

Design Concept Catalog

Attribute-Driven Design (ADD) in the Real World

Humberto Cervantes (hcm@xanum.uam.mx)

Rick Kazman (kazman@hawaii.edu)

SATURN 2013 Tutorial

May 3d, 2013

Table of contents

Introduction.....	4
Design Patterns.....	4
From mud to structure (structuring the system).....	4
Layers	4
Model View Controller (MVC).....	5
Presentation-Abstraction-Control (PAC).....	5
Microkernel.....	6
Pipes and Filters.....	7
Domain Object.....	8
Distribution infrastructure.....	8
Messaging.....	8
Publisher-Subscriber.....	9
Broker.....	10
Interface Partitioning.....	11
Explicit Interface.....	11
Proxy.....	12
Concurrency.....	12
Half-Sync/Half-Async.....	12
Leader-Followers.....	13
Object interaction.....	14
Observer.....	14
Command.....	15
Data Transfer Object.....	16
Resource Management.....	17
Lazy Acquisition.....	17
Database Access.....	17
Database Access Layer.....	17
Data Mapper (aka Data Access Object - DAO).....	18
Tactics.....	20
Availability tactics.....	20
Interoperability tactics.....	22
Modifiability tactics.....	23
Performance tactics.....	24
Security tactics.....	25
Testability tactics.....	26
Usability tactics.....	28
Frameworks.....	29
Structuring the system + supporting non-functional concerns.....	29
Spring framework.....	29
User interface.....	30
Java Server Faces framework.....	30
Swing Framework.....	31
OO-Relational Mapping.....	32
Hibernate Framework.....	32
MyBatis Framework.....	33

<u>Remote communication.....</u>	<u>34</u>
<u>RMI Application Programming Interface.....</u>	<u>34</u>
<u>Axis2 Web Services Framework.....</u>	<u>35</u>
<u>JMS Framework.....</u>	<u>35</u>
<u>Deployment.....</u>	<u>37</u>
<u>Java Web Start Framework.....</u>	<u>37</u>

Introduction

This catalog lists three types of *design concepts* that are to be used in the exercise part of the tutorial. The types of design concepts include design patterns, tactics and frameworks.

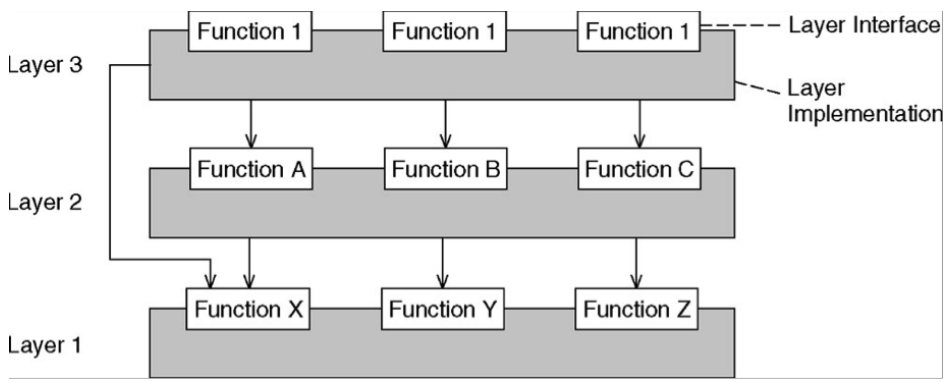
Design Patterns

Note: Parts of the design pattern catalog are based on the book *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4* by Frank Buschmann, Kevlin Henney and Douglas C. Schmidt, John Wiley & Sons, 2007

Numbers between parentheses (e.g. Domain Model (182)) represent the page in the book where the pattern is documented.

From mud to structure (structuring the system)

Layers

Problem and context	When transforming a Domain Model (182) into a technical software architecture, [...] we must support the independent development and evolution of different system parts.
Solution	Define one or more layers for the software under development, with each layer having a distinct and specific responsibility.
Structure	
Consequences and related patterns	Typically, each self-contained and coherent responsibility within a layer is realized as a separate Domain Object, to further partition the layer into tangible parts that can be developed and evolved independently.

Model View Controller (MVC)

Problem and context	When transforming a Domain Model (182) into a technical software architecture, [...] we must consider that the user interface of an application changes more frequently than its domain functionality.
Solution	Divide the interactive application into three decoupled parts: processing, input, and output. Ensure the consistency of the three parts with the help of a change propagation mechanism.
Structure	<p>The diagram illustrates the MVC pattern structure. On the left, a User interacts with the User Interface. The User Interface contains a View (with display and update components) and a Controller (with do something and update components). A vertical dashed line separates the User Interface from the Application Functionality and Model. The Application Functionality contains get data, function_2, and function_1. The Model contains data_1, data_2, and data_3, along with a notify component. Interactions are numbered: (1) <i>invoke</i> from Controller to Application Functionality; (2) <i>modify</i> from Application Functionality to Model; (3) <i>start change notification</i> from Model to Application Functionality; (4) <i>notify</i> from Application Functionality to View; (5) <i>update state</i> from View to Controller. Dashed arrows also show the User interacting with the View and Controller, and the View interacting with the Model's data elements.</p>
Consequences and related patterns	<p>Connect the model, view, and controller components via a change propagation mechanism: when the model changes its state, notify all views and controllers about this change so that they can update their state accordingly and immediately via the model's APIs.</p> <p>View renders model information into a predefined output format. A Transform View creates its output by rendering each data element individually that it retrieves from the model.</p> <p>Each view of the system is associated with one more controllers to manipulate the model's state. A controller receives input through an associated input device such as a keyboard or a mouse, and translates it into requests to its associated view or the model. There are three common types of controller: a controller associated with a specific function in the application's user interface, a Page Controller (337) that handles all requests issued by a specific form or page in the user interface, and a Front Controller (339) that handles all requests on the model. A controller per function is most suitable if the model supports a wide range of functions. A Page Controller is appropriate for form-based or page-based user interfaces in which each form or page offers a set of related functions. A Front Controller is most usable if the application publishes functions to the user interface whose execution can differ for each specific request, such as the HTTP protocol of a Web application.</p>

Presentation-Abstraction-Control (PAC)

Problem and context	User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications to the user interface software separate from the rest of the system. Users often wish to
----------------------------	---

	<p>look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.</p> <p>How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?</p>
Solution	<p>The presentation-abstraction-control (PAC) model structures a UI recursively as a hierarchy of PAC "agents". The functional core is implemented by the abstraction component. The presentation corresponds loosely to a presentation and interaction toolkit. The controller component manages the relationships between PAC agents. The controller may itself be a PAC hierarchy. Each PAC agent has its own abstraction which is only accessible to others through its controller. The controller components pass information up and down the composition hierarchy, potentially performing some transformations of this data.</p>
Structure	<p>The diagram illustrates the PAC model structure. It shows a hierarchy of PAC agents: Top-level, Intermediate-level, and Bottom-level. Each agent consists of three main components: Presentation (display, do something), Control (mediate), and Abstraction (function_1, function_2). A User interacts with the Presentation component of the Top-level agent. The Control component (mediate) acts as a central hub, coordinating between the Presentation and Abstraction components of the same agent and with the Control components of other agents in the hierarchy. Arrows labeled 'coordinates' indicate the flow of information between the Control components of different agents. The Abstraction components provide the core functionality, while the Presentation components handle user interaction.</p>
Consequences and related patterns	<p>The PAC model provides good support for multi-modal interfaces and for complex interfaces that are continually being changed and extended. Each of these agents can potentially have their own thread of control which can, in some cases, reduce latency. However this comes at a price of additional system complexity and communication overhead amongst the agents.</p>

Microkernel

Problem and context	<p>When transforming a Domain Model (182) into a technical software architecture we must design support for functional scalability and adaptability in different deployment scenarios.</p>
Solution	<p>Compose different versions of the application by extending a common but minimal core via a 'plug-and-play' infrastructure.</p>

