Distributed Benchmark

Software Design Case Study

http://sourceforge.net/projects/balancerbench/

© 2006 Anton Khritankov
ahritankov@hotmail.ru

Annotation

This document is an attempt to reconstruct how design of Distributed Benchmark was made. The document might be useful to developers who wish to extend or port Distributed Benchmark to another platform and to anyone interested in software design.

Distribution

The Problem

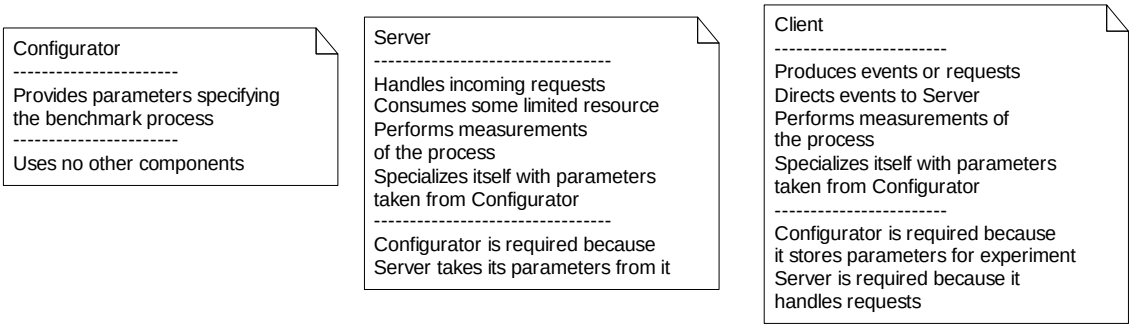
During my study at MIPT and more precisely at ISA RAS with IARnet project, there arose a problem in distributing tasks between processing units but there were no adequate knowledge about applicability of different load balancing techniques available at that moment. Therefore, it was decided to develop a system that would allow us to gather necessary data and make a justified decision what algorithms of load distribution we should employ in IARnet. The system would also be able to become a testing platform for new methods of load balancing, as there were several ones already.

The Idea

The basic idea of the system formed into the following general requirements:

- The system should provide means to perform measurements of different aspects of performance of balancing techniques in automatic mode
- The system should be portable across different message passing technologies such as CORBA, HTTP and others
- Algorithms used for producing requests and their handling should be configurable and modifiable
- (the list is not yet completed)
- ….

First ideas about how the system should be have driven us to the following basic scheme with three major components

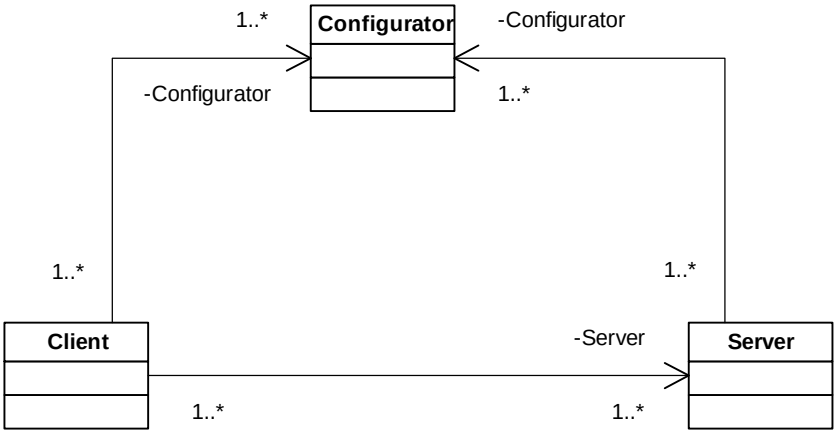


Roles of Client and Server are quite simple:

- Client creates a flow of requests and directs them to the Server
- Server receives events then performs some work over them, consuming some limited resources, and reporting back to the Client, if necessary

The role of Configurator may seem less obvious. Configurator is responsible for specifying measurement process in terms of providing parameters for algorithms of Client and Server. Configurator also facilitates automatic operation of the system.

Therefore, there are three entities in the system that has been identified so far. Their interaction occur as shown in the following diagram



As it was shown above, there should be at least one component of each type for process to begin. Client knows about Configurator and sends some sort of message to Configurator, which returns with parameters for Client. For example, this might be a remote procedure call. Client also sends messages containing requests to Server, which is also available to Client. Server uses only Configurator and do not have any knowledge about Client except that it produces requests that Server receives.

What about measurements and other data collected?

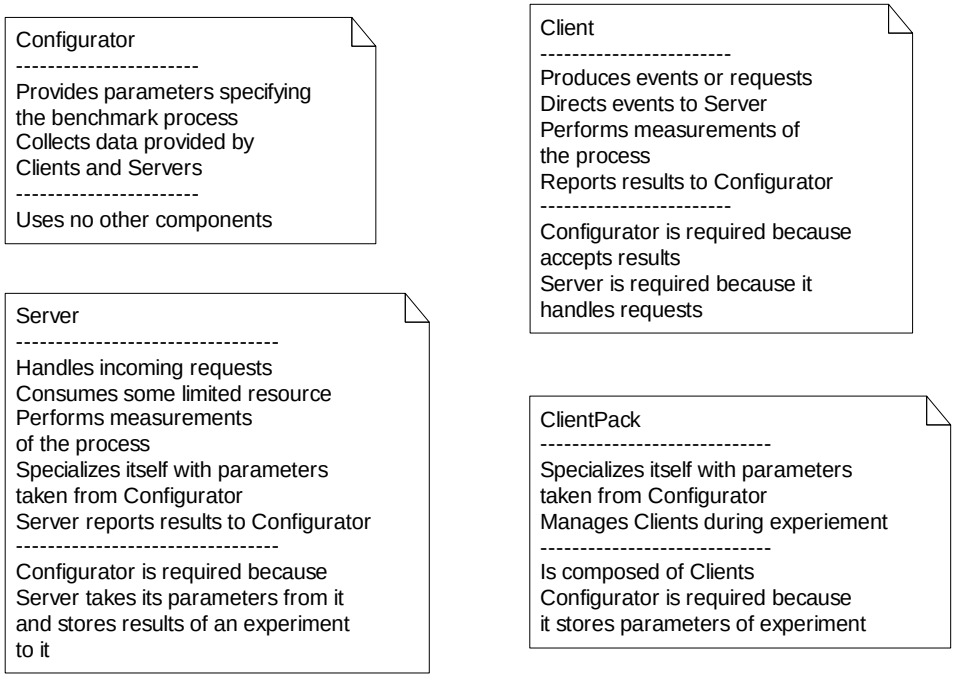
Defining responsibilities of components, we have missed the fact that there are some data collected during an experiment that is required to be stored persistently and analyzed later. Because the Configurators' responsibility is to maintain information about an experiment it was considered that Configurator should also be responsible for collecting results of experiment from Clients and Servers and stores them locally.

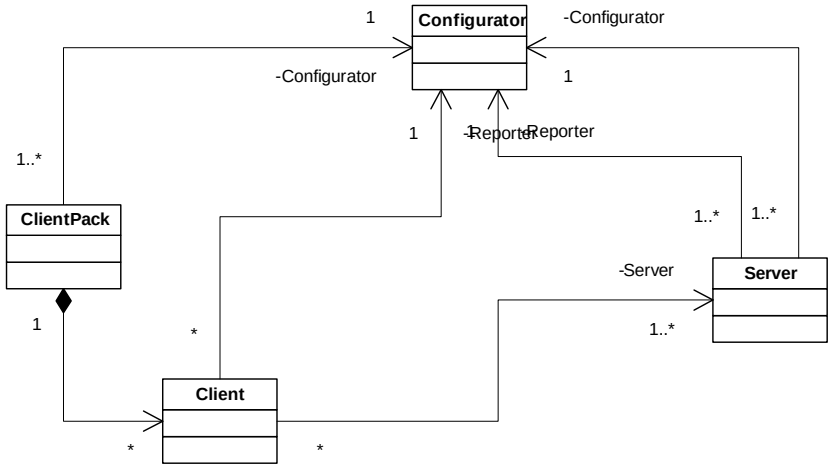
There are three significant changes at this step:

- Client was decomposed into ClientPack that is responsible for retrieving parameters from Configurator and Client itself which is responsible for production of requests

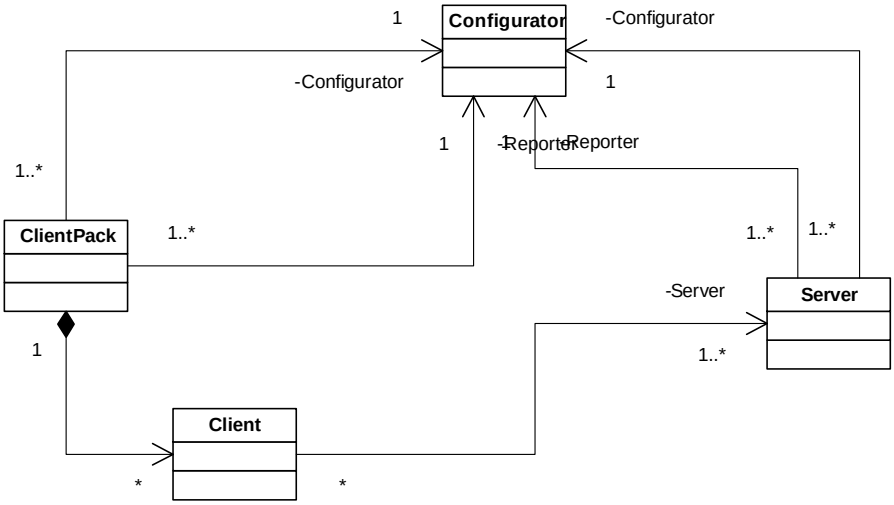
- Configurator now acts in two roles – it supplies parameters and stores results
- It was considered that in most cases only one Configurator is sufficient for each run of experiment.

ClientPack retrieves parameters from Configurator. In general, there can be an arbitrary number of Clients, which is defined by parameters taken from Configurator, and usually there are a lot of Clients and only a few Servers, so there was a disproportion and some "ugliness" in requiring user to manage each Client at the same level of detail as a Server. We can avoid these difficulties by introducing a ClientPack and by managing Clients in groups where each group is controlled by a ClientPack. Another responsibility of ClientPack is to produce required quantity of Clients that actually perform measurements and control them during the experiment. Originally ClientPack was not required to send results of experiments back to the Configurator because this remained Clients' responsibility.





