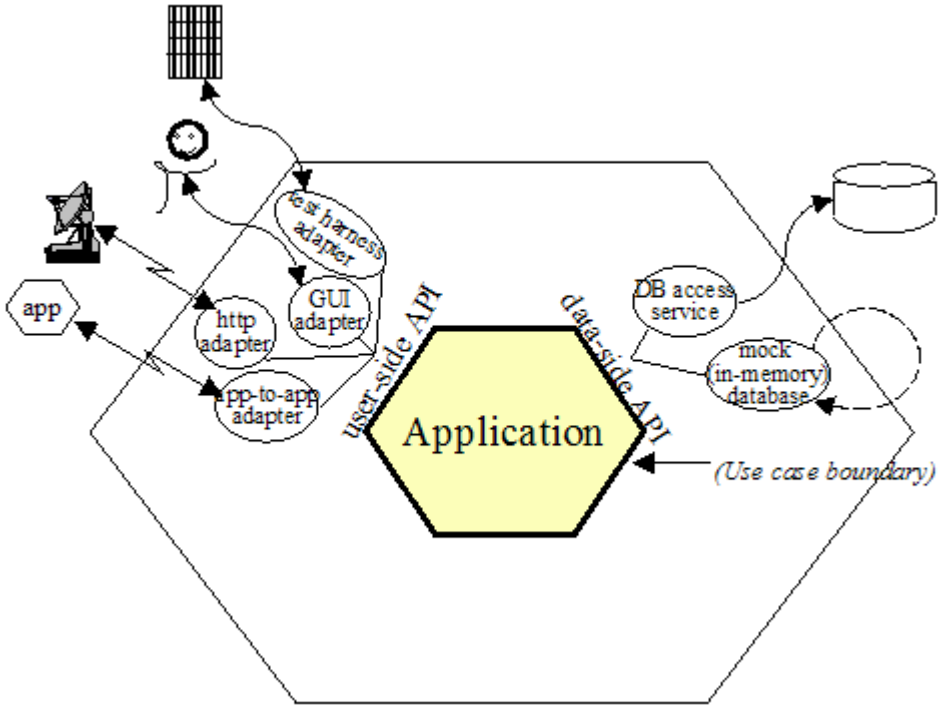


Figure 2 shows an application having two active ports and several adapters for each port. The two ports are the application-controlling side and the data-retrieval side. This drawing shows that the application can be equally driven by an automated, system-level regression

test suite, by a human user, by a remote http application, or by another local application. On the data side, the application can be configured to run decoupled from external databases using an in-memory oracle, or ''mock'', database replacement; or it can run against the test- or run-time database. The functional specification of the application, perhaps in use cases, is made against the inner hexagon's interface and not against any one of the external technologies that might be used.

Figure 3 : Hexagonal architecture barn door image.gif

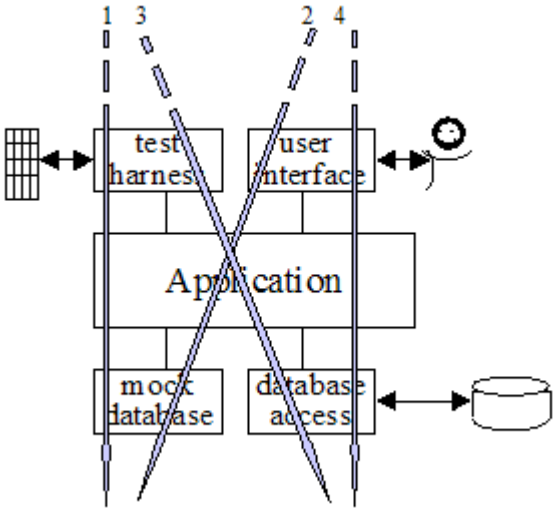


Figure 3 shows the same application mapped to a three-layer architectural drawing. To simplify the drawing only two adapters are shown for each port. This drawing is intended to show how multiple adapters fit in the top and bottom layers, and the sequence in which the various adapters are used during system development. The numbered arrows show the order in which a team might develop and use the application:

1. With a FIT test harness driving the application and using the mock (in-memory) database substituting for the real database;
2. Adding a GUI to the application, still running off the mock database;
3. In integration testing, with automated test scripts (e.g., from Cruise Control) driving the application against a real database containing test data;
4. In real use, with a person using the application to access a live database.

Sample Code

The simplest application that demonstrates the ports & adapters fortunately comes with the FIT documentation. It is a simple discount computing application:

```
discount(amount) = amount * rate(amount);
```

In our adaptation, the amount will come from the user and the rate will come from a database, so there will be two ports. We implement them in stages:

- With tests but with a constant rate instead of a mock database,
- then with the GUI,
- then with a mock database that can be swapped out for a real database.

Thanks to Gyan Sharma at IHC for providing the code for this example.

Stage 1: FIT App constant-as-mock-database

First we create the test cases as an HTML table (see the FIT documentation for this):

TestDiscounter	
amount	discount()
100	5
200	10

Note that the column names will become class and function names in our program. FIT contains ways to get rid of this “programmerese”, but for this article it is easier just to leave them in.

Knowing what the test data will be, we create the user-side adapter, the ColumnFixture that comes with FIT as shipped:

```
import fit.ColumnFixture;
public class TestDiscounter extends ColumnFixture
{
   private Discounter app = new Discounter();
   public double amount;
   public double discount()
   { return app.discount(amount); }
}
```

That’s actually all there is to the adapter. So far, the tests run from the command line (see the FIT book for the path you’ll need). We used this one:

```
set FIT_HOME=/FIT/FitLibraryForFit15Feb2005
java -cp %FIT_HOME%/lib/javaFit1.1b.jar;%FIT_HOME%/dist/fitLibraryForF
fit.FileRunner test/Discounter.html TestDiscount_Output.html
```

FIT produces an output file with colors showing us what passed (or failed, in case we made a typo somewhere along the way).

At this point the code is ready to check in, hook into Cruise Control or your automated build machine, and include in the build-and-test suite.

Stage 2: UI App constant-as-mock-database

I’m going to let you create your own UI and have it drive the Discounter application, since the code is a bit long to include here. Some of the key lines in the code are these:

```
...
 Discounter app = new Discounter();
public void actionPerformed(ActionEvent event)
{
    ...
   String amountStr = text1.getText();
   double amount = Double.parseDouble(amountStr);
   discount = app.discount(amount));
   text3.setText( "" + discount );
   ...
```

At this point the application can be both demoed and regression tested. The user-side adapters are both running.