Structure	<p>The diagram illustrates the Microkernel architecture. A User interacts with an External Server (GUI) (containing <i>display</i> and <i>do something</i>) and an External Server (API) (containing <i>function_1</i>, <i>function_2</i>, and <i>function_3</i>). The Microkernel contains <i>function_1</i>, <i>function_2</i>, a <i>route_request</i> component, and server management functions (<i>register_svr</i>, <i>unregister_svr</i>). An Internal Server contains <i>function_3</i>. A System (represented by a folder icon) interacts with the External Server (API). Arrows indicate the flow of requests and data between these components.</p>
Consequences and related patterns	<p>A microkernel implements the functionality shared by all application versions and provides the infrastructure for integrating version-specific functionality. Internal servers implement self-contained version-specific functionality, and external servers version-specific user interfaces or APIs.</p> <p>Each specific and self-contained function and responsibility within the microkernel can be realized as a Domain Object (208), which supports its independent implementation and evolution. [...]. This design also supports the upgrade of a particular application version with new, different, or modified functionality dynamically at runtime.</p>

Pipes and Filters

Problem and context	<p>Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice and so it is desirable to create these as independent, reusable parts. Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.</p>
Solution	<p>The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.</p>
Structure	<p>The diagram shows the structure of the Pipe and Filter pattern. It starts with an Input Device connected to Filter 1 (with an <i>input</i> port). The output of Filter 1 goes through Pipe 1 (with a <i>buffer</i>) to Filter 2 (with an <i>input</i> port). This sequence continues through Pipe N-1 (with a <i>buffer</i>) to Filter N (with an <i>input</i> port), which finally connects to an Output Device.</p>
Consequences and related patterns	<p>There are several weaknesses associated with the pipe and filter pattern. For instance this pattern is typically not a good choice for an interactive system as it disallows cycles (which are important for user feedback). Also, having large</p>

	<p>numbers of independent filters can add substantial amounts of computational overhead, since each filter runs as its own thread or process. Also, pipe and filter systems may not be appropriate for long-running computations, without the addition of some form of checkpoint/restore functionality, as the failure of any filter (or pipe) can cause the entire pipeline to fail.</p>
--	--

Domain Object

Problem and context	<p>When realizing a Domain Model (182), or its technical architecture in terms of Layers (185), [...] a key concern of all design work is to decouple self-contained and coherent application responsibilities from one another.</p>
Solution	<p>Encapsulate each distinct functionality of an application in a self-contained building-block—a domain object.</p>
Structure	<p>The diagram illustrates the structure of domain objects. It shows four domain objects: Domain Object 1 (containing Function A), Domain Object 2 (containing Function B and Function C), Domain Object 3 (containing Function X and Function Y), and Domain Object 4 (containing Function Z). A dashed line labeled 'Domain Object Interface' points to Function A. A solid line labeled 'Domain Object Implementation' points to Function X. Arrows indicate dependencies: Function A depends on Function X and Function Y; Function B depends on Function X and Function Y; Function C depends on Function Z.</p>
Consequences and related patterns	<p>The specific partitioning of an application's responsibilities into domain objects is based on one or more granularity criteria. There can be different types of domain objects such as Application Services that encapsulate self-contained and complete business feature or infrastructure aspect of an application, such as a banking, flight booking, or logging service. Components are domain objects that either encapsulates a functional building block such as an income tax calculation or a currency conversion, or a domain entity such as a bank account or a user. A domain object can also aggregate other domain objects of the same or smaller granularity.</p> <p>Split each domain object into an Explicit Interface (281) that exports its functionality and an Encapsulated Implementation (313) that realizes the functionality. This separation of interface and implementation minimizes inter-domain-object coupling: each domain object only depends on domain object interfaces, but not on domain object implementations. It is thus possible to realize and evolve a domain object implementation with minimal effect on other domain objects.</p>

Distribution infrastructure

Messaging

Problem and context	Some distributed systems are composed of services that were developed independently. To form a coherent system, however, these services must interact reliably, but without incurring overly tight dependencies on one another. Integrating independently developed services, each having its own business logic and value, into a coherent application requires reliable collaboration between services. However, since services are developed independently, they are generally unaware of each other's specific functional interfaces.
Solution	Connect the services via a message bus that allows them to transfer data messages asynchronously. Encode the messages (request data and data types) so that senders and receivers can communicate reliably without having to know all the data type information statically.
Structure	
Consequences and related patterns	<p>Services can interact without having to deal with networking and service location concerns.</p> <p>Asynchronous messaging allows services to handle multiple requests simultaneously without blocking.</p> <p>Allows services to participate in multiple application integration and usage contexts.</p> <p>However, the lack of statically typed interfaces makes it hard to validate system behavior prior to runtime. Furthermore service requests are encapsulated within self-describing messages that require extra time and space for message processing.</p>

Publisher-Subscriber

Problem and context	There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers is not predetermined or fixed, nor is the data that they share. How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way they are unaware of each other's identity, or potentially even their existence?
Solution	In the Publish-subscribe pattern components interact via announced messages, or events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus, the main form of connector in these patterns is an <i>event bus</i> . Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber

	components that have registered an interest in those events. Any component may be both a publisher and a subscriber.
Structure	<p>The diagram illustrates the Change Propagation Infrastructure pattern. It features four components: Publisher 1, Subscriber 2, Subscriber 3, and Publisher 2. Each component is represented by a gray rectangular box. Below each box is a smaller box labeled 'state change'. A line connects the 'state change' box to a small circle. These circles are connected to a central horizontal bar labeled 'Change Propagation Infrastructure'. Arrows indicate the flow of state changes from the publishers to the infrastructure and from the infrastructure to the subscribers.</p>
Consequences and related patterns	<p>All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system.</p> <p>This pattern typically increases latency and has a negative effect on scalability and predictability of message deliver time.</p> <p>Also there is less control over ordering of messages and delivery of messages is typically not guaranteed.</p>

Broker

Problem and context	<p>Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services. How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?</p>
Solution	<p>The Broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a Server which processes the request. The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client. In this way the client remains completely ignorant of the identity, location, and characteristics of the Server. Because of this separation, if a server becomes unavailable, a replacement can be dynamically chosen by the broker. If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change and so the client is unaffected. Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshalling and unmarshalling messages.</p>

Structure	
Consequences and related patterns	<p>The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy). The broker has a number of side effects and potential weaknesses that need to be considered:</p> <ul style="list-style-type: none"> • Brokers add a layer of indirection, and hence latency, between clients and servers and it may be a communication bottleneck. • The broker can be a single point of failure. • A broker adds up-front complexity. • A broker may be a target for security attacks. • A broker may be difficult to test.

Interface Partitioning

Explicit Interface

Problem and context	<p>When designing a Layers (185), Domain Object (208), [...] a major concern of all software architecture work is the effective and appropriate expression of component interfaces.</p> <p>A component represents a self-contained unit of functionality and deployment with a published usage protocol. Clients can use it as a building block in providing their own functionality. Direct access to the full component implementation, however, would make clients dependent on component internals, which ultimately increases application internal software coupling.</p>
Solution	<p>Separate the declared interface of a component from its implementation. Export the interface to the clients of the component, but keep its implementation private and location-transparent to the client.</p>
Structure	
Consequences and related patterns	<p>A call from the client through this explicit interface will be forwarded to the component, but the client code will depend only on the interface and not on the implementation.</p> <p>An Explicit Interface enforces a strict separation of the component's interface from its concrete implementation, which separates component usage issues from</p>

	concrete realization and location details. This separation also enables the transparent modification of component implementations independently of the clients using it, as long as the contract defined by the interfaces remains stable.
--	--

Proxy

Problem and context	<p>When specifying an Explicit Interface (281), we often want to avoid accessing services of a component implementation directly.</p> <p>Software systems consist of cooperating components: client components access and use the services provided by other components. It is often impractical, or even impossible, to access the services of a component directly, for example because we must first check the access rights of its clients, or because its implementation resides on a remote server.</p>
Solution	Encapsulate all component housekeeping functionality within a separate surrogate of the component—the proxy—and let clients communicate only through the proxy rather than with the component itself.
Structure	<pre> method_A () begin ## Do pre-processing. comp := load_comp_from_DB (compID); ## Call the method on the component. comp.method_A_imp (); ## Do Post-processing unload_comp_from_user_space (compID); end </pre>
Consequences and related patterns	A Proxy frees both the client and the component from implementing component-specific housekeeping functionality. It is also transparent to clients whether they are connected with the component or its proxy, because both publish an identical interface. The primary liabilities of a proxy are the hidden costs it can introduce for clients, although for many uses these costs are negligible compared to the execution time of the component's services.