Another option might be to make ClientPack responsible for communication with Configurator while Client communicates only with Server. This way the system should look like



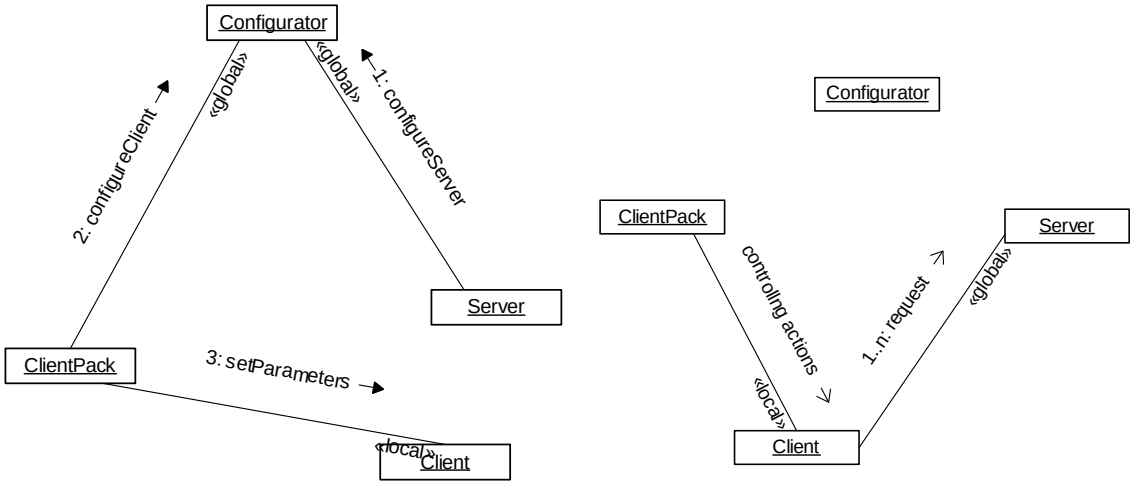
It was a really a matter of a coin flip that turned design process to the first option.

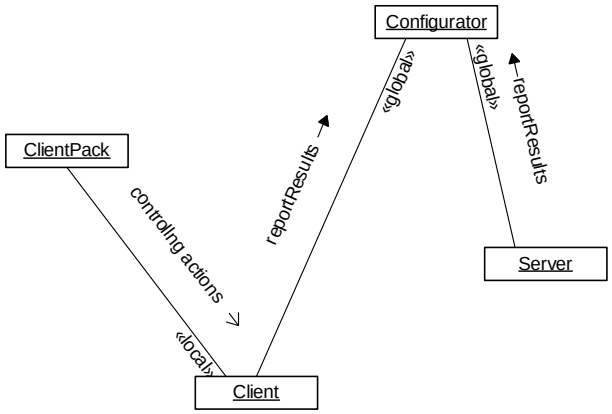
Dynamic Viewpoint

Let us consider the system from dynamic point of view. In general, an experiment could be represented as a sequence of three steps:

- Prepare an experiment
- Conduct the experiment making measurement
- Collect and store results

During preparation of an experiment, Client and Server components take their parameters from Configurator and specialize themselves as shown at the first to the left picture below





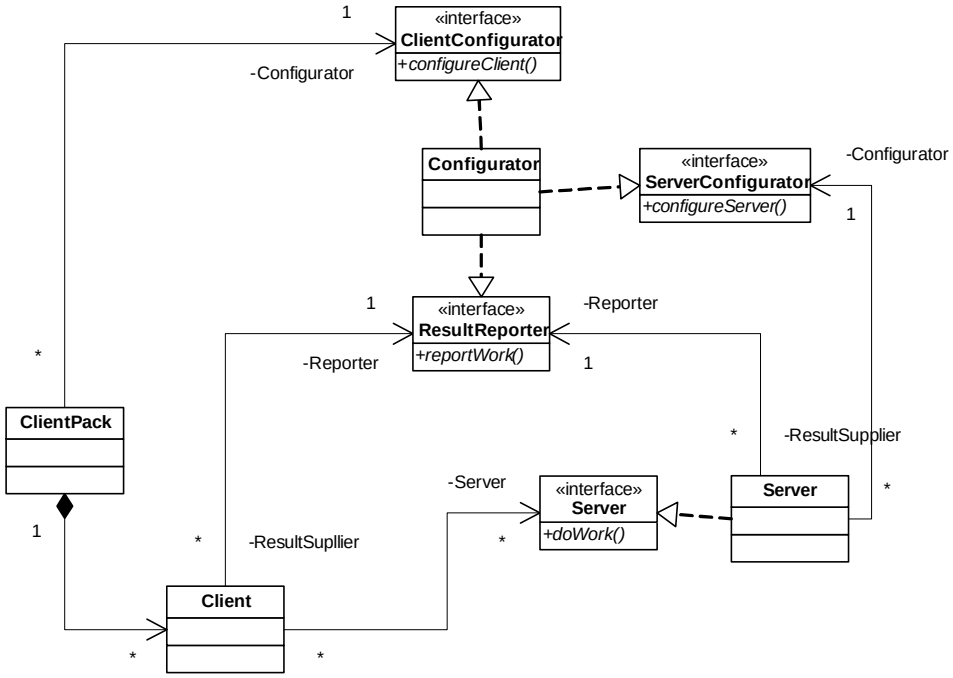
Note that in general Configurator might be located at different host, so interactions with it are "global". ClientPack and Client should be "local" to each other because ClientPack is responsible for creation of Clients and controlling their execution. The next phase is when Client sends requests to Server, which is also "global" while being managed by ClientPack. Configurator takes no part in experiment itself. The last phase is collecting the results. Client and Server report data gathered during the experiment to Configurator, Client being controlled by a ClientPack.

Second requirement in action

Following the ISP (Interface Segregation Principle), we should define a distinct interface for each role for Configurator and Server. This principle says that each user type should have its own interface. In our case, we have three users or roles: a client, a server and a result reporter (which can be both client and server); therefore, there should be three interfaces implemented by Configurator:

- An interface for clients – ClientConfigurator, through which Client communicates with Configurator in order to retrieve parameters for experiment
- An interface for servers – ServerConfigurator, similar to ClientConfigurator but used by Server
- An interface for result reporters – ResultReporter, a unified interface for reporting results of experiments, such as performance measures

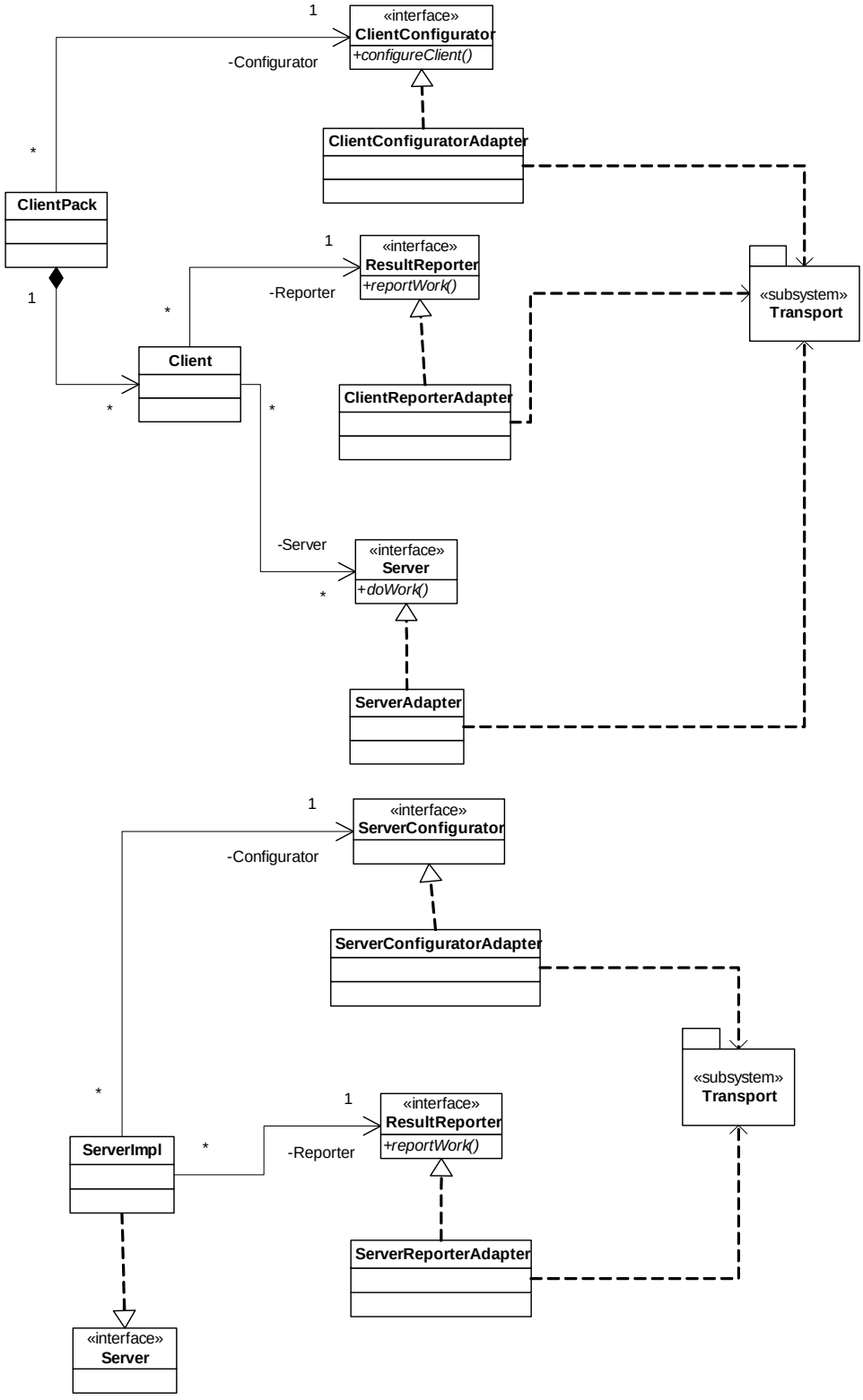
Therefore, we should have something like



If we wish to achieve independency from transport subsystem, we have to develop a way to isolate it providing generic interface that is specific to object of interaction itself and is not affected by message passing issues. There is a special pattern for this case – the *Adapter* pattern, which in our case will connect an interface, visible to a component of the system to an internal interface used by Transport. (if we abstract from presence of Transport subsystem it also will be a *Remote Proxy*)

Because Client uses ClientConfigurator, ResultReporter and Server there are three adapters required for Client and ClientPack.

Server uses two interfaces and there are corresponding adapters for Server.



Configurator does not use any other components thus there are no adapters required for it.