Stage 3: (FIT or UI) App mock database

To create a replaceable adapter for the database side, we create an ‘‘interface’’ to a repository, a ‘‘RepositoryFactory’’ that will produce either the mock database or the real service object, and the in-memory mock for the database.

```
public interface RateRepository
{
   double getRate(double amount);
 }
```

```
public class RepositoryFactory
{
   public RepositoryFactory() {  super(); }
   public static RateRepository getMockRateRepository()
   {
      return new MockRateRepository();
   }
}
public class MockRateRepository implements RateRepository
{
   public double getRate(double amount)
   {
      if(amount <= 100) return 0.01;
      if(amount <= 1000) return 0.02;
      return 0.05;
    }
 }
```

To hook this adapter into the Discounter application, we need to update the application itself to accept a repository adapter to use, and the have the (FIT or UI) user-side adapter pass the repository to use (real or mock) into the constructor of the application itself. Here is the updated application and a FIT adapter that passes in a mock repository (the FIT adapter code to choose whether to pass in the mock or real repository's adapter is longer without adding much new information, so I omit that version here).

```
import repository.RepositoryFactory;
import repository.RateRepository;
public class Discounter
{
   private RateRepository rateRepository;
   public Discounter(RateRepository r)
   {
      super();
      rateRepository = r;
    }
   public double discount(double amount)
   {
      double rate = rateRepository.getRate( amount );
      return amount * rate;
    }
}
import app.Discounter;
import fit.ColumnFixture;
public class TestDiscounter extends ColumnFixture
{
   private Discounter app =
       new Discounter(RepositoryFactory.getMockRateRepository());
   public double amount;
   public double discount()
   {
      return app.discount( amount );
   }
}
```

That concludes implementation of the simplest version of the hexagonal architecture.

For a different implementation, using Ruby and Rack for browser usage, see https://github.com/totheralistair/SmallerWebHexagon

Application Notes

The Left-Right Asymmetry

The ports and adapters pattern is deliberately written pretending that all ports are fundamentally similar. That pretense is useful at the architectural level. In implementation, ports and adapters show up in two flavors, which I'll call ''primary'' and ''secondary'', for soon-to-be-obvious reasons. They could be also called ''driving'' adapters and ''driven'' adapters.

The alert reader will have noticed that in all the examples given, FIT fixtures are used on the left-side ports and mocks on the right. In the three-layer architecture, FIT sits in the top layer and the mock sits in the bottom layer.

This is related to the idea from use cases of "primary actors" and "secondary actors". A ''primary actor'' is an actor that drives the application (takes it out of quiescent state to perform one of its advertised functions). A ''secondary actor'' is one that the application drives, either to get answers from or to merely notify. The distinction between ''primary ''and'' secondary ''lies in who triggers or is in charge of the conversation.

The natural test adapter to substitute for a ''primary'' actor is FIT, since that framework is designed to read a script and drive the application. The natural test adapter to substitute for a ''secondary'' actor such as a database is a mock, since that is designed to answer queries or record events from the application.

These observations lead us to follow the system's use case context diagram and draw the ''primary ports ''and'' primary adapters'' on the left side (or top) of the hexagon, and the ''secondary ports'' and ''secondary adapters'' on the right (or bottom) side of the hexagon.

The relationship between primary and secondary ports/adapters and their respective implementation in FIT and mocks is useful to keep in mind, but it should be used as a consequence of using the ports and adapters architecture, not to short-circuit it. The ultimate benefit of a ports and adapters implementation is the ability to run the application in a fully isolated mode.

Use Cases And The Application Boundary

It is useful to use the hexagonal architecture pattern to reinforce the preferred way of writing use cases.

A common mistake is to write use cases to contain intimate knowledge of the technology sitting outside each port. These use cases have earned a justifiably bad name in the industry for being long, hard-to-read, boring, brittle, and expensive to maintain.

Understanding the ports and adapters architecture, we can see that the use cases should generally be written at the application boundary (the inner hexagon), to specify the functions and events supported by the application, regardless of external technology. These use cases are shorter, easier to read, less expensive to maintain, and more stable over time.

How Many Ports?

What exactly a port is and isn't is largely a matter of taste. At the one extreme, every use case could be given its own port, producing hundreds of ports for many applications. Alternatively, one could imagine merging all primary ports and all secondary ports so there are only two ports, a left side and a right side.

Neither extreme appears optimal.

The weather system described in the Known Uses has four natural ports: the weather feed, the administrator, the notified subscribers, the subscriber database. A coffee machine controller has four natural ports: the user, the database containing the recipes and prices, the dispensers, and the coin box. A hospital medication system might have three: one for the nurse, one for the prescription database, and one for the computer-controller medication dispensers.

It doesn't appear that there is any particular damage in choosing the "wrong" number of ports, so that remains a matter of intuition. My selection tends to favor a small number, two, three or four ports, as described above and in the Known Uses.

Known Uses

Figure 4 : Hexagonal architecture complex example.gif

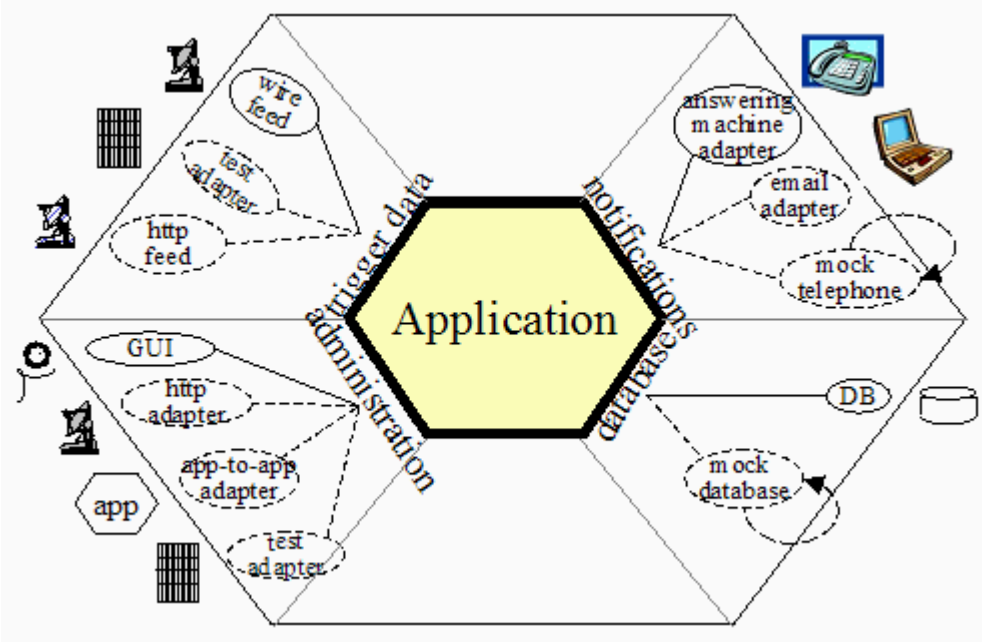


Figure 4 shows an application with four ports and several adapters at each port. This was derived from an application that listened for alerts from the national weather service about earthquakes, tornadoes, fires and floods, and notified people on their telephones or telephone answering machines. At the time we discussed this system, the system's interfaces were identified and discussed by ''technology, linked to purpose''. There was an interface for trigger-data arriving over a wire feed, one for notification data to be sent to answering machines, an administrative interface implemented in a GUI, and a database interface to get their subscriber data.

The people were struggling because they needed to add an http interface from the weather service, an email interface to their subscribers, and they had to find a way to bundle and unbundle their growing application suite for different customer purchasing preferences. They feared they were staring at a maintenance and testing nightmare as they had to implement, test and maintain separate versions for all combinations and permutations.