Concurrency

Half-Sync/Half-Async

Problem and context	<p>When developing concurrent software, [...] we need to make performance efficient and scalable while ensuring that any use of concurrency simplifies programming.</p> <p>Concurrent software often performs both asynchronous and synchronous service</p>
----------------------------	---

	<p>processing. Asynchrony is used to process low-level system services efficiently, synchrony to simplify application service processing. To benefit from both programming models, however, it is essential to coordinate asynchronous and synchronous service processing efficiently.</p>
Solution	<p>Decompose the services of concurrent software into two separated layers—synchronous and asynchronous—and add a queueing layer to mediate communication between them.</p>
Structure	<p>The diagram illustrates a three-layer architecture. The top layer, 'Synchronous Service Layer', contains three 'method' boxes. The middle layer, 'Queueing Layer', contains two 'Message' boxes and two operations: 'remove' and 'insert'. The bottom layer, 'Asynchronous Service Layer', contains two 'method' boxes, each connected to a 'Network I/O' icon. Arrows show data flow: from 'method' boxes in the top layer to 'Message' boxes in the middle layer; from 'Message' boxes to 'remove'/'insert' operations; and from 'method' boxes in the bottom layer to 'Network I/O'.</p>
Consequences and related patterns	<p>Process higher-level services, such as domain functionality, database queries, or file transfers, synchronously in separate threads or processes. Conversely, process lower-level system services, such as short-lived protocol handlers driven by interrupts from network hardware, asynchronously. If services in the synchronous layer must communicate with services in the asynchronous layer, have them exchange messages via a queueing layer.</p> <p>In general, a Half-Sync/Half-Async arrangement employs Layers (185) to keep its three distinct execution and communication models independent and encapsulated.</p>

Leader-Followers

Problem and context	<p>When developing concurrent software, [...] we must often react on and process multiple events from multiple event sources both concurrently and efficiently.</p> <p>Most event-driven software uses multi-threading to process multiple events concurrently. It is surprisingly hard, however, to allocate work to threads in an efficient, predictable, and simple manner.</p>
Solution	<p>Use a pre-allocated pool of threads to coordinate the detection, demultiplexing, dispatching, and processing of events. In this pool only one thread at a time—the leader —may wait for an event on a set of shared event sources. When an event arrives, the leader promotes another thread in the pool to become the new leader and then processes the event concurrently.</p>
Structure	<p>The diagram shows a thread pool structure. 'Network events' enter 'Event Sources'. An arrow points from 'Event Sources' to a 'wait_for_events' box. From 'wait_for_events', an arrow points to a 'Leader Thread' box. From 'Leader Thread', an arrow points to a 'handle_event' box. From this 'handle_event' box, an arrow points to a 'Processing Thread' box. From 'Processing Thread', an arrow points to a 'Follower Threads' box. From 'Follower Threads', an arrow points to another 'handle_event' box. Dashed arrows indicate a cycle where the 'Leader Thread' can become a 'Follower Thread' and vice versa.</p>

Consequences and related patterns	<p>By pre-allocating a pool of threads, a Leader/Followers design avoids the overhead of dynamic thread creation and deletion. Having threads in the pool self-organize and not exchange data between themselves also minimizes the overhead of context switching, synchronization, data movement, and dynamic memory management. Moreover, letting the leader thread perform the promotion of the next follower prevents performance bottlenecks arising from having a centralized manager make the promotion decisions.</p> <p>The price to pay for such performance optimizations is limited applicability. A Leader/Followers configuration only pays off for short-duration, atomic, repetitive, and event-based actions, such as receiving and dispatching network events or storing high-volume data records in a database. The more services the event handlers offer, the larger they are in size, while the longer they need to execute a request, the more resources a thread in the pool occupies and the more threads are needed in the pool. Correspondingly fewer resources are available for other functionality in the application, which can have a negative impact on the application's overall performance, throughput, scalability, and availability.</p>
--	--

Object interaction

Observer

Problem and context	<p>In a Layers (185), Model-View-Controller (188), Presentation-Abstraction-Control (191), [...], or Database Access Layer (538) arrangement we must provide a means to keep the state of a set of cooperating component objects consistent with each other.</p> <p>Consumer objects sometimes depend on the state of, or data maintained by, another provider object. If the state of the provider object changes without notice, however, the state of the dependent consumer objects can become inconsistent.</p>
Solution	<p>Define a change-propagation mechanism in which the provider— known as the 'subject'—notifies registered consumers—known as the 'observers'—whenever its state changes, so that the notified observers can perform whatever actions they deem necessary.</p>

Structure	
Consequences and related patterns	<p>Observers must define a specific update interface that is notified by the subject when its state changes. This interface is the primary coupling between the subject and its observers. Observers can register and unregister from the subject dynamically. When the subject notifies its observes, it can either push the state to the observers along with the state change notification, or the observers can selectively pull the changed state from the subject at their discretion after being notified.</p> <p>In an Observer arrangement, the dynamic registration of observers with the change notification mechanism avoids hard-coding dependencies between the subject and its observers: they can join and leave at any time, and new types of observer that implement the update interface can be integrated without changing the subject. The active propagation of changes by the subject avoids polling and ensures that observers can update their own state immediately in response to state changes in the subject.</p>

Command

Problem and context	<p>When implementing Layers (185), Model-View-Controller (188), Presentation-Abstraction-Control (191), [...] we must invoke actions on a component independently of selecting the actions to invoke.</p> <p>Accessing an object typically involves calling one of its methods. Sometimes, however, it is useful to decouple the sender of a request from its receiver. It may also be useful to decouple the selection of a request and receiver from the point of execution.</p>
Solution	<p>Encapsulate requests to the receiving object in command objects, and provide these objects with a common interface to execute the requests they represent.</p>
Structure	
Consequences	<p>Clients that want to issue a particular request create the corresponding command</p>

and related patterns	<p>object. When invoked, the command object initiates and controls the execution of the represented request with respect to any arguments it receives when executed.</p> <p>Command decouples the requestor of behavior from the recipient, as well as the selection of behavior from the point of execution. This loose coupling ensures that requestors do not depend on a specific receiver interface. Modifications to the receiver's interface therefore do not ripple through to its clients. In addition, the reification of requests into command objects allows the handling of requests as first-class entities within an application, which in turn enables the implementation of extra request-handling features such as undo and logging.</p>
-----------------------------	--

Data Transfer Object

Problem and context	<p>When realizing a Model-View-Controller (188), Presentation-Abstraction-Control (191), Microkernel (194), [...] we must make calls to query and update data in remote component objects.</p> <p>Many stateful component objects need an interface that supports querying and optionally updating of their attributes. Remote communication can incur significant overhead on each call, however, so individual methods for getting and setting each attribute are inefficient.</p>
Solution	Bundle all data items that might be needed into a single data transfer object used for querying or updating attributes together.
Structure	<pre> sequenceDiagram participant Client participant Component participant DTO as Data Transfer Object Note over Component: method Client->>Component: 1 invoke method Component->>DTO: 2 create data transfer object Component-->>Client: 3 return results Client->>DTO: 4 access data transfer object Note over DTO: get_results </pre>
Consequences and related patterns	<p>A Data Transfer Object has little behavior of its own, containing only the data corresponding to the attributes, queries to access them, and a way of initializing and optionally setting the data values.</p> <p>The resulting Data Transfer Object is more network-friendly, since only a single remote call is needed to query or update a set of attributes. If the set of attributes changes, moreover, the call interface of the remote object remains stable, even though the data transfer object changes.</p>