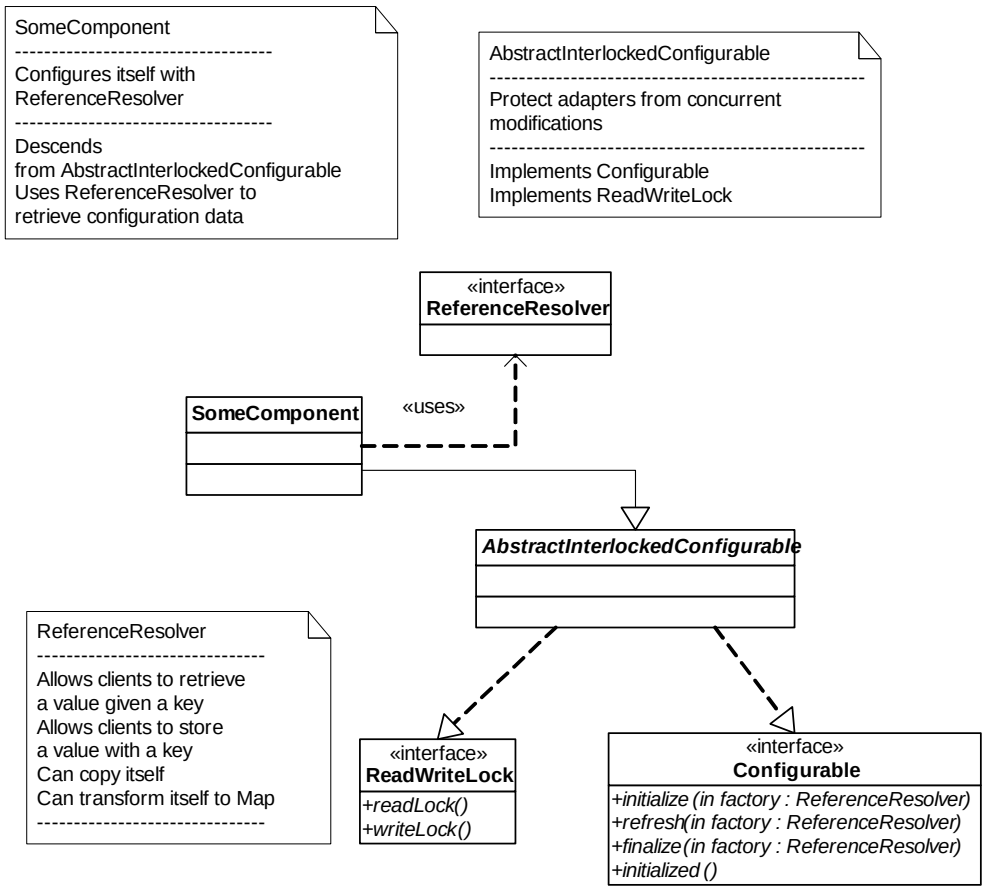
Who will manage so many adapters?

That might pose a problem, as we want our adapters to be configurable in order to dynamically construct the system. We also want our components to be easily configurable. It was a bit of opportunistic research to apply not a standard *Abstract Factory* pattern but a different approach

where families of classes implementing abstract interfaces are not strictly defined and even can be configured without rebuilding the system. Configuration issues are solved this way:

- At first, each object that needs to be configured dynamically was forced to implement Configurable interface
- ReferenceResolver interface was developed to provide a “context” or environment for configurable objects.
- For the sake of safe concurrency a new class AbstractInterlockedAdapter, which solves issues concerned with concurrent modification and reconfiguration, was introduced
- After that, each object that will ever operate in concurrent environment was made descendant of AbstractInterlockedAdapter

Brought together, this looks like:



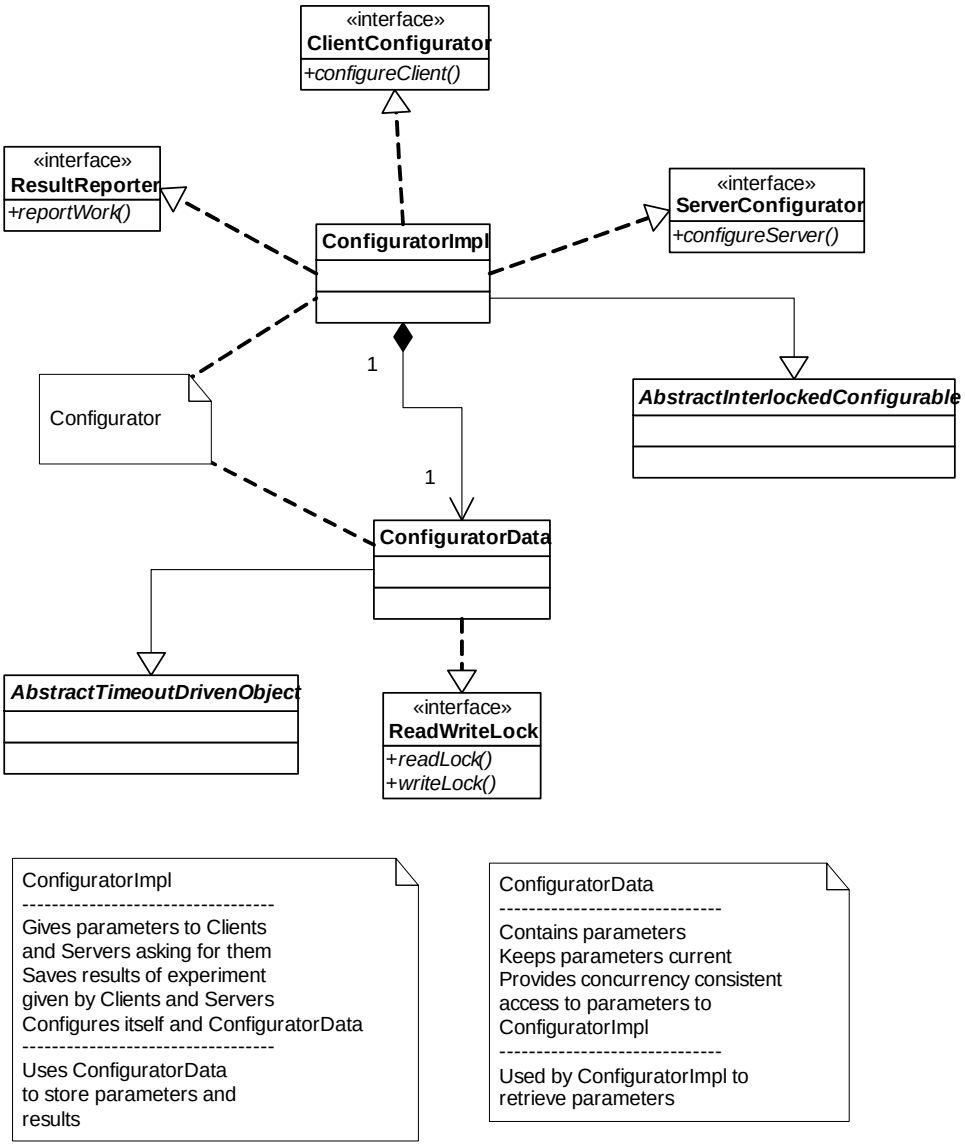
What classes do we want to be dynamically configurable? At first, of course, the adapters, as they may depend upon transport specific issues such as addresses of remote objects to which they should provide access. Second, Client should be dynamically configured, as it might wish to use not compile-time predefined algorithms but algorithms specified by Configurator and retrieve adapters from ReferenceResolver. Server uses dynamic configuration to retrieve parameters from Configurator during startup. Configurator uses dynamic configuration in order to manage initialization of parameters of experiment and load them into memory.

Configurator revisited

Before continuing with configuration of components, let us consider structure of Configurator, as it will affect how configuration of Configurator is implemented. Currently Configurator plays two roles:

- Configurator provides parameters to Client and Server and accepts results from them
- Configurator stores parameters and results and keeps them current

The last task should be done periodically while the first one is reactive and therefore executed upon request. This caused Configurator to be split into two parts with responsibilities as shown at the picture below



We will talk about AbstractTimeoutDrivenObject a little later; yet at this moment, it is sufficient to know that it allows its descendant to receive timed notifications. ConfiguratorData became event driven and now supports concurrent access due to implementation and use of ReadWriteLock. ConfiguratorImpl was made Configurable through extending AbstractInterlockedConfigurable. Diagram shows that ConfiguratorImpl and ConfiguratorData are parts of Configurator where ConfiguratorImpl contains ConfiguratorData and manages its

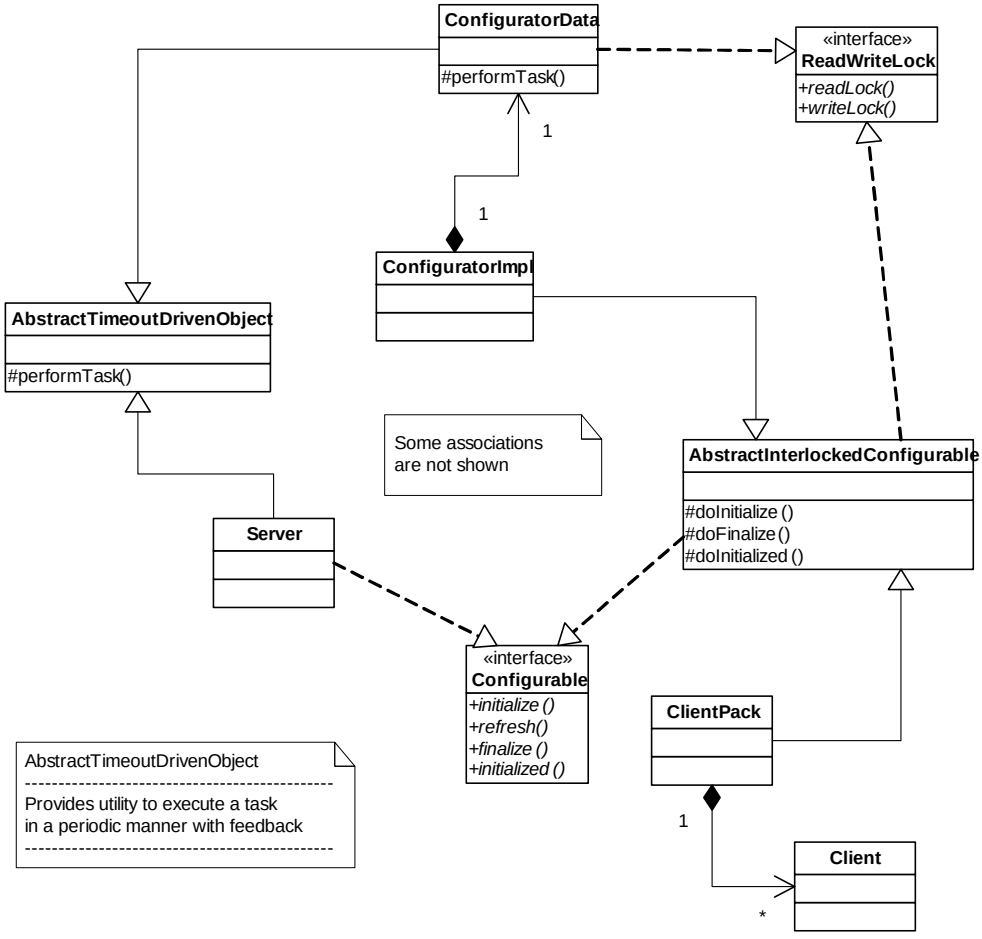
configuration. Locks in ConfiguratorImpl and ConfiguratorData do not intersect because the first one is active while configuring at startup or reconfiguring in the meantime between experiments and the other is active only during normal execution.

How to determine that experiment has already finished?