Their shift in design was to architect the system's interfaces ''by purpose'' rather than by technology, and to have the technologies be substitutable (on all sides) by adapters. They immediately picked up the ability to include the http feed and the email notification (the new adapters are shown in the drawing with dashed lines). By making each application executable in headless mode through APIs, they could add an app-to-add adapter and unbundle the application suite, connecting the sub-applications on demand. Finally, by making each application executable completely in isolation, with test and mock adapters in place, they gained the ability to regression test their applications with stand-alone automated test scripts.

Mac, Windows, Google, Flickr, Web 2.0

In the early 1990s, MacIntosh applications such as word processor applications were required to have API-drivable interfaces, so that applications and user-written scripts could access all the functions of the applications. Windows desktop applications have evolved the same ability (I don't have the historical knowledge to say which came first, nor is that relevant to the point).

The current (2005) trend in web applications is to publish an API and let other web applications access those APIs directly. Thus, it is possible to publish local crime data over a Google map, or create web applications that include Flickr's photo archiving and annotating abilities.

All of these examples are about making the ''primary ''ports' APIs visible. We see no information here about the secondary ports.

Stored Outputs

This example written by Willem Bogaerts on the C2 wiki:

"I encountered something similar, but mainly because my application layer had a strong tendency to become a telephone switchboard that managed things it should not do. My application generated output, showed it to the user and then had some possibility to store it as well. My main problem was that you did not need to store it always. So my application generated output, had to buffer it and present it to the user. Then, when the user decided that he wanted to store the output, the application retrieved the buffer and stored it for real.

I did not like this at all. Then I came up with a solution: Have a presentation control with storage facilities. Now the application no longer channels the output in different directions, but it simply outputs it to the presentation control. It's the presentation control that buffers the answer and gives the user the possibility to store it.

The traditional layered architecture stresses "UI" and "storage" to be different. The Ports and Adapters Architecture can reduce output to being simply "output" again. "

Anonymous example from the C2-wiki

"In one project I worked on, we used the SystemMetaphor of a component stereo system. Each component has defined interfaces, each of which has a specific purpose. We can then connect components together in almost unlimited ways using simple cables and adapters."

Distributed, Large-Team Development

This one is still in trial use and so does not properly count as a use of the pattern. However, it is interesting to consider.

Teams in different locations all build to the Hexagonal architecture, using FIT and mocks so the applications or components can be tested in standalone mode. The CruiseControl build runs every half hour and runs all the applications using the FIT+mock combination. As application subsystem and databases get completed, the mocks are replaced with test databases.

Separating Development of UI and Application Logic

This one is still in early trial use and so does not count as a use of the pattern. However, it is interesting to consider.

The UI design is unstable, as they haven't decided on a driving technology or a metaphor yet. The back-end services architecture hasn't been decided, and in fact will probably change several times over the next six months. Nonetheless, the project has officially started and time is ticking by.

The application team creates FIT tests and mocks to isolate their application, and creates testable, demonstrable functionality to show their users. When the UI and back-end services decisions finally get met, it “should be straightforward” to add those elements the application. Stay tuned to learn how this works out (or try it yourself and write me to let me know).

Related Patterns

Adapter

The ‘’Design Patterns’’ book contains a description of the generic ‘’Adapter’’ pattern: “Convert the interface of a class into another interace clients expect.” The ports-and-adapters pattern is a particular use of the ‘’Adapter’’ pattern.

Model-View-Controller

The MVC pattern was implemented as early as 1974 in the Smalltalk project. It has been given, over the years, many variations, such as Model-Interactor and Model-View-Presenter. Each of these implements the idea of ports-and-adapters on the primary ports, not the secondary ports.

Mock Objects and Loopback

“A mock object is a “double agent” used to test the behaviour of other objects. First, a mock object acts as a faux implementation of an interface or class that mimics the external behaviour of a true implementation. Second, a mock object observes how other objects interact with its methods and compares actual behaviour with preset expectations. When a discrepancy occurs, a mock object can interrupt the test and report the anomaly. If the discrepancy cannot be noted during the test, a verification method called by the tester ensures that all expectations have been met or failures reported.” — From http://MockObjects.com

Fully implemented according to the mock-object agenda, mock objects are used throughout an application, not just at the external interface The primary thrust of the mock object movement is conformance to specified protocol at the individual class and object level. I borrow their word “mock” as the best short description of an in-memory substitute for an external secondary actor.

The Loopback pattern is an explicit pattern for creating an internal replacement for an external device.

Pedestals

In “Patterns for Generating a Layered Architecture”, Barry Rubel describes a pattern about creating an axis of symmetry in control software that is very similar to ports and adapters. The ‘’Pedestal’’ pattern calls for implementing an object representing each hardware device within the system, and linking those objects together in a control layer. The ‘’Pedestal’’ pattern can be used to describe either side of the hexagonal architecture, but does not yet stress the similarity across adapters. Also, being written for a mechanical control environment, it is not so easy to see how to apply the pattern to IT applications.

Checks

Ward Cunningham’s pattern language for detecting and handling user input errors, is good for error handling across the inner hexagon boundaries.

Dependency Inversion (Dependency Injection) and SPRING

Bob Martin's Dependency Inversion Principle (also called Dependency Injection by Martin Fowler) states that "High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions." The ''Dependency Injection ''pattern by Martin Fowler gives some implementations. These show how to create swappable secondary actor adapters. The code can be typed in directly, as done in the sample code in the article, or using configuration files and having the SPRING framework generate the equivalent code.

Acknowledgements

Thanks to Gyan Sharma at Intermountain Health Care for providing the sample code used here. Thanks to Rebecca Wirfs-Brock for her book ''Object Design'', which when read together with the ''Adapter'' pattern from the ''Design Patterns'' book, helped me to understand what the hexagon was about. Thanks also to the people on Ward's wiki, who provided comments about this pattern over the years (e.g., particularly Kevin Rutherford's http://silkandspinach.net/blog/2004/07/hexagonal_soup.html).

References and Related Reading

FIT, A Framework for Integrating Testing: Cunningham, W., online at http://fit.c2.com, and Mugridge, R. and Cunningham, W., ''Fit for Developing Software'', Prentice-Hall PTR, 2005.

The ''Adapter'' pattern: in Gamma, E., Helm, R., Johnson, R., Vlissides, J., ''Design Patterns'', Addison-Wesley, 1995, pp. 139-150.

The ''Pedestal'' pattern: in Rubel, B., "Patterns for Generating a Layered Architecture", in Coplien, J., Schmidt, D., ''PatternLanguages of Program Design'', Addison-Wesley, 1995, pp. 119-150.