Resource Management

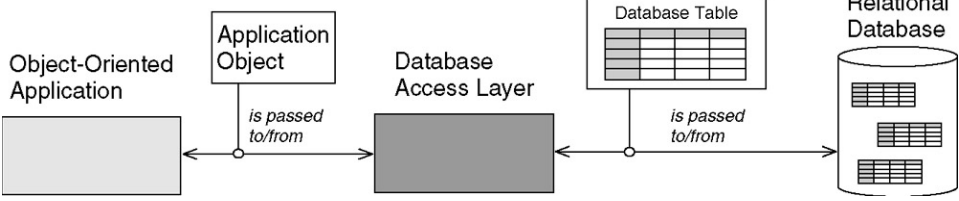
Lazy Acquisition

Problem and context	<p>Applications that access many resources, but which must also satisfy high availability requirements, need a way to reduce the initial cost of acquiring the resources they need or the resource usage footprint that they have at any point in time.</p> <p>In particular, acquiring all resources during system or subsystem initialization can make start-up unnecessarily or even unacceptably slow. Moreover, many resources may be acquired over-optimistically, making the initial acquisition wasteful if they are not consumed during the lifetime of the application.</p>
Solution	<p>Acquire resources at the latest possible point in time. The resource is not acquired until it is actually about to be used. At the point at which a resource user is about to use a resource, it is acquired and returned to the resource user.</p>
Structure	<pre> void method_A () begin ## Check if resource is acquired, ## and acquire it, if not. if (resource == NULL) resource = acquire_resource (); ## Execute method using the resource. ... end </pre>
Consequences and related patterns	<p>Lazy acquisition is an optimistic optimization that defers, but does not eliminate, the cost of resource acquisition. Its use should be fully encapsulated by the resource provider, shielding the resource user from the policy and mechanism details.</p>

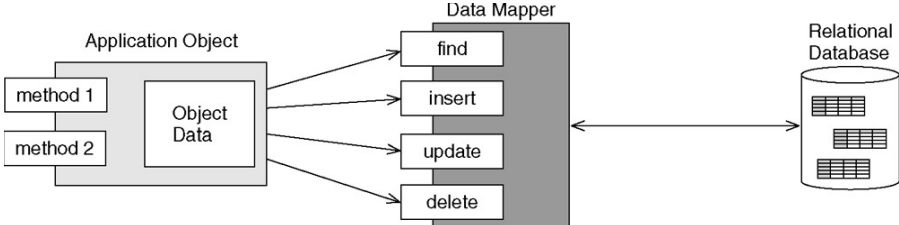
Database Access

Database Access Layer

Problem and context	<p>When realizing a Domain Model (182), [...] we must often connect an object-oriented application to a relational database.</p>
Solution	<p>Introduce a separate database access layer between the application and the relational database that provides a stable object oriented data-access interface for application use, backed by an implementation that is database-centric.</p>

Structure	
Consequences and related patterns	<p>Applications can store and retrieve their persistent data by calling an appropriate method on the database access layer, or by asking the data to persist itself, depending on the design. The database access layer is responsible for mapping between the data structures used by the applications and the format required by the database tables.</p> <p>Database Access Layer decouples an object-oriented application from the details of the database. All concrete mappings of objects to tables are encapsulated within this layer, so that it appears to the application as if it were storing and retrieving 'its own' objects rather than table entries. Database Access Layer thus offers a suitable bridge to the underlying persistence technology. In addition, modifications to the Database Access Layer do not affect application components directly</p>

Data Mapper (aka Data Access Object - DAO)

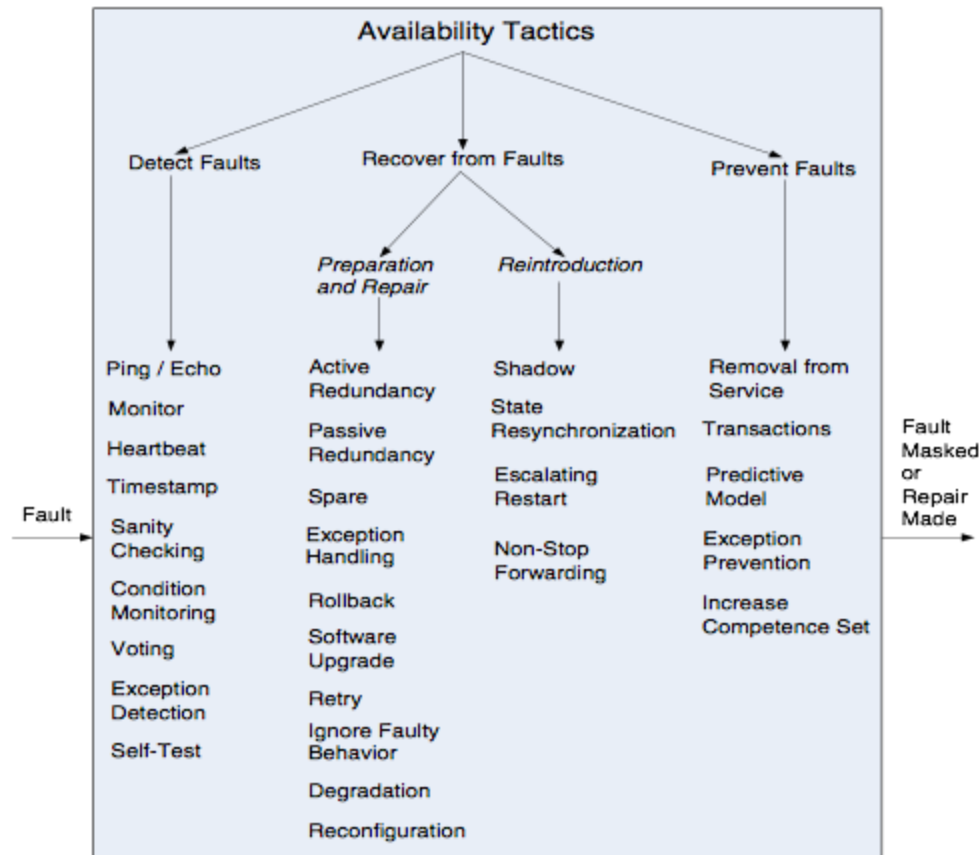
Problem and context	<p>When designing a Database Access Layer (538) we must shield applications from the way in which data is represented in persistent storage.</p> <p>Object-oriented applications and relational databases use different mechanisms for structuring data. While it is still necessary to transfer data between the two, if the object-oriented domain model knows about the relational database schema, and vice versa, changes in one tend to ripple to the other.</p>
Solution	<p>Introduce a data mapper for each type of persistent application object whose responsibility is to transfer data from the object to the relational database, and vice versa.</p>
Structure	
Consequences and related patterns	<p>A data mapper is a mediator between an object-oriented domain model and a relational database. A client can use the data mapper to retrieve an application object from the database, or ask it to store an application object in it. The data mapper performs all necessary data transformations and ensures consistency</p>

	<p>between the two representations.</p> <p>Using a Data Mapper, in-memory objects need not know that a database is present. Moreover, they require no SQL interface code and have no knowledge of the database schema. Data Mapper allows the relational database schema and the object-oriented domain model to evolve independently. This design also simplifies unit testing, allowing mappers to real databases to be replaced by mock objects that support in-memory test fixtures.</p> <p>Data Mapper simplifies application objects both programmatically and in terms of their dependencies. It offers a degree of isolation and stability, protecting both application objects and schemas from changes in the other. Data Mapper is not without its own complexity, however, and changes in either the application object model or the database schema may require changes to a data mapper.</p>
--	--

Tactics

Note: The Tactics catalog is based on the book *Software Architecture in Practice, Third Edition* by Len Bass, Paul Clements and Rick Kazman, Addison Wesley, 2013

Availability tactics



Detect faults

- Ping/echo: asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.
- Monitor: a component used to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- Heartbeat: a periodic message exchange between a system monitor and a process being monitored.
- Timestamp: used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- Sanity Checking: checks the validity or reasonableness of a component's operations or outputs; typically based on a knowledge of the internal design, the state of the system, or the nature of the

information under scrutiny.

- Condition Monitoring: checking conditions in a process or device, or validating assumptions made during the design.
- Voting: to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy.
- Exception Detection: detection of a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, parameter typing, timeout.
- Self-test: procedure for a component to test itself for correct operation.