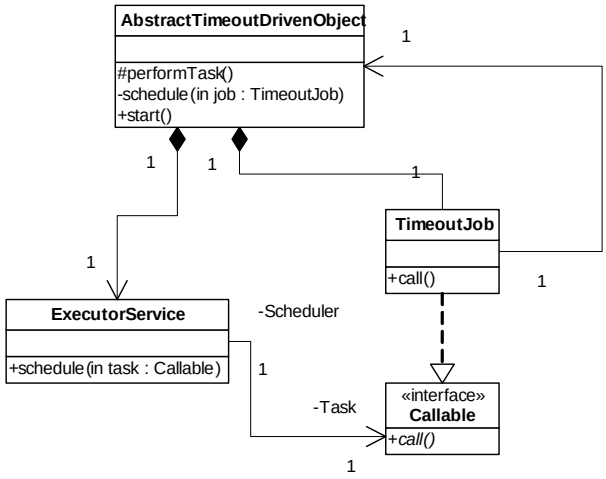
The question is important because the system aims at supporting scheduled runs in automatic mode. For Client the answer is that it has completed executing the algorithm and therefore knows that for the instance of Client the experiment has finished. For Configurator the question itself is illegal because there are no bounds of experiment for Configurator. Server cannot itself determine when experiment finishes, i.e. it cannot figure out when a delay has been due to a configured pause in Client or the delay means Client has finished. The first solution came into mind was to introduce a timeout for Server. If timeout expires after a request handled then Server considers experiment finished. This solution implies that Server is timeout driven.

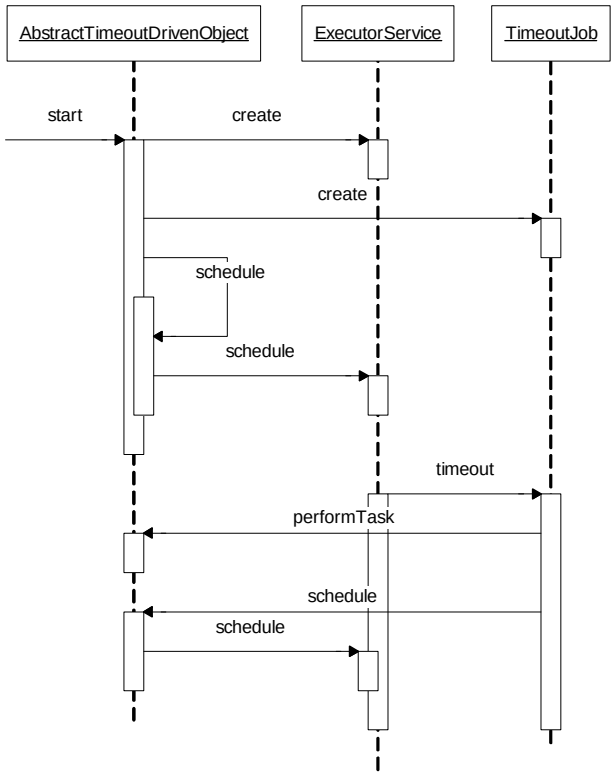
Timeouts

Now we take a closer look at how components are configured and how they respond to timeouts. Very general scheme is presented at the diagram below. What could be seen from it?



Well, the diagram tells us, that there are Server, ClientPack and ConfiguratorImpl that are all Configurable. It also says that Server and ConfiguratorData can respond to timeout events. AbstractTimeoutDrivenObject defines a *Template Method* that uses performTask abstract protected method that should be implemented in a descendant class to perform actions required at timeout event arrival. Current implementation of AbstractTimeoutDrivenObject uses a ScheduledExecutorService that calls inner class, which in turn calls performTask defined in AbstractTimeoutDrivenObject. The structure of AbstractTimeoutDrivenObject is presented at the diagram.



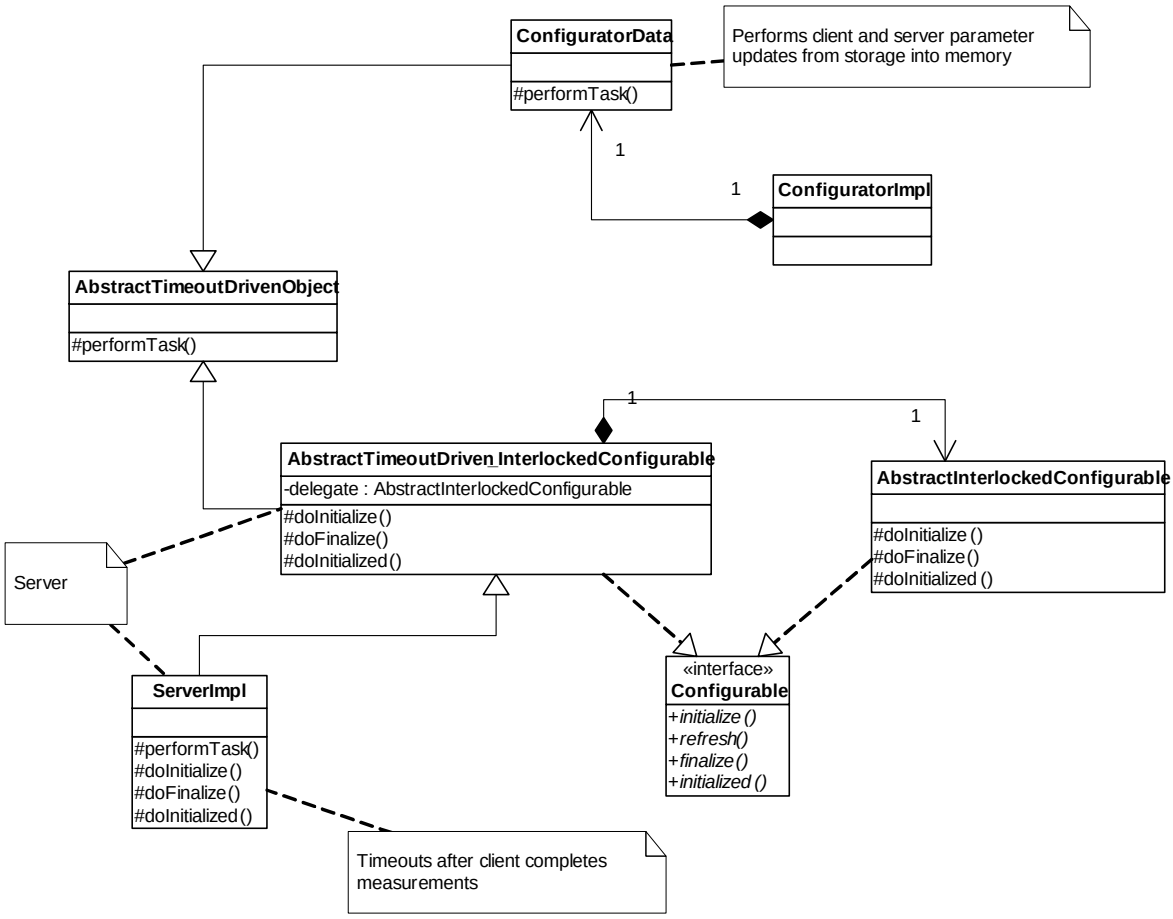


From dynamic point of view when somebody calls start on AbstractTimeoutDrivenObject it triggers re-creation of ExecutorService and TimeoutJob. Then AbstractTimeoutDrivenObject schedules TimeoutJob with zero delay this leads to immediate invocation of TimeoutJob that calls performTask that returns next value for the delay, which is set by TimeoutJob before going to sleep. AbstractTimeoutDrivenObject waits until TimeoutJob completes its first invocation and then returns to caller. When timeout occurs, ExecutorService calls TimeoutJob that again reschedules itself at the delay returned by performTask in case there are no exceptions caught.

Server implementation at first glance

As we have seen, Server should be both Configurable and respond to timeout events. In order to avoid multiple inheritance we apply a *Delegate* pattern. That is AbstractTimeoutDriven_ InterlockedConfigurable extends AbstractTimeoutDrivenObject and delegates functionality of Configurable to its private part AbstractInterlockedConfigurable.

Server extends resulting class thus achieving necessary behavior.



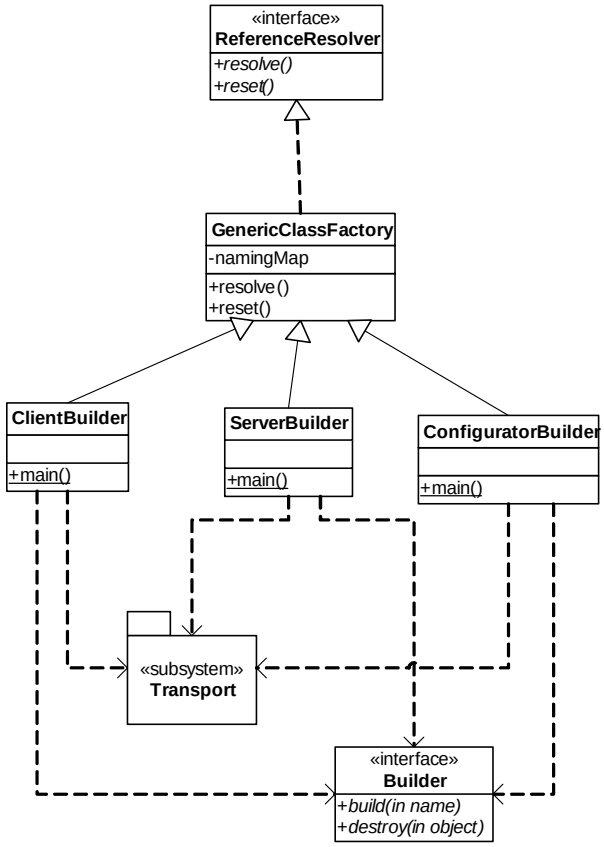
This completes a brief description of Server and Configurator details of implementation.