The ''Checks'' pattern: by Cunningham, W., online at http://c2.com/ppr/checks.html

The ''Dependency Inversion Principle'': Martin, R., in ''Agile Software Development Principles Patterns and Practices'', Prentice Hall, 2003, Chapter 11: "The Dependency-Inversion Principle", and online at http://www.objectmentor.com/resources/articles/dip.pdf

The ''Dependency Injection'' pattern: Fowler, M., online at http://www.martinfowler.com/articles/injection.html

The ''Mock Object'' pattern: Freeman, S. online at http://MockObjects.com

The ''Loopback'' pattern: Cockburn, A., online at http://c2.com/cgi/wiki?LoopBack

''Use cases:'' Cockburn, A., ''Writing Effective Use Cases'', Addison-Wesley, 2001, and Cockburn, A., "Structuring Use Cases with Goals", online at http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm

Comments from the old site:

André Boonzaaijer's blog While True discusses an application using the Hexagonal architecture and also has a cool picture of the architecture.

Kevin Rutherford has started several notes and discussions around it:

- Gravity and software adaptability
- Hexagonal architecture
- Databases as life-support for domain objects
- Hexagonal soup

Timo wrote a piece called Wrap it thinly about its use with TDD.

Gerard Meszaros in his book on Xunit patterns wrote http://xunitpatterns.com/Hexagonal%20Architecture.html.

Brian Anderson spent several blog entries noodling over it:

> Success!
> Problems with Smart Clients today
> compile time vs runtime views
> the use of symmetry in the hexagonal approach
> Back to Hexagonal Architecture
> some thoughts on the “Design Pattern” pattern
> http://www.brianmandersen.com/blog/page/2/

The original page was on Ward’s wiki at http://c2.com/cgi/wiki?HexagonalArchitecture

Utah Code Camp Sept 19, 2009 : Coding Assignment

The simplest application comes with the FIT documentation. It is a simple discount computing application:

```
discount(amount) = amount * discountRate(amount);
```

- ‘Amount’ comes from the user or a test framework (or a file)
- ‘Rate’ comes from a database or an in-memory mock of a database

Implement the pp in stages:

1. Input from a test framework, using a constant for the discountRate,
2. Input from a GUI, still using a constant for the discountRate.
3. Input from either test or GUI, discountRate from a mock database that can be swapped out for a real database.

Ruby / Rack (no Rails) implementation

I made a small web reader version of this, see it at

- https://github.com/totheralistair/SmallerWebHexagon

That one takes either browser, or Rack driver, or just a test driver on the left side, and either a constant, or in-code, or from-a-file rate db on the right side. I wrote this to show the implementation in a simple but real (2014) setting.

Other discussions and implementations

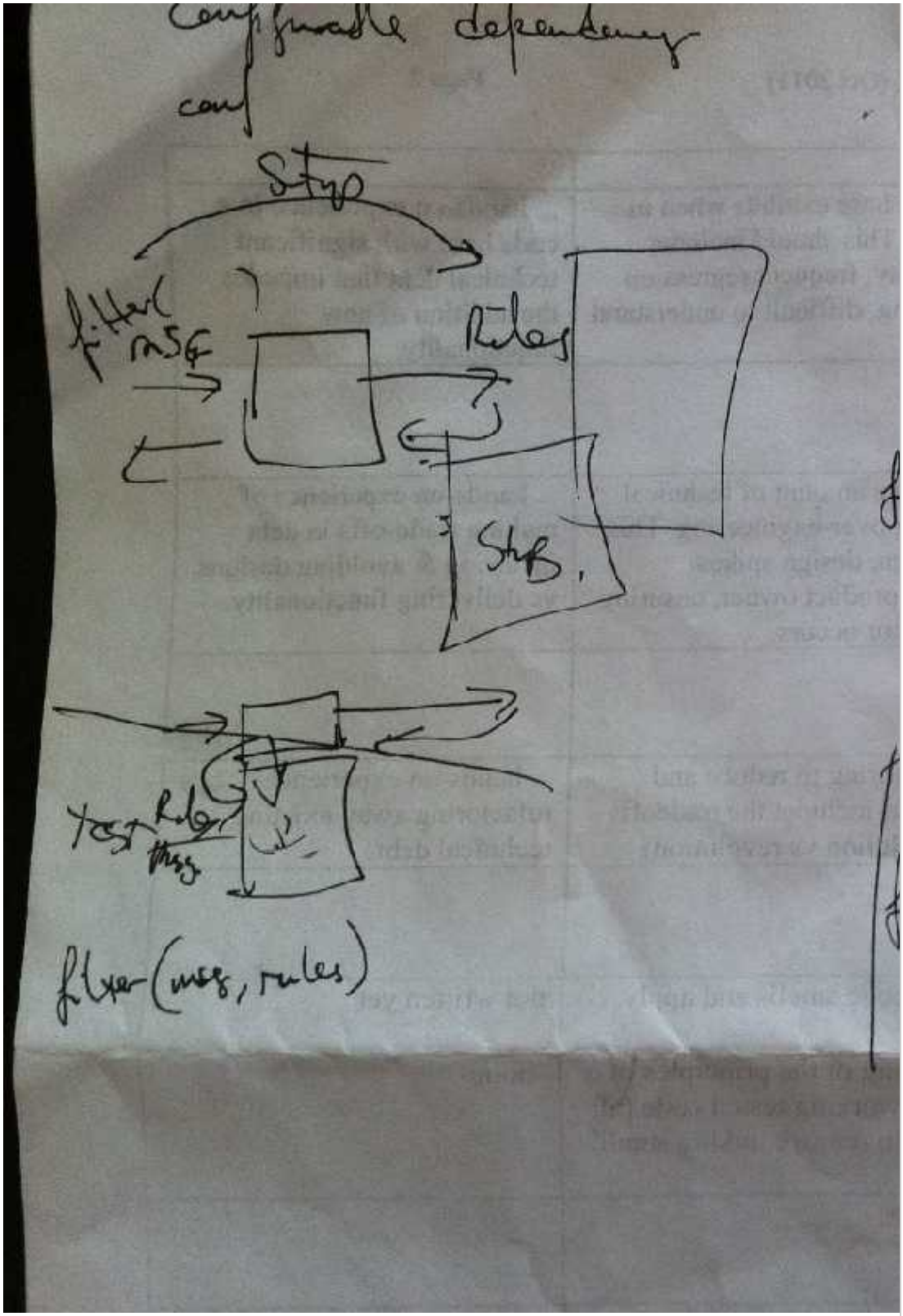
You can find more online about this architecture by searching through Google or Twitter (in particular). Also see:

- http://tpierrain.blogspot.fr/2013/08/a-zoom-on-hexagonalcleanonion.html
- The slide show for the talk on this I gave at from Utah Code is http://alistair.cockburn.us/Hexagonal+Architecture+Keynote+at+Utah+Code+Camp+Septem
- also nice elaboration with notes for himself by Duncan Nisbet at http://www.duncannisbet.co.uk/hexagonal-architecture-for-testers-part-1
- my video lightning talk at the Mountain West Ruby Conference in 2010: Video of Alistairs hexagonal architecture CQRS lightning talk Mountain West Ruby Conference 2010
- https://twitter.com/search?q=%22hexagonal%20architecture%22
- http://twitter.com/andrzejkrzywda/status/267420878487310336