Recover from Faults (Preparation and Repair)

- Active Redundancy (hot spare): all nodes in a protection group receive and process identical inputs in parallel, allowing redundant spare(s) to maintain synchronous state with the active node(s).
- Passive Redundancy (warm spare): only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates.
- Spare (cold spare): redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service.
- Exception Handling: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- Rollback: revert to a previous known good state, referred to as the “rollback line”.
- Software Upgrade: in-service upgrades to executable code images in a non-service-affecting manner.
- Retry: where a failure is transient retrying the operation may lead to success.
- Ignore Faulty Behavior: ignoring messages sent from a source when it is determined that those messages are spurious.
- Degradation: maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- Reconfiguration: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

Recover from Faults (Reintroduction)

- Shadow: operating a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role.
- State Resynchronization: partner to active redundancy and passive redundancy where state information is sent from active to standby components.
- Escalating Restart: recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected.
- Non-stop Forwarding: functionality is split into supervisory and data. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.

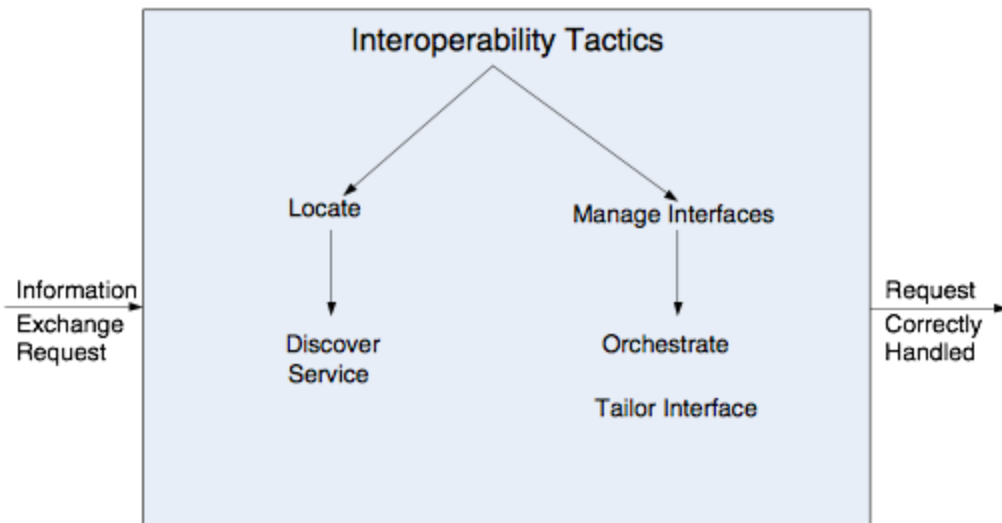
Prevent Faults

- Removal From Service: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures
- Transactions: bundling state updates so that asynchronous messages exchanged between

distributed components are atomic, consistent, isolated, and durable.

- Predictive Model: monitor the state of health of a process to ensure that the system is operating within nominal parameters; take corrective action when conditions are detected that are predictive of likely future faults.
- Exception Prevention: preventing system exceptions from occurring by masking a fault, or preventing it via smart pointers, abstract data types, wrappers.
- Increase Competence Set: designing a component to handle more cases—faults—as part of its normal operation.

Interoperability tactics



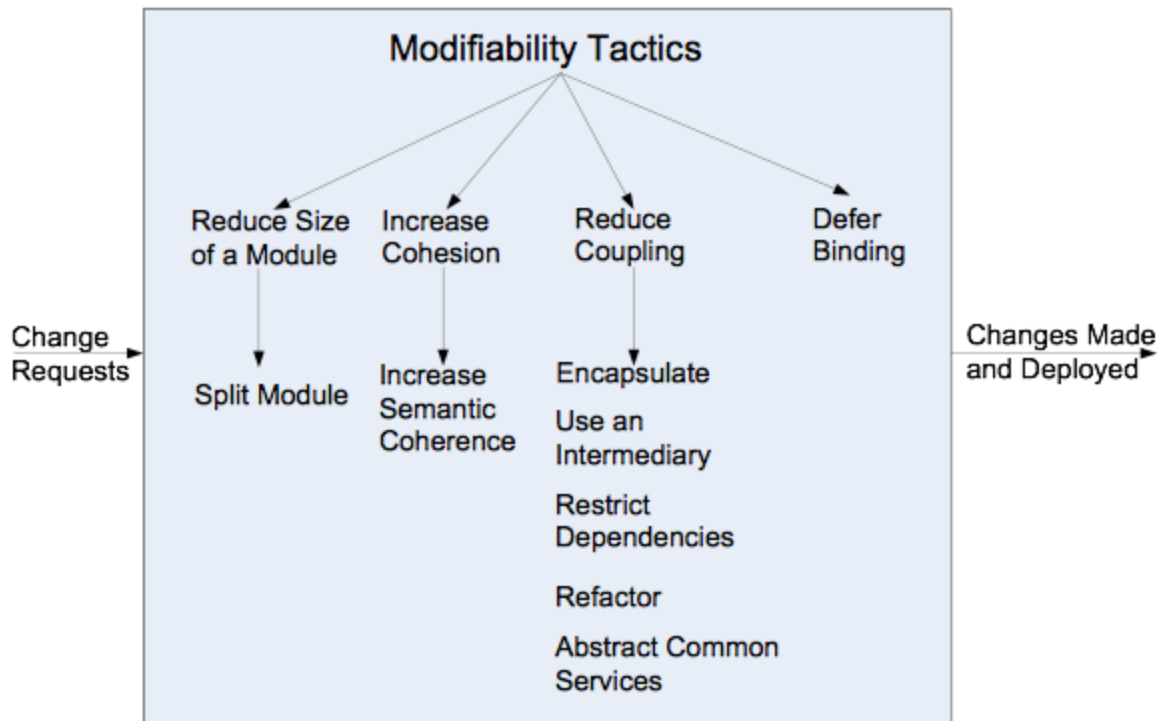
Locate

- Discover service: Locate a service through searching a known directory service. There may be multiple levels of indirection in this location process – i.e. a known location points to another location that in turn can be searched for the service.

Manage Interfaces

- Orchestrate: uses a control mechanism to coordinate, manage and sequence the invocation of services. Orchestration is used when systems must interact in a complex fashion to accomplish a complex task.
- Tailor Interface: add or remove capabilities to an interface such as translation, buffering, or data-smoothing.

Modifiability tactics



Reduce Size of a Module

- Split Module: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

- Increase Semantic Coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module.

Reduce Coupling

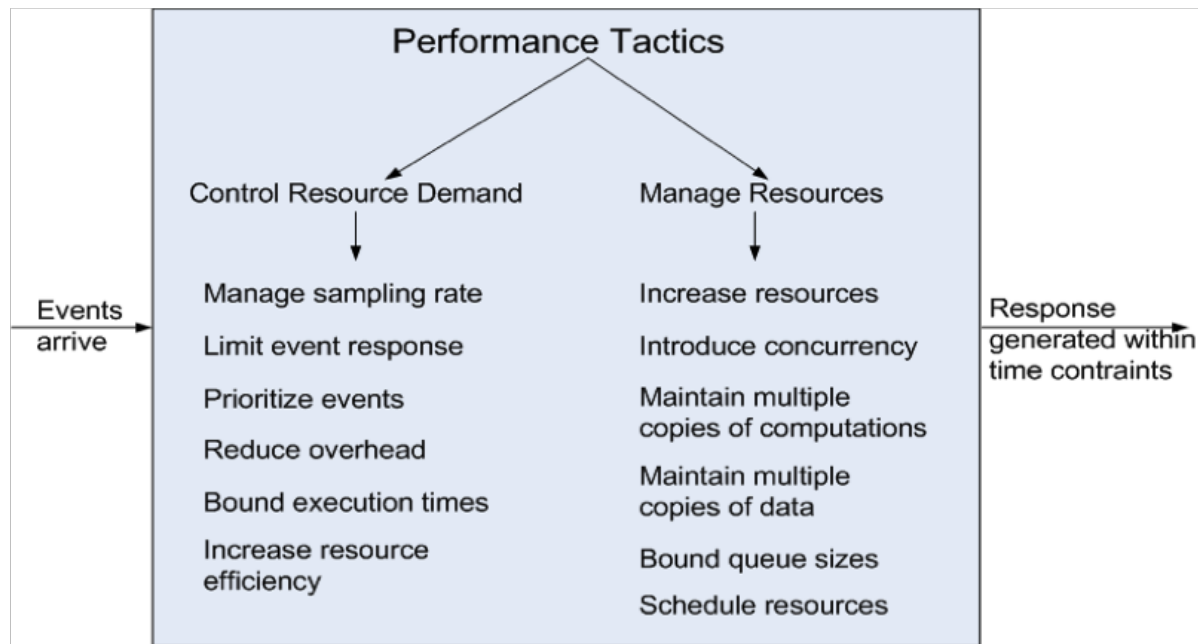
- Encapsulate: Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”
- Use an Intermediary: Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary.
- Restrict Dependencies: restricts the modules which a given module interacts with or depends on.
- Refactor: undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other.

