Architecture Patterns

(Beyond REST)

Download and Read Along

http://bit.ly/nygard-ldn-2017-cards

Big Ball of Mud

Context

- Big, ugly systems emerge from throwaway code
- Well-defined architectures subject to structural erosion

Discussion

"Shantytowns are squalid, sprawling slums. Everyone seems to agree they are a bad idea, but forces conspire to promote their emergence anyway. What is it that they are doing right?

Shantytowns are usually built from common, inexpensive materials and simple tools. Shantytowns can be built using relatively unskilled labor. Even though the labor force is "unskilled" in the customary sense, the construction and maintenance of this sort of housing can be quite labor intensive. There is little specialization. Each housing unit is constructed and maintained primarily by its inhabitants, and each inhabitant must be a jack of all the necessary trades."

– Brian Foote and Joseph Yoder, Pattern Languages of Program Design 4

Forces

- Time - insufficient
- Cost - pressure to minimize
- Experience & Skill - may be insufficient
- Visibility - problems can grow unseen
- Accidental complexity
- Frequent change without refactoring

You need to deliver quality software on time, and under budget.

Therefore, focus first on features and functionality, then focus on architecture and performance.

http://www.laputan.org/mud/mud.html#BigBallOfMud

See also: "Worse is Better", Richard Gabriel
Aliases: Shantytown, Spaghetti Code

Question: Is this just cynicism? Is Big Ball of Mud an antipattern?

Blackboard

Context

An immature domain in which no closed approach to a solution is known or feasible.

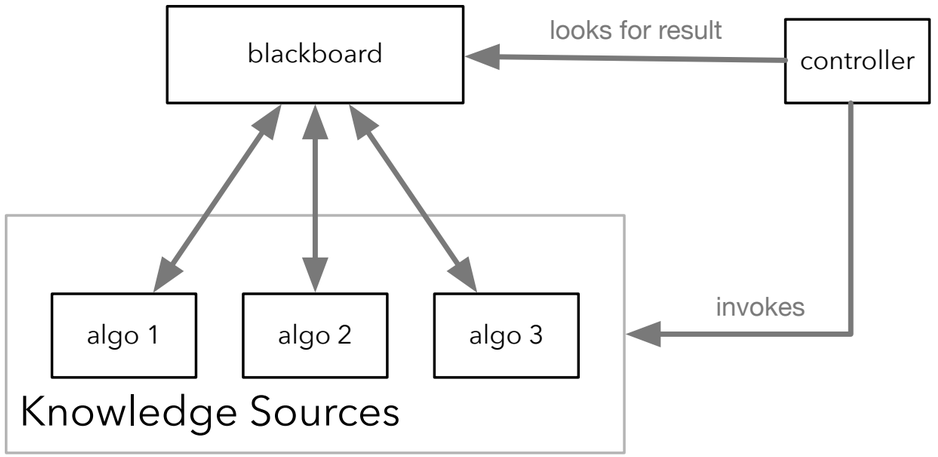
Solution

Independent programs cooperate indirectly by taking data from or placing data into a shared repository, called the *blackboard*. A program can start running once the input it needs appears on the blackboard. When it completes, it places its result onto the blackboard.

The processors work on data at different (usually increasing) levels of abstraction.

Forces

- Complete search of solution space is not feasible
- May need to experiment with different algorithms
- Different algorithms partially solve problems.
- Input, intermediate, and final results have different representations.
- An algorithm may use (but not consume) results of other algorithms.
- There exists potential for parallelism.



Question: In what other contexts would you apply the Blackboard pattern?

Broker

Context

Distributed, heterogeneous systems with independent components.