Configuring and Building

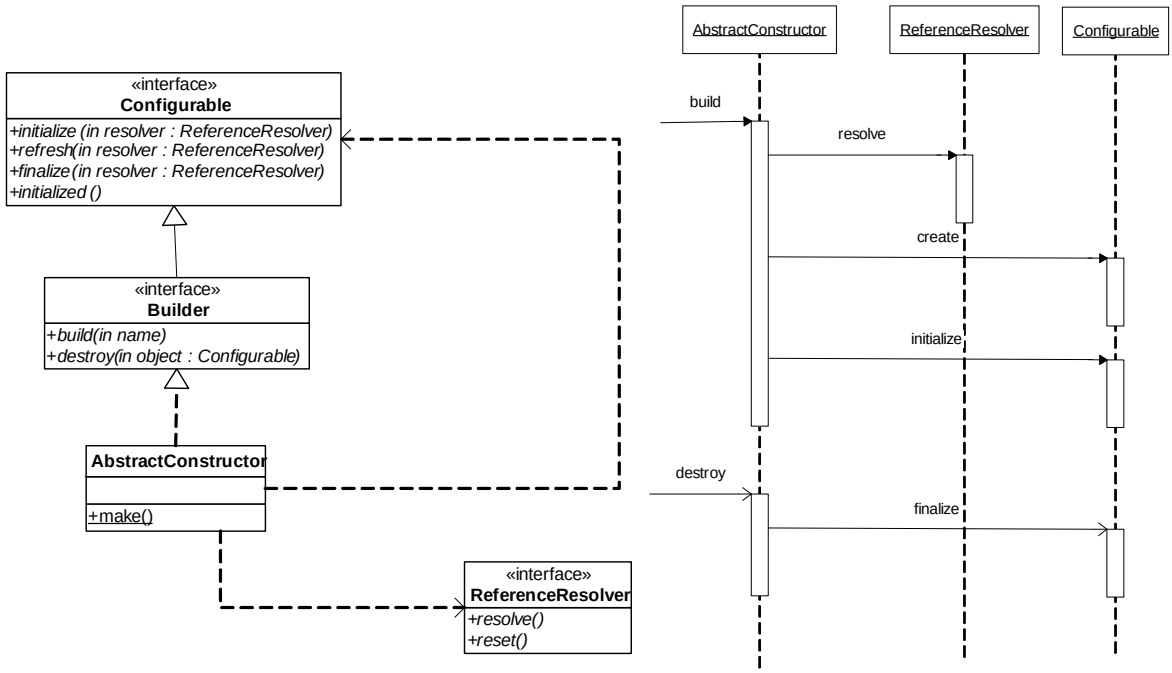
Remember that all Configurable objects use ReferenceResolver to determine parameters and resolve other objects they need. In general, there expected to exist three independent and separately hosted parts of the system: Client, Server and Configurator. Each of them should define its specific instance of ReferenceResolver, albeit there are common features defined by ReferenceResolver contract in all these implementations:

- ReferenceResolver should store and allow to change key-value pairs describing configuration of a component or components
- ReferenceResolver should allow copying and converting to a common class providing similar functionality similar to Map

Moreover, an instance of ReferenceResolver defines the context for the application through which different components can communicate and know about each other. In order to define such an instance for each application a builder object was introduced, which class extends GenericClassFactory and declares main(). In main() each class creates an instance of itself which is used as ReferenceResolver for the application. Each builder object is automatically singleton because it is an entry point for an application. GenericClassFactory serves as a default implementation of features described previously. Builders may use a Builder interface for constructing objects in context defined by ReferenceResolver.



Each implementation of Builder interface is responsible for creation of an object given its name and performing initialization of the object and finalization of that object, thus every object to be created by Builder should implement Configurable.



The diagrams presented above describe the *Builder* pattern where common mechanism applied for constructing objects is initialization and destruction. AbstractConstructor uses

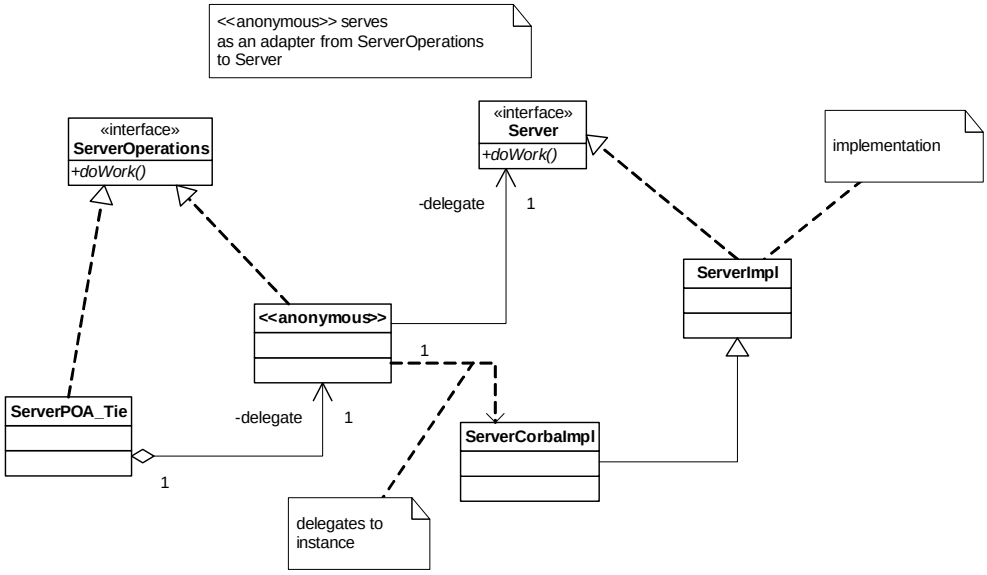
ReferenceExplorer to resolve names of objects it receives from clients to class names. The diagram to the right explains dynamic details of the solution.

Server Implementation for CORBA

This chapter covers existing implementation of Server for CORBA. Main issues connected with CORBA implementation that need to be solved are:

- Translation from CORBA interface of the object to generic interface
- Server initialization and finalization for CORBA.

The first issue was addressed by applying *Adapter* pattern with introduction of ServerPOA_Tie class that delegates methods of ServerOperations through anonymous class to Server interface backed by ServerCorbaImpl class. The second issue was solved by separating common CORBA initialization tasks executed in ServerCorbaBuilder and server object specific issues that were solved in SorveCorbaImpl from algorithm defined in ServerImpl. Note that ServerCorbaBuilder defines main() thus being application entry point for Server.

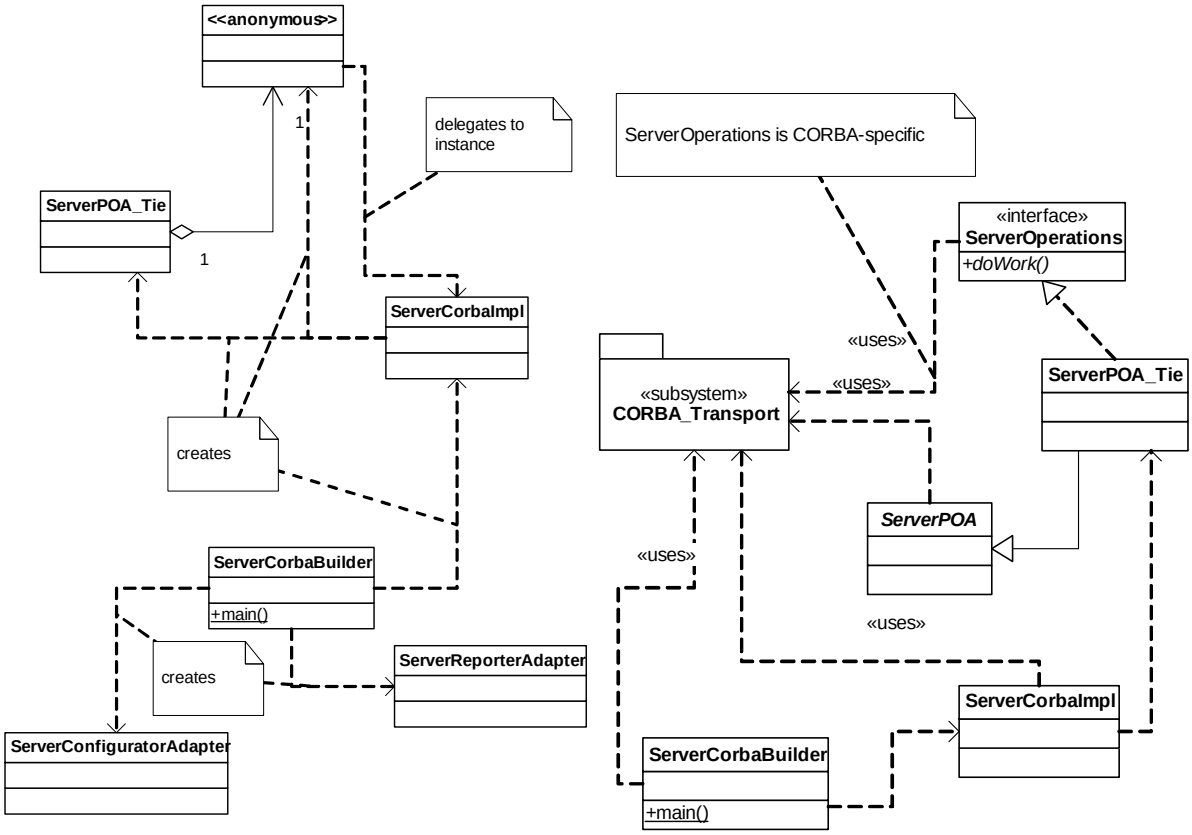


Which concrete implementation of ServerCorbaImpl and adapters builder should use is defined in configuration files for the application that are transparently loaded by a GenericClassFactory during classloading. Construction of a Server and its destruction is done this way:

- User launches application that leads to invocation of ServerCorbaBuilder and CORBA initialization
- ServerCorbaBuilder creates an instance of itself, which becomes a context for Server, remember that ServerCorbaBuilder implements ReferenceResolver.
- Then ServerCorbaBuilder uses AbstractConstructor to create adapters for Server and Server itself using configuration data.
- ServerCorbaImpl during its initialization registers Server with CORBA.
- After initialization completed ServerCorbaBuilder starts execution of a server and suspends
- When server stops ServerCorbaImpl receives finalize() call and unregisters server with CORBA thus causing ServerCorbaBuilder to continue execution.

- ServerCorbaBuilder finalizes components it created and finalizes CORBA subsystem then exits causing application to terminate

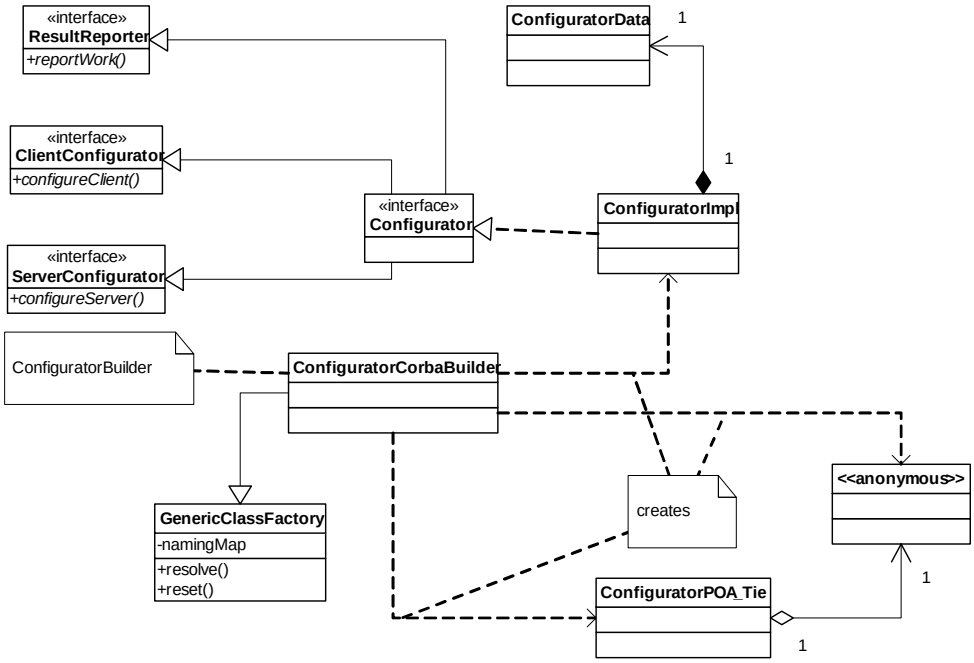
Construction process and dependencies between objects are presented below. It is worth mentioning that ServerCorbaImpl deals with issues concerning interactions with CORBA balancer service being the only part of the system dealing with load balancing. It could be thought of as a main class in the whole system because the goal of everything described was to benchmark load balancing algorithms in "the field".