- **Abstract Common Services:** where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

Defer Binding

- allow decisions to be bound after development time

Performance tactics



Control Resource Demand

- **Manage Sampling Rate:** If it is possible to reduce the sampling frequency at which a stream of data is captured, then demand can be reduced, typically with some loss of fidelity.
- **Limit Event Response:** process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed.
- **Prioritize Events:** If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.
- **Reduce Overhead:** The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency.
- **Bound Execution Times:** Place a limit on how much execution time is used to respond to an event.
- **Increase Resource Efficiency:** Improving the algorithms used in critical areas will decrease latency.

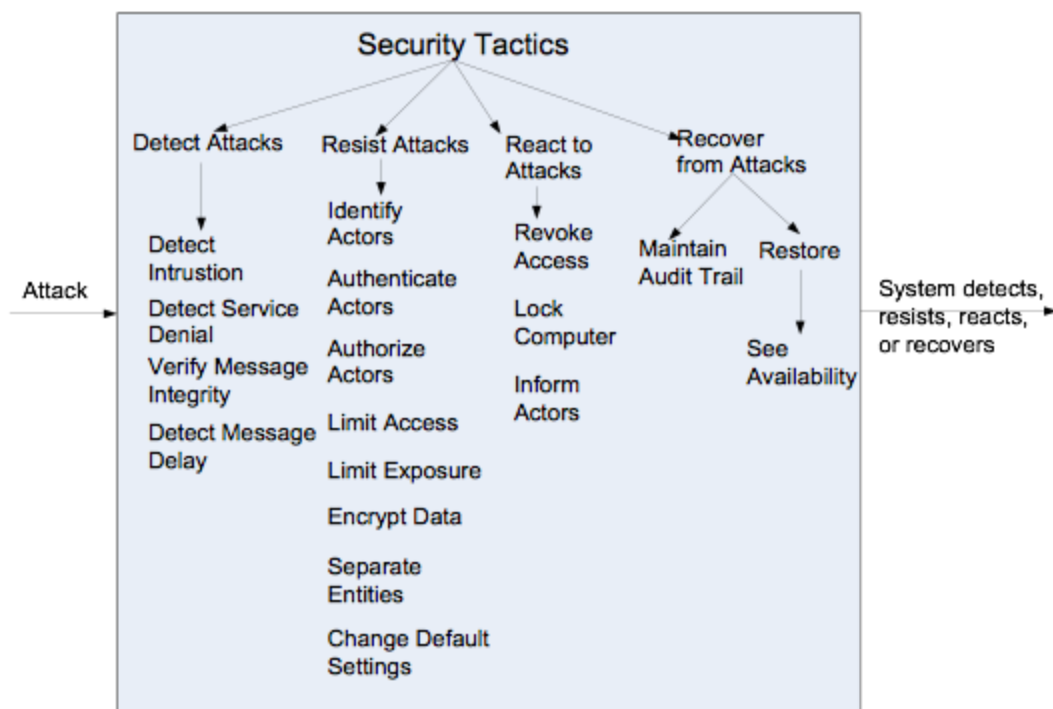
Manage Resources

- **Increase Resources:** Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.
- **Increase Concurrency:** If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by

creating additional threads to process different sets of activities.

- Maintain Multiple Copies of Computations: The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server.
- Maintain Multiple Copies of Data: keeping copies of data (possibly one a subset of the other) on storage with different access speeds.
- Bound Queue Sizes: control the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- Schedule Resources: When there is contention for a resource, the resource must be scheduled.

Security tactics



Detect Attacks

- Detect Intrusion: compare network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database.
- Detect Service Denial: comparison of the pattern or signature of network traffic coming into a system to historic profiles of known Denial of Service (DoS) attacks.
- Verify Message Integrity: use techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- Detect Message Delay: checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior.

Resist Attacks

- Identify Actors: identify the source of any external input to the system.
- Authenticate Actors: ensure that an actor (user or a remote computer) is actually who or what it purports to be.
- Authorize Actors: ensuring that an authenticated actor has the rights to access and modify either data or services.
- Limit Access: limiting access to resources such as memory, network connections, or access points.
- Limit Exposure: minimize the attack surface of a system by having the fewest possible number of access points.
- Encrypt Data: apply some form of encryption to data and to communication.
- Separate Entities: can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”.
- Change Default Settings: Force the user to change settings assigned by default.

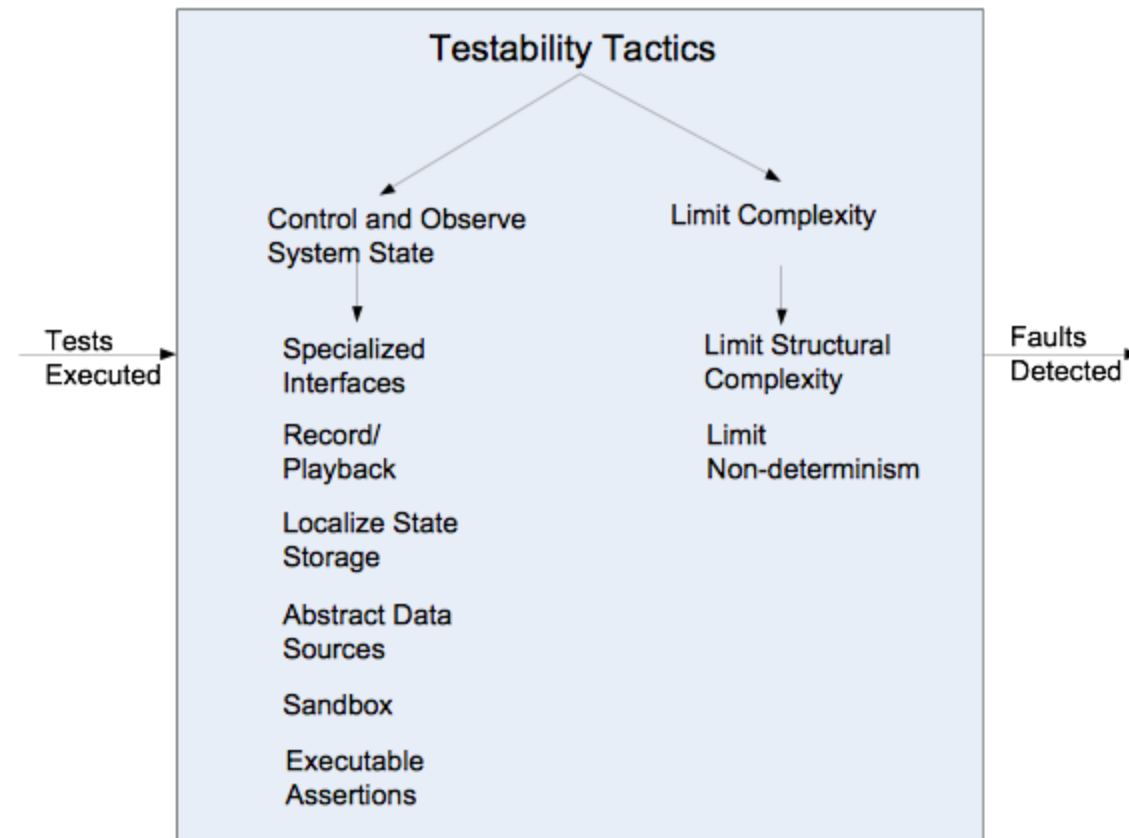
React to Attacks

- Revoke Access: limit access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.
- Lock Computer: limit access to a resource if there are repeated failed attempts to access it.
- Inform Actors: notify operators, other personnel, or cooperating systems when an attack is suspected or detected.