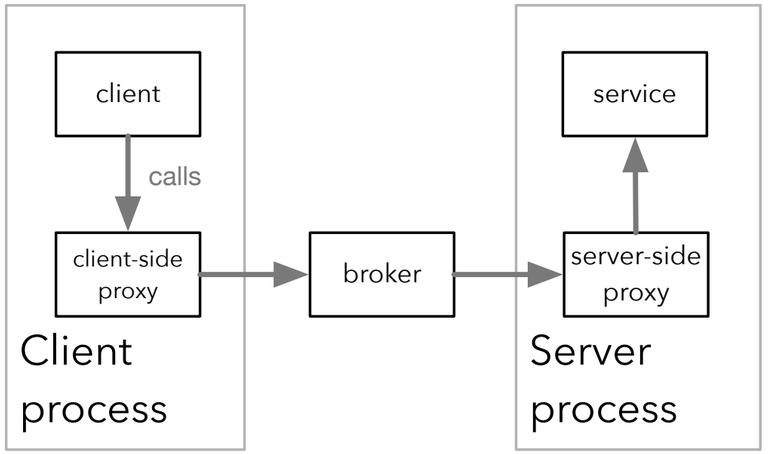
Solution

- Introduce a *broker* between *clients* and *servers*.
- *Services* register themselves with the broker.
- *Proxies* and *bridges* may help cross network types.
- Client proxy presents a function-call interface over remote calls.

See also: Peter Deutsch's "Fallacies of Distributed Computing"

Forces

- Need to communicate across process boundaries.
- Possibly heterogeneous network or transport.
- Desire for transparent remote access.
- Run-time replacement, exchange, or substitution of components.
- Architecture should hide implementation details from users.



Question: Is this an antipattern?

Components and Glue

ancient

Context

Components that can be used and reused in different combinations.

Discussion

Some long-lived systems use this pattern:

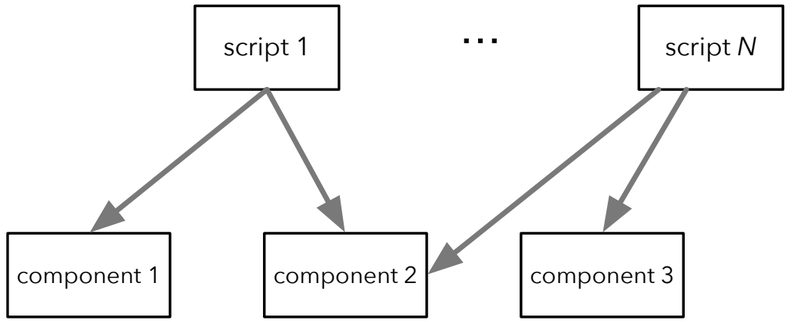
- Visual Basic (VB/VBA and controls)
- Delphi
- Emacs

Some new architectures are instances.

- Node API over microservices
- AWS Lambda

Forces

- Component authors are separate from the assemblers
- End-user programming may be present
- Scripts may appear or disappear at any time
- Components may be substituted or intermediated.
- Components may be written in different languages than the glue.
- Components support dynamic interfaces (usually with introspection or discovery)
- Components are not “aware” of their consumers



Consider: Where do the scripts execute?

Question: How does this differ from a plugin architecture?

Dispatcher/Worker

Context

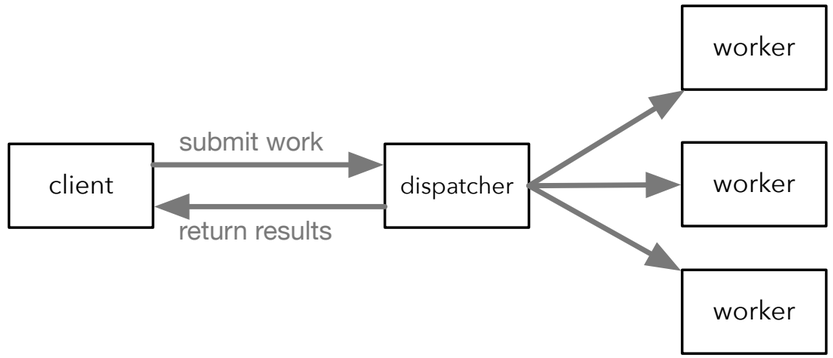
Partitioning work into semantically identical sub-tasks.

Forces

- Clients should not be aware that the whole body of work was subdivided.
- Sub-tasks may need to be coordinated, depending on the algorithm.
- The means of partitioning work and the task granularity should not affect the client or the processors.