- (user-port only) a tiny CMS https://github.com/totheralistair/SmallWebHexagon
- https://github.com/patmaddox/hexarch2 Pat Maddox starts with a Hexagonal Architecture and morphs it into Event Based and then CQRS. Take a look.
- Lovely long detailed discussion about hexagonal architecture and Rails with BadrinathJanakiraman and Martin Fowler: http://thoughtworks.wistia.com/medias/uxjb0lwrcz
- detailed evolution at https://github.com/Lunch-box/SimpleOrderRouting/wiki/Logbook-4#day-15-october-27th-2014

Configurable Dependencies, Primary and Secondary Actors

I tried to make this pattern truly symmetric, hence the hexagon. However, watching several implementations, it slowly became clear that there is an asymmetry (which is one thing that makes this fundamentally different from neighboring patterns such as the onion architecture). As stated above (see The Left-Right Asymmetry), the asymmetry matches Ivar Jacobson's **primary** and **secondary actors** concept, and affects how the Configurable Dependency is implemented. (This is shown briefly in the Configurable Dependency sketch:



Configurable Dependency illustrated1-800pxV.jpg

The difference between a primary and secondary actor lies only in who *initiates* the conversation. A **primary actor** knows about and initiates the conversation with the system or application; for a **secondary actor**, it is the system or application that knows about and initiates the discussion with the other. That is actually the only difference between the two, in use case land.

In implementation, that difference matters: Whomever will initiate the conversation must be handed the handle for the other.

In the case of **primary actor ports**, the macro constructor will pass to the UI, test framework, or driver the handle for the app and say, “Go talk to that”. The primary actor will call into the app, and the app will probably never know who called it. (That is normal for recipients of a call).

In stark contrast, for **secondary actor ports**, the macro constructor will pass to the UI, test framework, or driver the handle for the secondary actor to be used, that will get passed in as a parameter to the app, and the app will now know who/what is the recipient of the outgoing call. (This is again normal for sending out a call).

Thus, the system or application is constructed differently for primary and secondary actor ports: ignorant and initially passive for the primary actors, and having a way to store and call out to the secondary actor ports.

Both ports implement Configurable Dependency , but differently.

Simple example for a 3-port system, such as a coffee machine or a hospital medical unit dispensing medications intravenously (How odd that they come out the same, architecturally!):

- The purchaser, test harness or hospital admin is a primary actor driving the system
- The recipe database or medical database is a secondary actor, offering its information
- The chemical dispensers in either are secondary actors.

End result: the primary / secondary aspect of a port cannot be ignored.

See also Hexagonal Architecture FAQ

Discussion

Edit

Spanish translation of this article at
http://academyfor.us/posts/arquitectura-hexagonal
— courtesy of Arthur

AndrÃ© Boonzaaijer’s blog "While True" discusses an application using the Hexagonal architecture and also has a cool picture of the architecture.

Kevin Rutherford has started several notes and discussions around it:

> http://silkandspinach.net/2005/11/28/gravity-and-software-adaptability
> http://wordpress.com/tag/hexagonalarchitecture
> http://silkandspinach.net/2005/05/23/databases-as-life-support-for-domain-objects
> http://silkandspinach.net/2004/07/16/hexagonal-soup

Timo wrote a piece called http://ng-embedded.blogspot.com/2007/07/wrap-it-thinly.html about its use with TDD.

Gerard Meszaros in his book on Xunit patterns wrote
http://xunitpatterns.com/Hexagonal%20Architecture.html

Brian Anderson spent several blog entries noodling over it:

> http://www.brianmandersen.com/blog/2005/03/29/success/
> "Problems with Smart Clients today"

"compile time vs runtime views"
"the use of symmetry in the hexagonal approach"
"Back to Hexagonal Architecture"
"some thoughts on the 'Design Pattern' pattern"
http://www.brianmandersen.com/blog/page/2/

The original page was on Ward's wiki at http://c2.com/cgi/wiki?HexagonalArchitecture

-by Alistair on 10/05/2008 at 3:00 PM

I'd like to add the Naked Objects pattern to the list of known uses.

The (open-source) Naked Objects framework is most well-known for its ability to automatically build an object-oriented user interface for domain objects at runtime, the two main implementations being a rich-client, and an HTML viewer. All the developer writes is the domain objects (POJOs), and the user interface "comes for free".

The latest version, NO 4.0, also adds in the ability to exercise and interact with the domain model using generic FitNesse fixtures. So one can modify state, invoke actions, assert business rules and so on. Again, no custom FitNesse coding is required.

For the persistence layer, NO has long had the ability to switch between in-memory object store and other object stores (such as a Hibernate-based one). We've found this immensely useful, especially combined with the FitNesse stuff.

If you're interested in learning more, [and I hope a tiny bit of self-promotion here isn't inappropriate] I write about NO and its implementation to the hexagonal architecture in my pragprog book, Domain-Driven Design using Naked Objects (http://www.pragprog.com/titles/dhnako).

Cheers
Dan

-by Dan Haywood on 5/15/2009 at 8:06 AM

(Thanks Dan … Looking forward to your book. Alistair)

Matteo Vaccari shared his programming kata on hexagonal architecture at http://matteo.vaccari.name/blog/archives/154. Thanks, Matteo!

-by Alistair on 2/20/2010 at 7:25 PM

On Thu, Mar 12, 2009 at 7:43 AM, Rickard Öberg wrote at http://www.mail-archive.com/qi4j-dev@lists.ops4j.org/msg02835.html

Hey,

So, yesterday I tried reworking my StreamFlow workflow app into using the hexagonal architecture. So far I am extremely happy with the results. One of the things I have had big trouble with before is to implement the "TellDontAsk" principle. It seemed like no matter what I did I had to, in the end, ask for model information in various ways, thus showing all the inner details that I had been trying to encapsulate with my private mixins etc.

With the hexagonal architecture, where UI can be "at the bottom", and considered "output", this problem went away. Let me give you an example. In the app there is a search field and a search result view. In a normal layered app there would be a UI component that takes the search string and sends it to the application layer, and then presents the results. The app layer would have a method like this:

SearchResult search(String query);

This is very problematic though: first of all the search field has to know about the search result view, so they are coupled. If I then also want to update some other part in the UI the search field has to know about this too. Also, it is highly likely that once I get the result, I have to query the application for other things in order to present the result.

With hexagonal architecture this mess goes away. Since the flow is only “in-out” rather than “up-down-up”, the application layer method becomes:

void search(String query)

The application layer performs the query. When it is done it then simply looks up all services that implement SearchObserver, iterates over them, and pass the result to them. This can be easily done with a SideEffect of the search method, and gives a good example of when to use SideEffects. The code is something like this in the SideEffect:

```
@Service Iterable < SearchObserverobservers;
@This Searcher searcher.
public void search(String query)
 {
   for(SearchObserver observer: observers)
   {
      observer.refresh(searcher.searchResult().get());
   }
 }
```

Since the app layer uses() the UI layer, one of the observers just happens to be the search result view, which presents the results. If there had been a status bar it could have also consumed the results and showed a message like “Found 14 matches”. Or more like, a SearchStatus service would have Observed the search results, which would have produced the string, which is then in turn sent to StatusObservers, one of which happens to be the status bar.