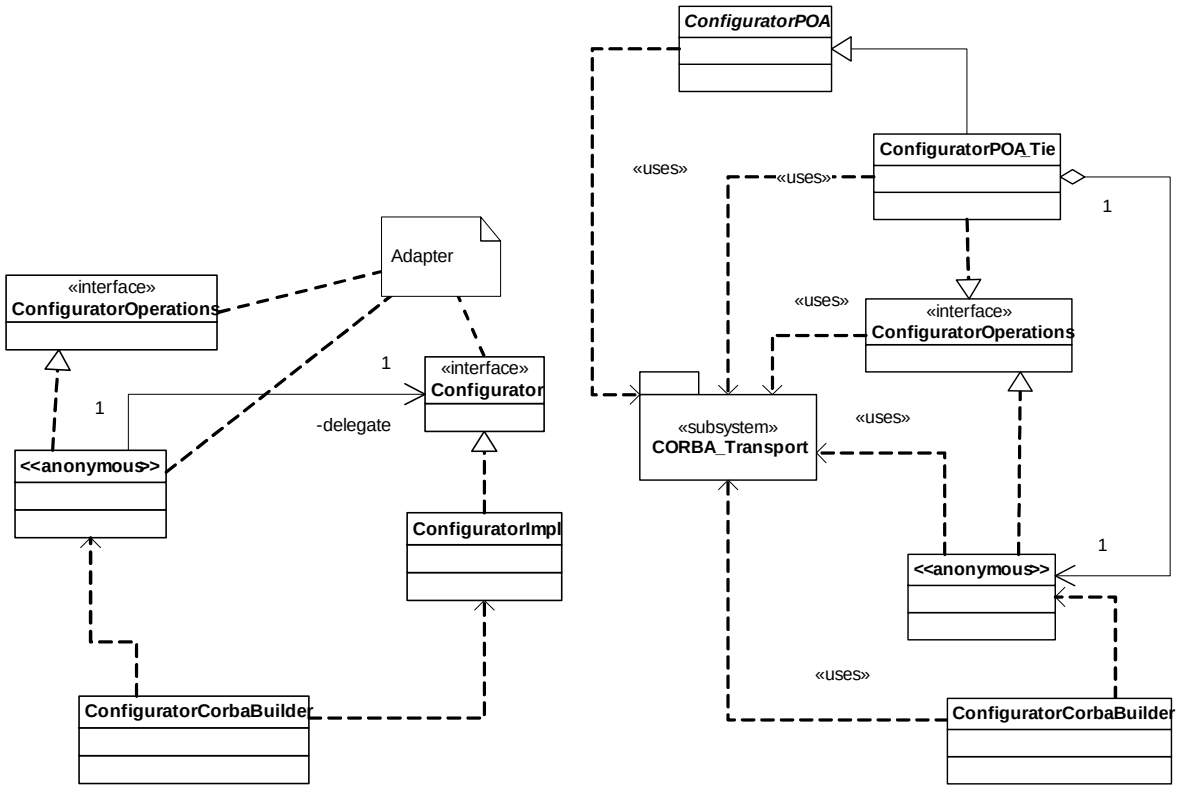
Anyway, there exists a basic solution in which initialization is done in one method, algorithms applied and requests served in another one and the other method performs finalization. This solution is simple enough to put it into one class but it is wrong in general, albeit simple and well understood at the first time.

Configurator Implementation for CORBA

The same idea of builder, delegation and construction was used for Configurator. ConfiguratorCorbaBuilder creates ConfiguratorPOA_Tie and anonymous class to translate calls of unified due to simplicity ConfiguratorOperations interface. Then it creates an object that implements Configurator and delegates to it. This object is defined in configuration files by user and constructed with AbstractConstructor. Context of Configurator is like in Server defined by ConfiguratorCorbaBuilder.

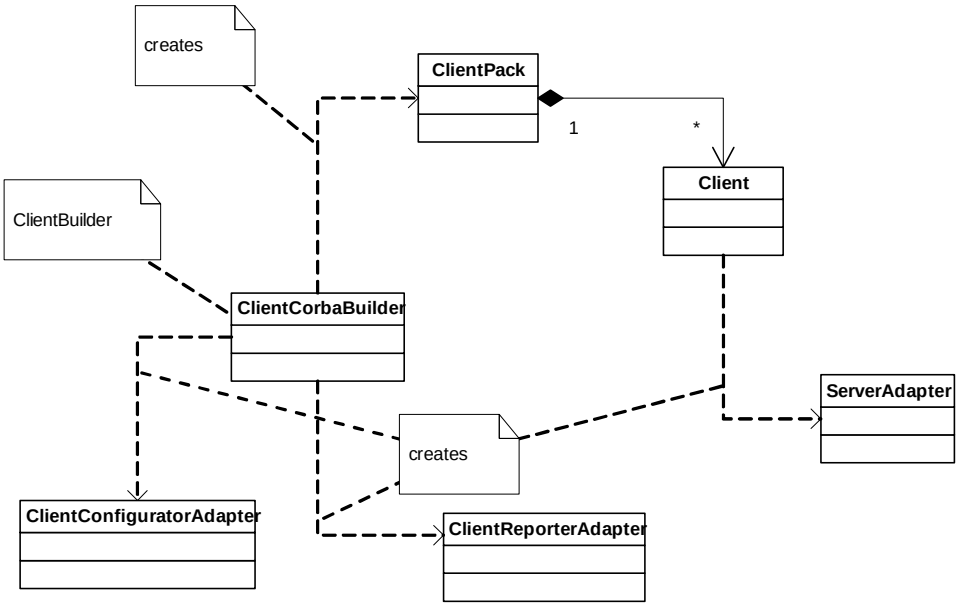


Because Configurator uses only simple CORBA server object techniques the entire CORBA initialization was placed into ConfiguratorCorbaBuilder. Diagrams below describe *Adapter* pattern separately from other classes involved in Configurator implementation and usage dependencies occurred in implementation.



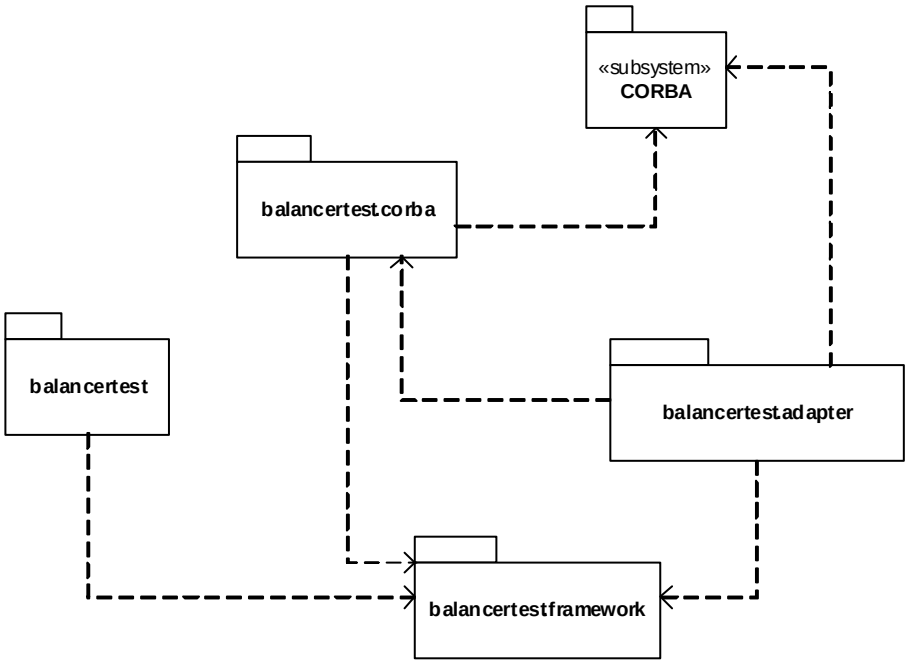
Client implementation for CORBA

Client implementation for CORBA is particularly simple because of it does not implement any CORBA server objects. Therefore, all necessary CORBA initialization is present in ClientCorbaBuilder. The diagram explains the solution.



Packaging

Let us now consider classes packaging and components that should exist in the system. First of all, there is a CORBA subsystem with its own packages. Then all classes dependent on transport but that are not adapters were placed into balancertest.corba package. All adapters were placed into balancertest.adapter package. To tell the truth, this separation is due to historical reasons. All classes that implement algorithms for Client, Server and Configurator were placed into balancertest package. Supplementary classes occurred in balancertest.framework package.



There are no circular dependencies between packages thus packaging satisfies Acyclic Dependencies Principle. Let us calculate instability for each package as an exercise.

Package name	Afferent coupling, C_a=incoming dependencies	Efferent coupling, C_e=outgoing dependencies	Instability, $I = C_e / (C_e + C_a)$
balancertest	0	14	1

balancertest.corba	~10 (from adapter)	6 (to framework) + ~20 (to CORBA) = ~30	3/4
balancertest.adapter	0	5 (to framework)	1
balancertest.framework	many	0	0
CORBA	many	0	0

Well, what do we have is almost perfect conformance to the Stable Dependencies Principle! There are two packages that contain abstract classes:

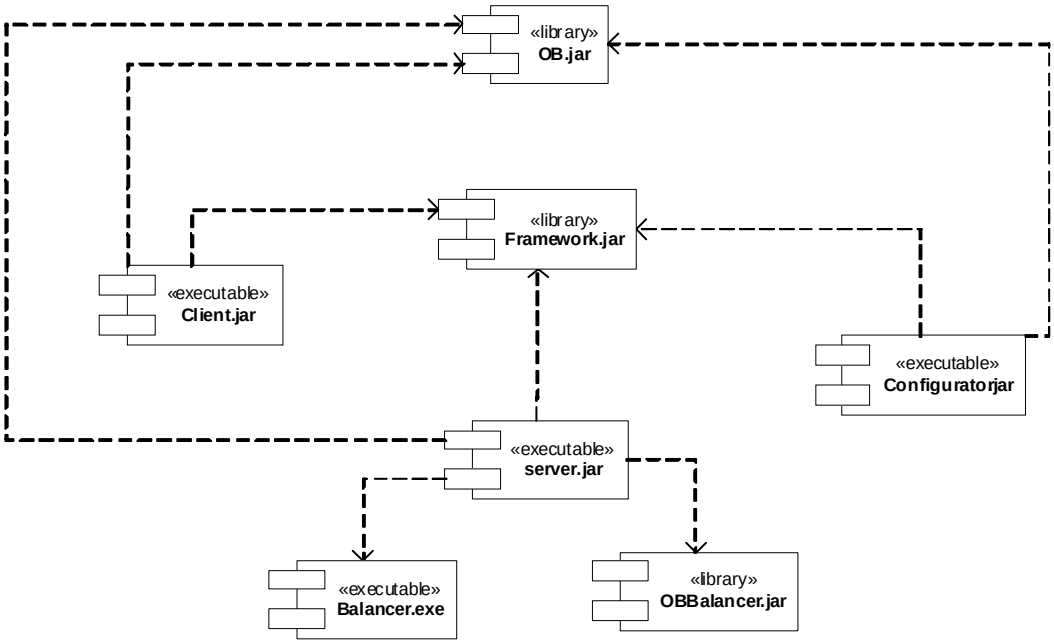
- The corbatest.framework package with abstractness equal to 3/7 and it has instability equal to zero.
- The corbatest package with abstractness equal to ½ and instability equal to 1

And there the problem is according to the Stable Abstractions Principle. These are two packages that quite a lot derivate from safety line Abstractness + Instability = 1. For framework this issue might be solved by splitting it into two packages – one for abstractions and another one for utilities. Package corbatest could be "saved" if we take all generic interfaces from it and place them into separate package or into framework.