Discussion

- The dispatcher may have the same API as the workers, to allow substitution or recursive task breakdown.
- If running in many processes, fault tolerance is a concern.
- Some tasks parallelize easily, others require additional processing to combine the results.
- Workers may be permanent, ephemeral, or drawn from a pool.



Examples: Fork-join, Map/Reduce, GPUs

Aliases: Master/slave, Manager/worker.

Event stream/CQRS

Context

Scalable distributed systems with many writers

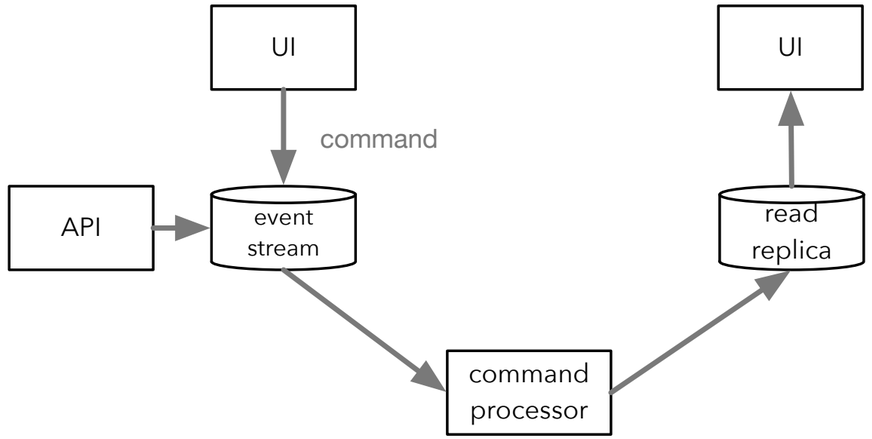
Solution

Segregate operations that read data from operations that update data by using separate interfaces.
Reads and writes may operate on different schemas.
A processor applies writes to the permanent record.
Event streams may be replayed speculatively or as a recovery strategy.
Uniform interface for GUI, admin, and API updates

https://msdn.microsoft.com/en-us/library/dn568103.aspx

Forces

Queries and updates operate on the same entities in the same datastore.
Relational DBMSes exhibit locking and variable performance in these cases.
Read scaling and write scaling are rivalrous
Access control can be difficult if frameworks expose objects with both read and write capabilities



CQRS = “Command-Query Responsibility Separation”

Layers

Context

A large system that requires decomposition.
Mix of high-level and low-level concerns
Several operations are at the same level of abstraction

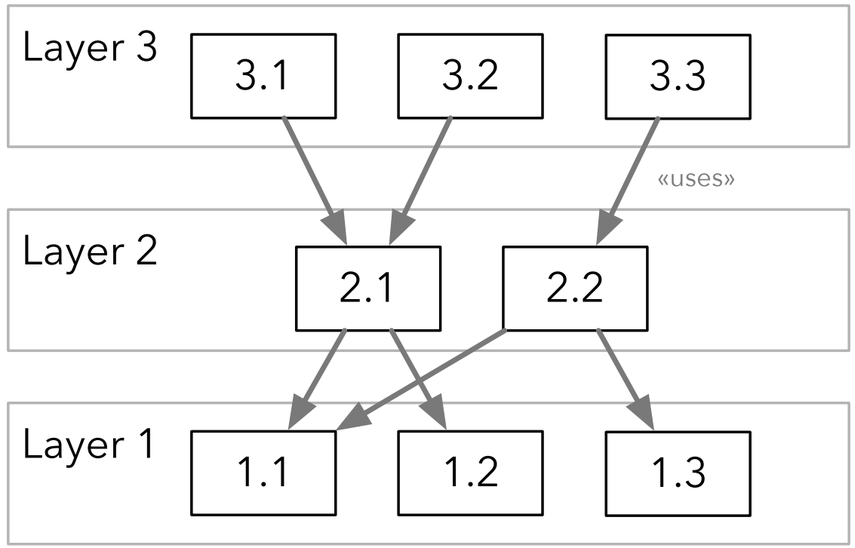
Solution

Therefore, structure your system into an appropriate number of layers and place them “on top” of each other.

– Buschmann, et. al., Pattern Oriented Software Architecture, Vol 1.