If the search takes a long time, the UI would be in trouble with the first method, as it would essentially freeze when calling search. With hexagonal architecture the search(string) method can accept the string, return immediately, and then spawn off an asynchronous search that only when completed notifies the observers. The time between search and result can be quite long, but the UI will still be responsive in between, without the UI having to do the thread trickery. When consuming the results the UI does, however, have to ensure that it is on the Swing thread.

In any case, a key point is that the search field **does not have to know** how to present the results. All it does is take the string and send it to the application for querying. What happens then is up to the application and observers of the model that the processing changes. Input and output are separated in code, but still both are presented on the UI screen.

In this way there **is only TELL**, no ask. All events come from the outside, goes to the inside (through app-domain), and then goes out again. And sometimes the initiator (UI) just happens to be the output too.

This would also simplify testing, as the call to the app and introspection of the resulting model using a mock observer is quite easy to do.

NEAT!

Rickard

-by Alistair on 2/20/2010 at 8:01 PM

Hi Alistair,

It is mentioned in this article that with mvc ports and adapter are present for primary ports and not for secondary ports! how? Is it like we have api's for view in mvc. I am not getting this statement. Is it possible to elaborate here with detail of your thoughts on this?

Thanks,
Ak

-by Ak on 4/9/2010 at 2:10 AM

Thanks, Dan, for the Naked Objects use of this pattern; I particularly liked your comment about the RESTful API at http://danhaywood.com/2009/07/24/hexagonal-architecture-for-naked-objects/comment-page-1/#comment-710

cheers, Alistair

-by Alistair on 4/12/2010 at 1:02 AM

Hi Ak

I think that the point about MVC not really being a ports & adapter pattern comes down to the fact that may implementations of MVC allow a "fast-path" from the View to the Model, typically for bulk data retrieval during population of of controls like lists and trees. If you force all communication from the View to go to the Controller and use the Controller as the single point of access to the Model then I think you can argue that MVC can be a variation on the ports & adapters pattern.

-by Jonathan on 4/14/2010 at 9:09 AM

Good read. I have nothing further of interest to add. Thank you.

-by Jacolyte on 7/13/2010 at 7:38 PM

Just saw http://hendryluk.wordpress.com/2009/08/17/software-development-fundamentals-part-2-layered-architecture/ in which Hendry Luk derives the same architecture (without the hexagonal shape) from dependency considerations. Nicely done, nice read. —Alistair

-by Alistair on 8/1/2010 at 11:27 PM

Dear Mr. Cockburn

Posted by: **Alistair** on **6/19/2008 1:07:02 PM** Last modified by: **Alistair** on **9/24/2017 11:16:08 AM** Visits: **758979**

I've read your post on hexagonal architecture. I think you've made clear where enterprise software development should be going. I've been working on adapter-based standalone Java enterprise (web)applications for the last 7 years. The largest online supermarket in the Netherlands (http://albert.nl) is based on an adapter-based architecture. The architecture enabled us to cut costs of development, maintenance and administration tremendously.

I'm still used to applying layering to achieve a high-level separation of concerns. In my view, it's useful to distinguish between infrastructural layering and data access layering, because they imply different sets of abstraction levels. I wrote an article on the subject, meant for the Java community. I've noticed already that it's going to be hard to convince J(2)EE-minded developers.

I'd like to know your thoughts on the subject. If you find the article interesting you can link to it at: http://ijsberg.org/documents/PESA_two_dimensional_layering.pdf

Best regards,
Jeroen Meetsma

Partner IJsberg ICT Architects
http://ijsberg.org
-by Jeroen Meetsma on 9/17/2010 at 1:27 PM

> *Hi, Jeroen, nice article indeed, careful development. Thanks for the note – it's good to see all the variants people derive on their own that are similar. I'll point people to your article. Cheers – Alistair.*

This reminds me of classic UNIX software design. For example, one writes a core engine that reads from a command stream and writes to a result stream (could be pipes, console, sockets, files, etc.). The command line is a light wrapper over this. GUIs connect over sockets. The engine has little UI concern, the UIs focus on the user.

In what ways does hexagonal differ from this design?

-by B. K. Oxley (binkley) on 1/24/2012 at 11:50 AM

> *Unix stdi/o is one implementation of the left-hand side of the standard architectural drawing (user side). There are others (MVC, APIs, web-apps). (2) Unix stdi/o does not cover the infrastructure side of the architecture (db, network, etc). Hexagonal architecture requires both. In principle, hexagonal architecture doesn't have a left side and a right side, since each facet is only a port; just by habit we tend to draw the driver ports on the left and the infrastructure or service ports on the right. cheers.*

> I think it is similar to Model View Presenter Pattern in Dolphin Smalltalk where Model and View intercomunicate via the Presenter in a Mediator pattern. The Presentar was the Adapter. An extension to make the Model intercomunicate with a Persistence layer via another Adapter keep the Model in the center of the hexagon. Other concerns should be added as Authenticatin and more. IMHO. What do you think?

-by Francisco Ary Martins on 4/3/2012 at 4:20 PM

I think it is similar to Model View Presenter Pattern in Dolphin Smalltalk where Model and View intercomunicate via the Presenter in a Mediator pattern. The Presentar was the Adapter. An extension to make the Model intercomunicate with a Persistence layer via another Adapter keep the Model in the center of the hexagon. Other concerns should be added as Authenticatin and more. IMHO. What do you think?

-by Francisco Ary Martins on 4/3/2012 at 4:20 PM

> *Same reply as the one just above to B.K.Oxley*

-by Alistair on 4/3/2012 at 7:42 PM

Alistair,

I'm currently engaged in a discussion about how the above does, or does not, relate to Service Oriented Architecture.

Is the following an accurate paraphrase of your concept? (i) a port is a purpose-based "window" into the core of an application (ii) over each port there sits one or more adapters, to

adapt the port to the needs of external consumers (be they humans, other apps, and so on).

If that's right, let's imagine a case where there are two consumers of an app: (i) a UI, which is "the" UI for the app, and (ii) "other apps in the business", which will consume it using SOA-style services. We are trying to decide between these two alternatives:

Option A: both external interfaces connect to a common Port. The Port itself is NOT an SOA service, its just an API written in an OO language.
So we get:
UI => Port => App
SOAP Service => Port => App

Option B: the SOAP service IS the Port, and the UI connects to it.
So we get:
UI => SOAP Service => App
SOAP Service => App

It's my impression that the former comes more naturally to some agile teams, particularly if there is no immediate need for the SOA Service, but rather it is something that might or will be needed "one day". The first option also seems, to me, to be more in the spirit of the Hexagonal Architecture. However, the latter seems to be the default choice for many SOA practitioners.

Do you see any strong reasons to choose one over the other?