Recover from Attacks

- In addition to the Availability tactics for recovery of failed resources there is Audit.
- Audit: keep a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.

Testability tactics



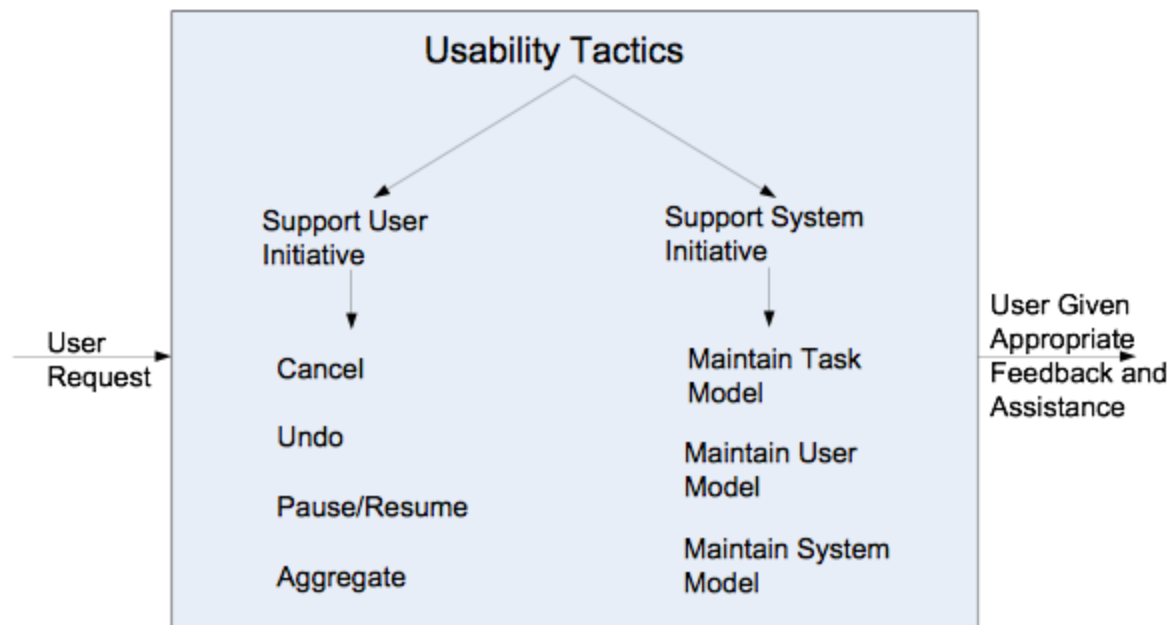
Control and Observe System State

- Specialized Interfaces: to control or capture variable values for a component either through a test harness or through normal execution.
- Record/Playback: capturing information crossing an interface and using it as input for further testing.
- Localize State Storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.
- Abstract Data Sources: Abstracting the interfaces lets you substitute test data more easily.
- Sandbox: isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- Executable Assertions: assertions are (usually) hand coded and placed at desired locations to indicate when and where a program is in a faulty state.

Limit Complexity

- Limit Structural Complexity: avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general.
- Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and weeding them out as far as possible.

Usability tactics



Support User Initiative

- Cancel: the system must listen for the cancel request; the command being canceled must be terminated; resources used must be freed; and collaborating components must be informed.
- Pause/Resume: temporarily free resources so that they may be re-allocated to other tasks.
- Undo: maintain a sufficient amount of information about system state so that an earlier state may be restored, at the user's request.
- Aggregate: ability to aggregate lower-level objects into a group, so that a user operation may be applied to the group, freeing the user from the drudgery.

Support System Initiative

- Maintain Task Model: determines context so the system can have some idea of what the user is attempting and provide assistance.
- Maintain User Model: explicitly represents the user's knowledge of the system, the user's behavior in terms of expected response time, etc.
- Maintain System Model: system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user.

Frameworks

Structuring the system + supporting non-functional concerns

Spring framework

Framework name	Spring framework
Language	Java
Purpose	Application Framework that allows the objects that form an application to be connected. It also supports different concerns through aspect-oriented programming.
Overview	<ul style="list-style-type: none">- The Spring Container connects standard Java objects, or POJOs (Plain Old Java Objects), by using information from an XML file called "Application Context". This is the "Inversion of Control and Dependency Injection" pattern, since object dependencies are injected by the container.- The framework supports several aspects using Aspect Oriented Programming (AOP). These aspects are introduced as proxies between the java objects when the container connects them. Supported aspects include:<ul style="list-style-type: none">- Security- Transaction Management- Publishing object interfaces so the objects can be accessed remotely, for example via Web Services

Structure	<pre> graph TD SC["Spring Container (factory)"] AC["Application Context (xml file)"] JO1["<<java object>>"] JO2["<<java object>>"] Proxy["Aspect-supporting proxy may be introduced here"] SC -.-> <<use>> AC SC -.-> <<create>> JO1 SC -.-> <<create>> JO2 JO1 --- JO2 JO1 -.-> Proxy JO2 -.-> Proxy </pre>
Implemented design patterns and tactics	<p>Patterns</p> <ul style="list-style-type: none"> - Inversion of Control and Dependency Injection Pattern - Factory - Proxy <p>Tactics</p> <ul style="list-style-type: none"> - Availability: Transactions - Testability: Abstract data sources (separate interface and implementation)
Benefits	<ul style="list-style-type: none"> - Excellent tool support - Simple integration with other frameworks such as web UI (Spring MVC, JSF), and persistence (JPA, Hibernate, iBatis) and integration (JMS)
Limitations	<ul style="list-style-type: none"> - Apache License 2.0 - Complex framework

User interface

Java Server Faces framework

Framework name	Java Server Faces (JSF) framework
Language	Java
Purpose	Web application framework that facilitates the development of Web User Interfaces and supports Asynchronous Javascript and XML (AJAX)

Overview	<ul style="list-style-type: none"> - Web Pages (views) are developed as Java Server Pages - The model is represented by a set of Java classes called “Managed Beans” which have perform validations and handle events. In general, there is one Managed Bean per View which forwards requests to other objects in the business layer. - The controller is a servlet which is configured declaratively in an xml file called “faces-config” where navigation rules are described along with the associations between views and model classes. - AJAX-enabled components can easily be added to enrich JSF-based user interfaces
Structure	<pre> classDiagram class facesConfig["<<xml file>>\nfaces-config\n(controller)"] class View["<<JSP>>\nView"] class ManagedBean["<<java class>>\nManagedBean\n(model)"] facesConfig ..> View facesConfig ..> ManagedBean View ..> ManagedBean </pre>
Implemented design patterns and tactics	<p>Patterns</p> <ul style="list-style-type: none"> - Model View Controller
Benefits	<ul style="list-style-type: none"> - Good tool support - Simplifies web development
Limitations	<ul style="list-style-type: none"> - Complex page lifecycle can impact performance and scalability (in case of a large number of clients)

Swing Framework

Framework name	Swing Framework
Language	Java
Purpose	Framework to support the creation of portable local (non-web) user interfaces
Overview	The swing framework provides a library of user interface components including JFrame (windows), JMenu, JTree, JButton, JList, JTable etc... These