Let us now consider component structure of Distributed Benchmark. It consists of several deliverable components as shown at the picture below. CORBA subsystem is presented by three components:

- OB.jar – generic ORBacus library (not present in system delivery)
- OBBalancer.jar – a library for accessing ORBacus Balancer (not present in system delivery)
- Balancer.exe – ORBacus balancer itself (not present in system delivery)

Client is represented by client.jar that contains all classes from balancertest related to Client and its adapters and classes from balancertest.corba that construct client. Server is deployed as server.jar, which contains server classes from balancertest package and adapters to Configurator and classes from balancertest.corba that create Server. Configurator is packaged into configurator.jar. All balancertest.framework classes were put into framework.jar along with remote interfaces and helper classes for CORBA objects.

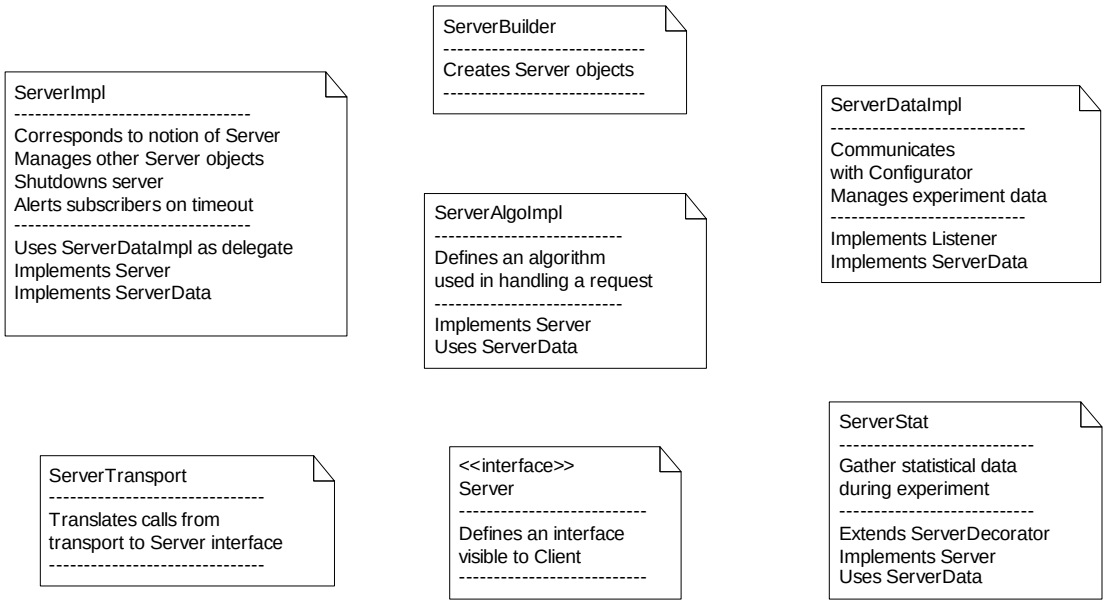


Further design steps

Study of the Distributed Benchmark design conducted some time after the system was implemented showed that implementation of Server has several drawbacks:

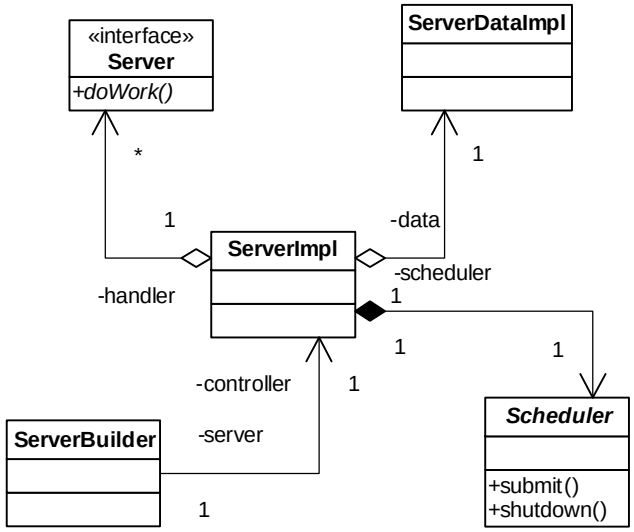
- It is difficult to change an algorithm of handling a request, as it is hardcoded into ServerImpl class even as a separate method.
- ServerImpl class plays several roles – serving the requests and gathering statistical data and storing results and retrieving experiment parameters from Configurator
- It might be difficult to port Server to a different platform because generic construction mechanism is absent.

In order to address these issues new design iteration was conducted during which the following classes were devised:

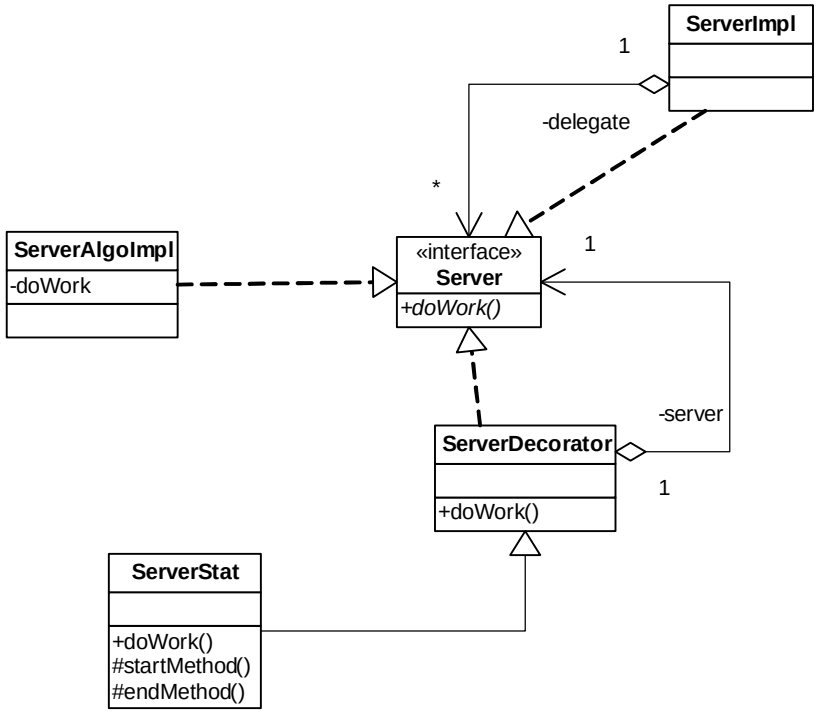


The general structure of implementation was thought of as shown at the diagram. ServerImpl plays the governing role, knows about and manages all other parts of the implementation but

itself actually does nothing – ServerImpl does not process requests neither creates objects nor stores experiment related data.

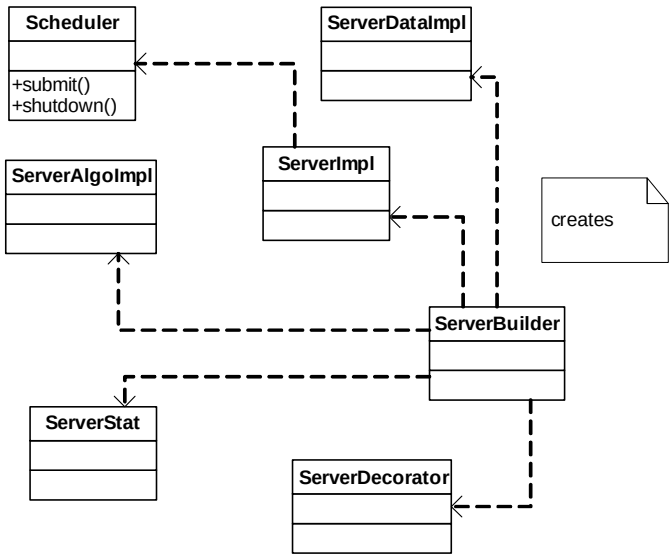


ServerImpl can receive notifications from Scheduler, it implements Server interface and can delegate calls to a private handler, which is also Server interface. ServerImpl remains Configurable and registers itself in ReferenceResolver to be known to other components and also implements ServerData interface and redirects calls to ServerDataImpl. Thus, we have a *Façade* pattern here. In order to decide dynamically, at least at construction phase, what algorithms we want to use, the *Delegate* pattern was applied to handling requests. Its scheme is presented at the diagram below.



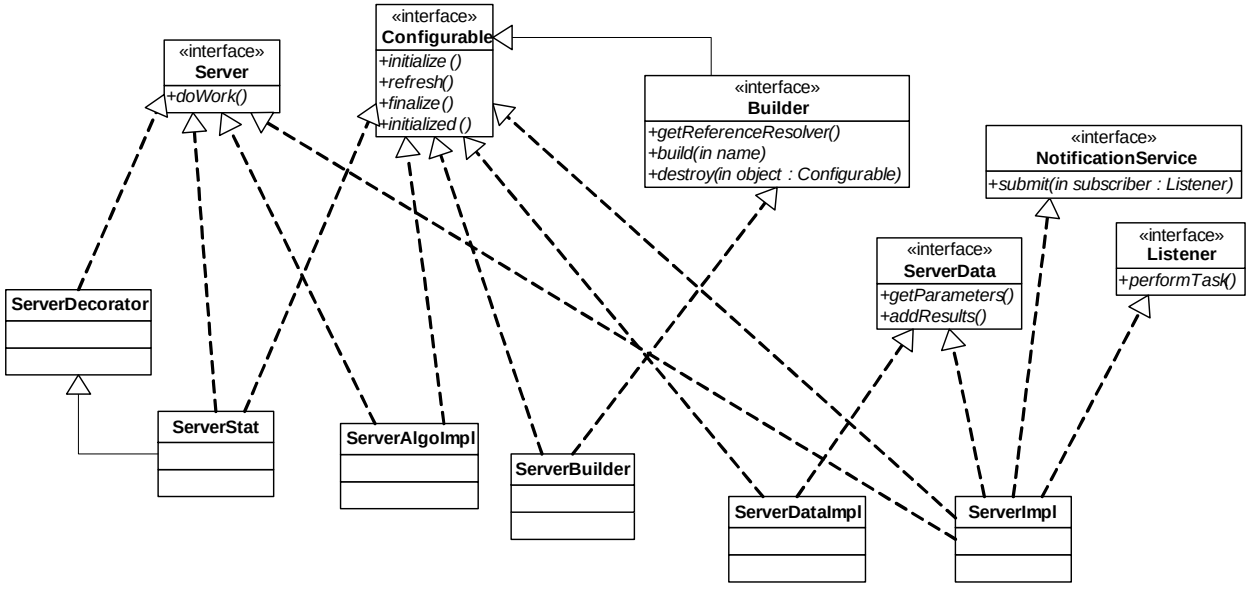
There are two algorithm depicted here: ServerStat, which gathers statistical data and then redirects through ServerDecorator to the next handler, and ServerAlgoImpl which is "leaf" implementation and actually handles requests. Both ServerStat and ServerAlgoImpl are Configurable and created by ServerBuilder. ServerBuilder is a concrete builder that descends from AbstractConstructor and "knows how" to create this implementation of Server, initialize, make its parts work together and finalize it. ServerBuilder creates all parts of Server implementation with inherited implementation of AbstractConstructor. Note that ServerBuilder

is a Configurable too, and all its building work it does during self-initialization. This way client should only know that there exists a name for ServerBuilder and its implementation and that is all! Construction dependencies are shown at the picture.