-by John Rusk on 5/15/2012 at 10:06 PM

> *My view is that if no code is needed for the SOA adapter, then the port = the API = the SOAP service. I don't know enough to know if code is needed between the API and the externally visible SOAP service. i have not seen your option B implemented: I initially imagine that would not be practical in real systems, but allow someone to show me otherwise by describing a real system that does that. Alistair*

-by Alistair on 5/18/2012 at 10:56 AM

Thanks for your reply Alistair. I suspect that in many cases, option B is in fact done by combining the code for the port, plus the code for the SOA adapter, into *one* thing called "the Service". In my current project, I think we'll keep them separate and go with Option A.

-by John Rusk on 5/21/2012 at 12:17 AM

Thanks for developing this architecture, Alistair. It seems to match our needs for a better testable and more flexible framework structure just fine.

In order to get it accepted within our company, and to extend my collection of software architectures, I wrote a compact description for it:

http://www.dossier-andreas.net/software_architecture/ports_and_adapters.html

I would like to know if you think it does the architecture justice. I emphasised the distinction between primary and secondary ports/adapters, since I think it is very important for the right implementation.

greetings!

Patrick

-by Patrick van Bergen on 6/27/2012 at 3:35 AM

> *thanks, Patrick. well done with the URL :). I also add here the two other articles you reference:* The birthday greetings kata by Matteo Vaccariand Visualising Test Terminology by Nat Price

I like in particular this bit from Matteo's writeup:

> The traditional three-layers architecture has many drawbacks.
>
> > It assumes that an application communicates with only two external systems, the user (through the user interface), and the database. Real applications often have more external systems to deal with than that; for instance, input could come from a messaging queue; data could come from more than one database and the file system. Other systems could be involved, such as a credit card payment service.
> >
> > It links domain code to the persistence layer in a way that makes external APIs pollute domain logic. References to JDBC, SQL or object-relational mapping frameworks APIs creep into the domain logic.
> >
> > It makes it difficult to test domaain logic without invoving the database; that is, it makes it difficult to write unit tests for the domain logic, which is where unit tests should be more useful.

see also http://www.duncannisbet.co.uk/hexagonal-architecture-for-testers-part-1 by Duncan Nisbet

-by Alistair on 8/15/2012 at 9:39 AM

"A bin of ports" may be the accurate description for an existing "system". But an architect should know better and aim for a different pattern, like Bridge.

-by Mark on 11/27/2012 at 4:37 AM

Thanks, Alistair!

Popped in to look for inspiration and, as per, found just what I was looking for! Share the fascination with hexagonal structures in my own "busy world," but wasn't sure how to apply it. Will add this page to my links!

Domo!

-by RMullen on 7/3/2013 at 12:58 PM

> *hi, back! and best wishes (from Riga, at the moment)*

-by Alistair on 7/3/2013 at 3:24 PM

Nice article; thanks.

Near the beginning you say "As events arrive from the outside world at a port, a technology-specific adapter converts it into a usable procedure call or message and passes it to the application." That makes it sound like an event from the outside world first hits a port and is then processed by an adapter.

But, if I understand correctly, an event from the outside world is processed by an adapter and then passed (in a new form) to a port.

Am I understanding correctly? The ports are the edges of the inner hexagon, right?

-by Simon Katz on 7/23/2013 at 12:24 PM

I have been pushing a similar architecture for 20+ years, mostly focusing on providing the ability to support automated testing. (My thesis was on component based programming.)

I find your explanation more approachable and concise than what is written about most software architecture patterns. I really like the hexagonal architecture name, although I suspect that ports and adapters might be more descriptive and get more traction.

Do you know of any current work on the topic?
Are there any documented examples of real systems implemented this way?

Is the persistence port really viable in real-world systems? Most of the systems that I have seen with a database for persistance have a significant amount of DB knowledge in the domain layer in order to fix performance problems. Maybe that is really an object – relational mismatch?

-by Julie Jones on 8/10/2013 at 11:37 AM

> Julie, try https://twitter.com/search?q=%22hexagonal%20architecture%22 and http://twitter.com/andrzejkrzywda/status/267420878487310336 for starters. The short answer is Yes, maybe, and Yes. Only occasionally.

-by Alistair on 8/10/2013 at 1:24 PM

The one question that escapes me about ports-and-adapters — other renditions are the "onion architecture," and Bob Martin's "Clean Architecture," which I have also been studying — is the primary port for a rich javascript/html5 front-end. I'm interested in your thinking.

If such a front-end exists, then necessarily much UI logic resides on it. E.g., a keyboard interface — like the IPad virtual keyboard — encapsulates much logic that's not really relevant to the application, as such.

So is the primary port strictly entail the request/response payloads transmitted to the server via http? Or via a rest API?

-by wil.pannell on 9/5/2013 at 4:48 PM

Good question, Wil, and honestly I don't know. I'll be implementing something in that direction myself later this year, so I'll get to find out. There's something w rich UI JS front end to be investigated. But even so, there needs to be an API before the mouse clicks – still want the properties of the hexagonal architecture while questioning whre exactly that boundary is.
Alistair

-by Alistair on 9/6/2013 at 5:12 AM

Alistair,

Thanks for your reply. I look forward to your findings.

I've been practicing on the web front-end for a number of years now, most recently in javascript. My approach has been uniformly the same:

separation of concerns using MVP in the spirit of Michael Feathers', "Humble Dialog Box" (objectmentor.com), and

an approach to TDD based on Atomic Objects'. "Presenter First" (atomicobject.com).

I typically end up with (in all generality)

(1) a presenter that, in effect, implements a use case;

(2) a view that's tantamount to a primary port;

(3) html/css that implements the view — and so is tantamount to a primary port adapter;

(4) a dispatcher that's tantamount to a secondary port; and

(5) an instance that implements the dispatcher — and so is tantamount to a secondary port adapter — that issues an http request with a json payload, and asynchronously receives an http response, also packaged as json.

My current thinking is that there is an analogous use case that can be driven test first on the server side, for which the primary port will receive a json payload over http at the server side boundary, and for which the same primary port will package a json payload to return to the client-side in the http response.

This server-side use case may entail considerable work — persistence, messaging, transactions — to be robust.

I continue to be interested in your thoughts.

-by wil.pannell on 9/6/2013 at 5:23 PM

Hello Alister.

I'm quite happy to find out page with this kind of architecture description.
After few small and big projects using J2EE stack I have started private project. Besides others, objective was to develop more flexible architecture. To be honest I'm surprised, the one I came up with, is the same as described here. And has a name :-).

Soon few refactorings will be done.
1. Application core placed in the centre of the Bus, will be moved to dedicated port. It will provide more options to modularize design.
2. Data resource will be moved to dedicated port. To be honest I'm not sure about protocol this port should use.
3. Port will become Java component.
4. Logging will be done using port.

Together with Bus, persistent engine is developed as well, trying to bypass JPA.

Another design decision that was made, is using one central datastructure flowing between ports. It is called ExecutionContext(EC). Motivation behind EC is to be able to dump or browse the context in which business logic is executed.

Besides ports I'm using connectors also. Connector is transport layer abstraction. Its purpose is to extract data from transport layer, initialize EC and send it for further processing. One port can be shared between many connectors. Each connector using same port, has to be able to serialize data from transport layer to one common format. If data coming from client is using different format, port dedicated to that format must be used. Port will handle out data to normalization handler. Data normalization can be tagged also at connector level, but the design would be not as clear.