	<p>components are built around the Model View Controller and Observer patterns.</p> <p>Components such as JTables are Views and Controllers and each has a corresponding model class (e.g. TableModel).</p> <p>Components allow Observers (called “Listeners”) to be registered in order to manage different events. For example, JButtons allow ActionListeners to be registered as observers so that when the button is pressed a callback method (action performed) is invoked.</p>
Structure	<pre> classDiagram JComponent < -- JButton JComponent < -- JTable JComponent < -- JTree JComponent "*" --> "1" JFrame : children JTree --> "1" TreeModel : model JButton --> "1" ActionListener : listeners class ActionListener { <<interface>> actionPerformed(ActionEvent evt) } </pre>
Implemented design patterns and tactics	<p>Patterns:</p> <ul style="list-style-type: none"> - Model View Controller - Observer - others such as Composite and Iterator
Benefits	<ul style="list-style-type: none"> - Portable (can run on any OS) - Part of Java API - Good tool support
Limitations	<ul style="list-style-type: none"> - Slower than using native UI elements, not same look & feel as native UI elements

OO-Relational Mapping

Hibernate Framework

Framework name	Hibernate
Language	Java
Purpose	Simplify persistence of Objects in a relational database
Overview	Hibernate allows to objects to be easily persisted in a relational database (it supports different database engines). Object - relational mapping rules are described declaratively in an xml called hibernate.cfg file or using annotations

	<p>in the classes whose objects need to be persisted.</p> <p>Hibernate supports transactions and it also provides a query language called HQL (Hibernate Query Language) used to retrieve objects from the database. Hibernate utilizes multilevel caching schemes to improve performance. It also provides mechanisms to allow Lazy Acquisition of dependent objects in order to improve performance and reduce resource consumption. These mechanisms are configured declaratively in the configuration files.</p>
Structure	<pre> classDiagram class HibernateRuntime["Hibernate Runtime"] class hibernateCfg["hibernate.cfg"] class SomeEntity["SomeEntity"] class databaseTable["database table"] HibernateRuntime ..> hibernateCfg : <<use>> SomeEntity ..> databaseTable : <<persist>> hibernateCfg ..> databaseTable : <<persist>> </pre>
Implemented design patterns and tactics	<p>Patterns:</p> <ul style="list-style-type: none"> - Data Mapper - Resource Cache - Lazy Acquisition <p>Tactics:</p> <ul style="list-style-type: none"> - Availability: Transactions - Performance: Maintain multiple copies of data (cache)
Consequences	<ul style="list-style-type: none"> - Complex API - Slower than JDBC (Java Database Connectivity) - Difficult to map to legacy database schemas

MyBatis Framework

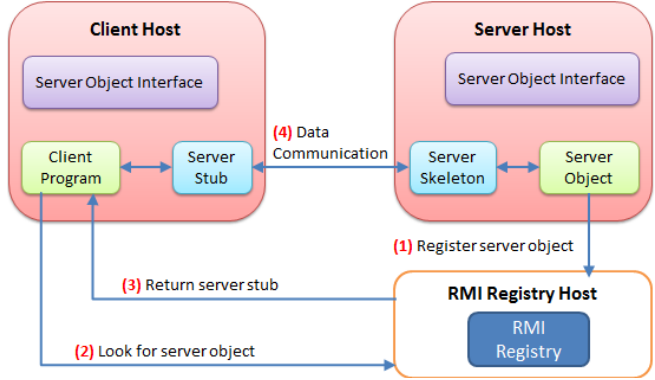
Framework name	MyBatis
Language	Java
Purpose	Persistence Framework
Overview	<p>MyBatis is a persistence framework that allows objects to be persisted in a relational database. Similar to Hibernate, configuration is performed declaratively using XML files or annotations. As opposed to Hibernate, where developers start with an object model, in MyBatis, developers start with a SQL Database and the framework automates the creation of the objects.</p>

Structure	
Implemented design patterns and tactics	Patterns: - Data Mapper
Consequences	- Good for legacy databases - Easy to learn

Remote communication

RMI Application Programming Interface

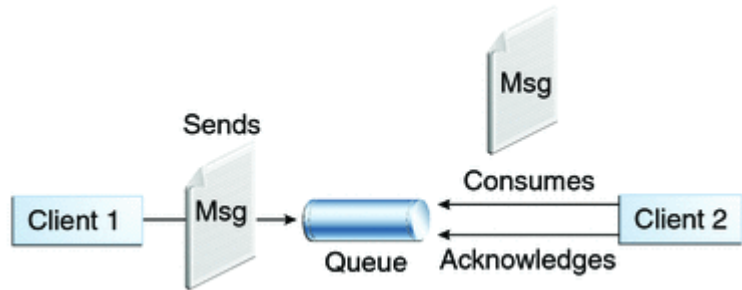
Framework name	Remote Method Invocation Framework
Language	Java
Purpose	Perform Remote Object-Oriented Procedure Calls
Overview	<p>RMI is an Application Programming Interface that performs the object-oriented equivalent of remote procedure calls, with support for direct transfer of serialized java objects and distributed garbage collection.</p> <p>Server objects are registered in an RMI registry (1). Clients then look up for server objects (2) in the registry which returns server stubs (3). Clients communicate with the server objects through the stub and the data is received on the server object side by a server skeleton (4).</p>


Structure	 <p>The diagram illustrates the RMI (Remote Method Invocation) structure. It consists of three main components: a Client Host, a Server Host, and an RMI Registry Host. The Client Host contains a Server Object Interface, a Client Program, and a Server Stub. The Server Host contains a Server Object Interface, a Server Skeleton, and a Server Object. The RMI Registry Host contains an RMI Registry. The interactions are numbered: (1) Register server object (Server Object to RMI Registry), (2) Look for server object (Client Program to RMI Registry), (3) Return server stub (RMI Registry to Server Stub), and (4) Data Communication (Client Program to Server Skeleton).</p> <p>Source: http://lycog.com/distributed-systems/java-rmi-overview/</p>
Implemented design patterns and tactics	<p>Patterns:</p> <ul style="list-style-type: none"> - Broker <p>Tactics:</p> <ul style="list-style-type: none"> - Interoperability: Discover service
Benefits	Part of Java API
Limitations	Requires a particular port to be accessible, which limits access through firewalls

Axis2 Web Services Framework

Framework name	Axis2 (client side)
Language	Java
Purpose	Web Services / SOAP / WSDL engine
Overview	For the client side, Axis2 provides mechanisms that allow a web service to be invoked programmatically. Service invocations can be synchronous or asynchronous.
Structure	Not available (see structure of broker pattern)
Implemented design patterns and tactics	<p>Patterns:</p> <ul style="list-style-type: none"> - Broker
Benefits	- Stable framework
Limitations	- Impacts performance

JMS Framework

Framework name	Java Message Service
Language	Java
Purpose	<p>Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication.</p> <p>The JMS API enables communication that is not only loosely coupled but also:</p> <ul style="list-style-type: none"> Asynchronous: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them. Reliable: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.
Overview	<p>Most implementations of the JMS API support both the point-to-point and the publish/subscribe approaches.</p> <p>A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.</p> <p>In a publish/subscribe (pub/sub) product or application, clients address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.</p>
Structure	<p>Point to point messaging</p>  <pre> graph LR C1[Client 1] -- "Sends (Msg)" --> Q[(Queue)] C2[Client 2] -- "Consumes" --> Q C2 -- "Acknowledges" --> Q </pre> <p>Publish / Subscribe messaging</p>

	 <p>Source: http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html</p>
Implemented design patterns and tactics	Patterns: <ul style="list-style-type: none"> - Messaging - Publisher-subscriber
Benefits	- Part of Java EE API
Limitations	Implementation dependent

Deployment

Java Web Start Framework

Framework name	Java Web Start Framework
Language	Java
Purpose	Provide a platform-independent, secure and robust deployment technology.
Overview	By using a web browser, end users can start java applications and Java Web Start ensures they are running the latest version. To launch an application, users click a link on a page. If this is the first time the application is used, Java Web Start downloads the application. If the application has been previously used, Java Web Start verifies that the local copy is the latest version and launches it or downloads the newest version.
Structure	Not available
Implemented design patterns and tactics	Tactics: <ul style="list-style-type: none"> - Security: Limit access (sandbox) - Performance: Maintain multiple copies of data (cache)
Benefits	- Applications run in a sandbox but can read and write to local files.

	- Since the application is cached, once it has been downloaded startup time is greatly reduced.
Limitations	- First launch may take some time