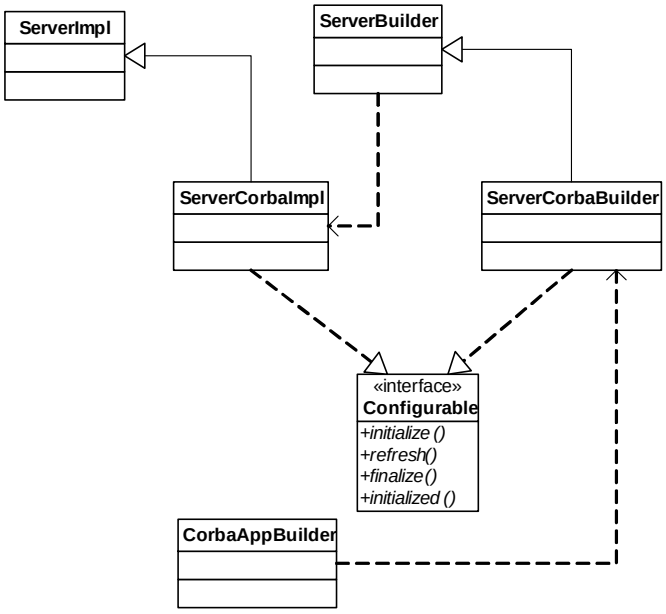


There are several roles in the server implementation:

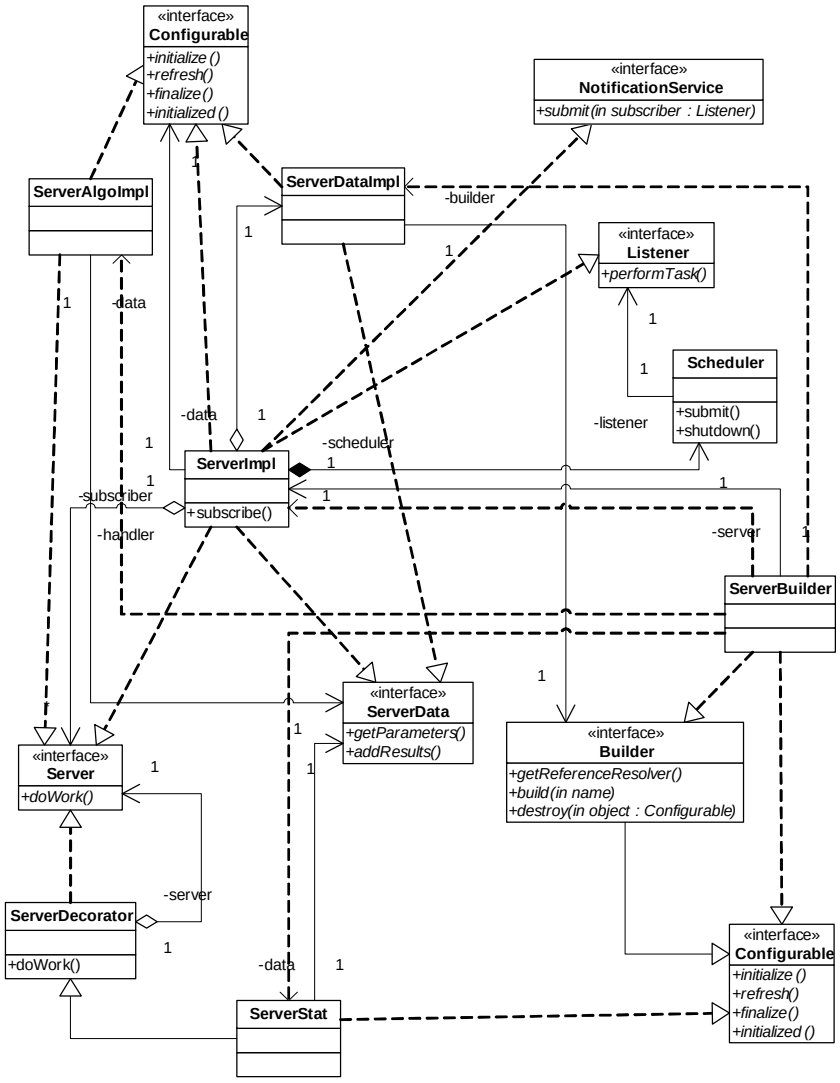
- Server – marks the ability to handle requests
- Configurable – defines a dynamically configurable object
- Builder – mark the Server builder which instance is accessible through ReferenceResolver
- NotificationService – generic service for scheduling alerts
- ServerData – implementing class promises to supply experiment parameters and store results and then pass them to Configurator
- Listener – marks the ability to respond to a timed alert and return new delay



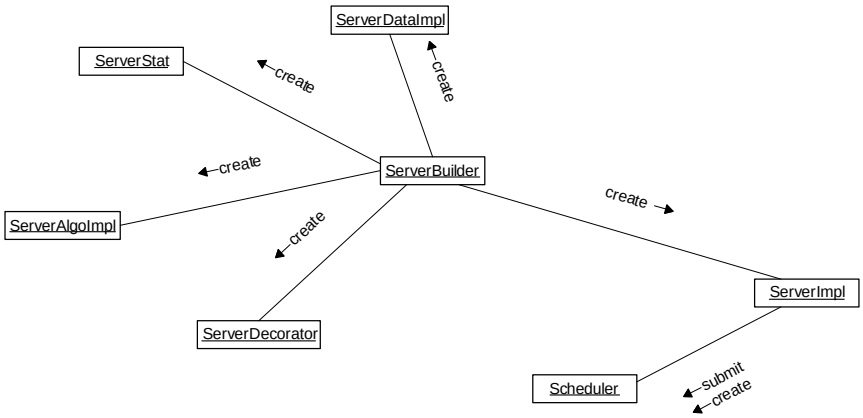
Although implementation grows much complex, extending and porting of Server remained simple and still consists of deriving corresponding classes from participating ones and defining new names in configuration data.

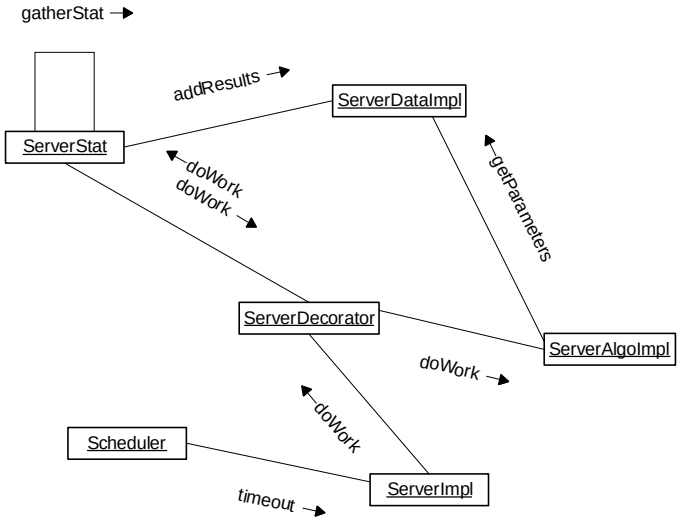


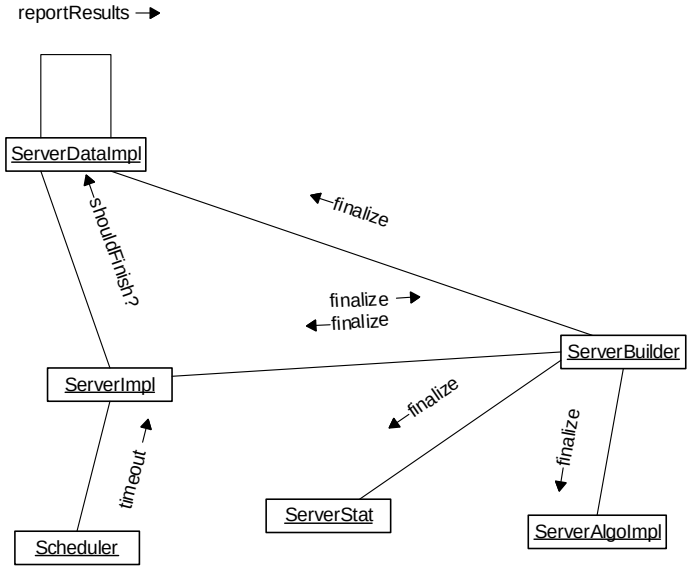
The overall picture is presented at the picture below. Note that this solution is still under development and this diagram is a bit cumbersome.



Dynamic of how this all works is sketched at the diagrams below first one describing construction process, second – request handling and the last one – the process of shutdown.


ServerDataImpl
ServerStat
create
create
create
ServerBuilder
create
ServerAlgoImpl
create
ServerImpl
ServerDecorator
Scheduler
submit
create


gatherStat
addResults
ServerDataImpl
ServerStat
doWork
doWork
getParameters
ServerDecorator
ServerAlgoImpl
doWork
doWork
Scheduler
ServerImpl
timeout


reportResults
ServerDataImpl
finalize
shouldFinish?
finalize
finalize
ServerBuilder
ServerImpl
finalize
finalize
timeout
ServerStat
ServerAlgoImpl
Scheduler

References

A published article on design principles and patterns,
“Principles and Patterns”, Robert C. Martin, objectmentor.com, 2000
http://www.objectmentor.com/

This paper was motivated by another Object Mentor article,
“Walking through a UML Design”, Robert C. Martin , James W. Newkirk, 1998
http://www.objectmentor.com/

Most diagrams presented are in UML 1.5,
“Unified Modeling Language”, OMG UML
http://www.uml.org/