Code is far from state of the art and API is suboptimal, but all authentication and multitenancy is actually working quite good. I can confirm that compared to traditional J2EE development, I'm not running into any major issues. Architecture is clean, flexible and easy to hack.

Initial implementation was done in Python then rafactored to Java.

Code is accessible at github
/projectscopt/scopt/wiki – is the original Python version

/projectscopt/jacob – Java reimplementation

Regards
Robert

-by Robert on 9/11/2013 at 2:43 PM

In figure 4, there is an app-to-app adapter. How would App1 make a call to App2 without creating some sort of dependency on it? You can provide or recommend an example to illustrate this implementation? Thanks.

-by Bill on 10/21/2013 at 3:39 PM

of course there's a dependency.

-by Alistair on 10/21/2013 at 9:41 PM

Hi Alistair,

I have finished documentation about Kivix. Java application library for distributed computing. Kivix is built using Hexagonal Architecure. Examples can be viewed at:

http://gabert.github.io

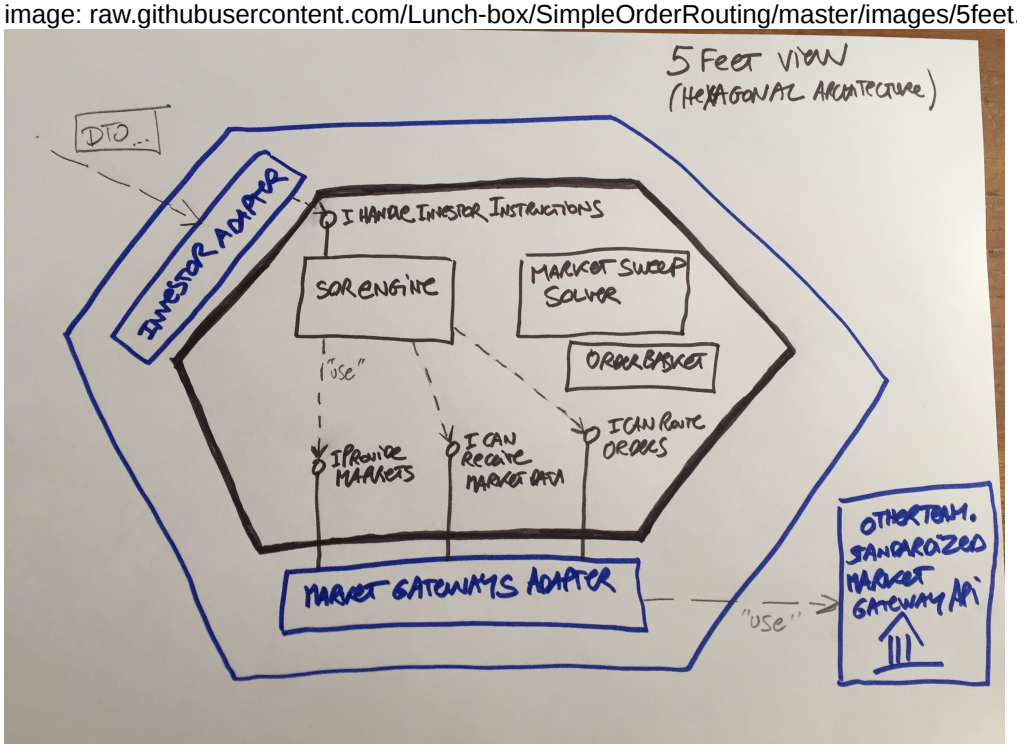
Robert

-by Robert on 12/18/2013 at 3:57 PM

see also https://github.com/Lunch-box/SimpleOrderRouting/wiki/Logbook-7
thanks to Thomas PIERRAIN

-by Alistair on 5/10/2016 at 5:15 AM

see also https://github.com/Lunch-box/SimpleOrderRouting/wiki/Logbook-7
thanks to Thomas PIERRAIN
image: raw.githubusercontent.com/Lunch-box/SimpleOrderRouting/master/images/5feet.jpg



-by Alistair on 5/10/2016 at 5:17 AM

Your post design so old-fashioned and so hard to read, that I used EasyReader extension from Chrome to read this post.

But, the provided Idea of Hexagonal architecture is great and I'm using this approach in all my project.
Sometimes it's needed more effort to isolate dependencies and even framework dependency from core app, but it's worth this efforts in future.

-by Seyfer on 9/22/2016 at 1:19 AM

Hi Alistair,
It is correct to think about the port-adapter architecture for the runtime definition of the domain and its logic starting from artifacts expressed in different languages / DSL? The internal domain would result from the translation of these artifacts. What do you think?
Thanks

-by Igor on 11/25/2016 at 5:32 AM

Hi Alistair, excellent post.
I have a facebook page that tries to diffuse many important posts (like yours or from Martin Fowler) in software engineering. Please, give me your permission to translate your post to Spanish. I'm just a colleague student from Perú.

I'm waiting for your reply. Good luck
Pd: sorry for my bad english :(

-by Arthur on 2/6/2017 at 10:03 PM

> *But of course you have my permission. Let me know the URL when you have it.*
> *(this site won't let you post a link, so just put it in minus the "http://")*

-by Alistair on 2/7/2017 at 8:09 PM

Hi Alistair, I finished to translate the article, it take me some time to translate it because I created a very simple blog for the post & I was a busy (and also the article is complex to translate). Here is the article:
http://academyfor.us/posts/arquitectura-hexagonal
Again thanks for the permission :).
pd: I will be improving it

-by Arthur on 2/22/2017 at 6:25 PM

Hello Alistair,

I wanted to ask you how do you recommend or prefer to implement hexagonal architecture, just in two modules ("layers"):

-The hexagon
-The out of the hexagon with all adpaters here.

(Where the apapters module depends on hexagon)

Or

-The hexagon
-One module per adapter, where each adapter depends on the port of the hexagon that it implements.

From your explanation of the architecture, I consider more correct the second one (that's the way I implement my projects), but almost every example I see over the internet is implemented like the first one, which I see more like an "onion" architecture than ports and adapters.

Im I wrong?

Thank you very much.

-by Garrigher on 9/15/2017 at 12:09 AM

Hello Alistair,

I wanted to ask you how do you recommend or prefer to implement hexagonal architecture, just in two modules ("layers"):

-The hexagon
-The out of the hexagon with all adpaters here.

(Where the apapters module depends on hexagon)

Or

-The hexagon
-One module per adapter, where each adapter depends on the port of the hexagon that it implements.

From your explanation of the architecture, I consider more correct the second one (that's the way I implement my projects), but almost every example I see over the internet is implemented like the first one, which I see more like an "onion" architecture than ports and adapters.

Im I wrong?

Thank you very much.

-by Garrigher on 9/15/2017 at 12:21 AM

Display Name

Email (not displayed)

Comment

NOTE: Comments containing html hyperlinks will not be saved.

Verification

Remember Me ☐