Forces

- Late changes should have limited ripple effect
- Interfaces should be stable, may be standardized
- Parts should be interchangeable
- Other systems may reuse lower layers
- Responsibilities should be grouped for comprehension and maintainability
- Further decomposition is needed for team structure and design.



Examples: TCP stack, MVC

Related: Microkernel, Virtual Machine

Question: What makes a good or bad implementation of Layers?

Microkernel

Context

Developing several applications that use similar APIs on top of the same core functionality.

Solution

Encapsulate fundamental platform services in a *microkernel*:

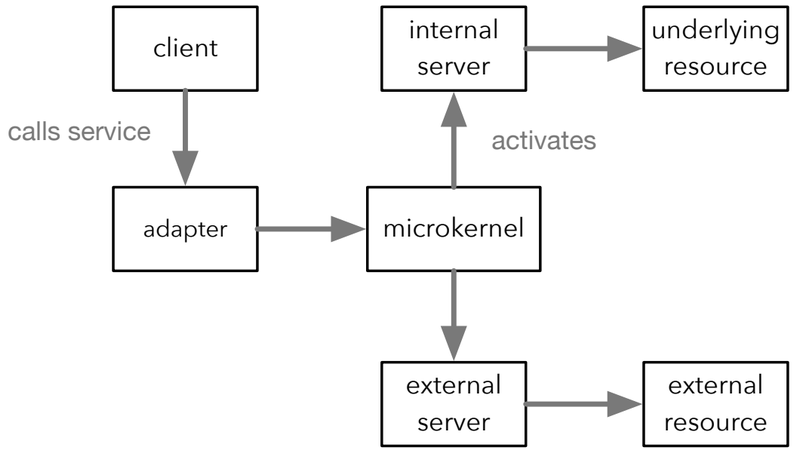
- Mediates communication between services
- Manages systemwide resources
- Offers APIs to access its functionality

Internal servers provide core functions, but outside the process space of the microkernel

External servers provide functionality to clients. Clients locate external servers via the microkernel.

Forces

- Application must cope with continuous hardware and software evolution.
- Applications must support different but similar application platforms.
- Platform should be portable, extensible, and adaptable to allow integration of new technology.



Discussion: What does this look like at datacenter scale?

Pipes and Filters

Context

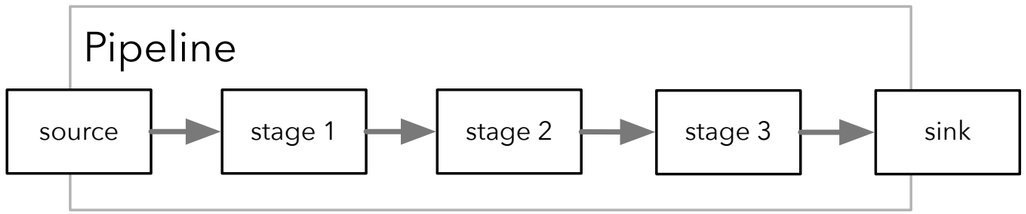
Processing data streams

Solution

Divide task into separate processing stages. Connect them sequentially: the output of one stage is the input to the next. A filter consumes and delivers data **incrementally**. A *data source* supplies the initial input, while a *data sink* consumes the final output. The data source, filters, and data sink are connected via pipes. The assembly is a processing pipeline. Some external entity constructs the pipeline.

Forces

- Data stream processing which naturally subdivides into stages
- May want to recombine stages
- Non-adjacent stages don't share information
- May desire different stages to be on different processors



Example: Unix pipes, Apache Spark

Question: What weaknesses do you see in pipes&filters?
How do you decide between *push* and *pull* data flow?

Proxy

Context

Client needs to access services. Direct access is possible, but not desired.

Solution

Introduce a *proxy* that supports the same interface as the server.
The proxy may enforce access control, caching, or other pre- and post-processing on requests.

Types of proxy:

- Remote proxy
- Protection proxy
- Cache proxy
- Synchronization
- Counting
- Virtual
- Firewall

Forces

- Security requirements need enforcement
- Performance under direct access is insufficient
- Access to the provider should be transparent
- Remote communication may or may not be involved

Question: How does this differ from broker?