A **framework** is a programming environment that

**Dictates** programming style and approach. **Solves** some common programming problems. **Owns** and solely manipulates specified resources. **Imposes limits** on other uses of those resources.
Look at what you lose, as well as what you gain!
Frameworks versus IDEs

Integrated Development Environment:
- Editor
- Libraries
- Compilation/testing.
- No limit on what you can do with the IDE.

Framework
- Doesn't have to include an editor or even a library.
- **Has limits:** cannot do just anything with a framework.
- **Has benefits:** once you subscribe to the framework and obey the limits, common tasks become easier.
- **Has detriments:** uncommon tasks may become much more difficult.

It is a very bad bet that a specific framework will get along with another specific programming style.
The concept of exposure

A piece of information is exposed about a framework if it is publicly documented and its use is recommended. Exposure is a **guarantee of continuity**.

Generally, exposed information:
- Is fully documented and supported.
- Is subject to rigorous change control, including orderly feature change and deprecation.
- Persists between major revisions of the framework.

In any framework, much information is not exposed:
- Exact implementation of features.
- Exact details of utilization of external resources.

It is potentially deadly to build systems based upon unexposed information; then:
- Major revisions of the framework can change details **without notice or deprecation**.
- Your software will break.
- Cost of re-engineering may be high or even impractical.

Why is limited exposure desirable?
- Because it limits what the vendor has to support.
- Because it allows freedom of response in changing
other things.

For the vendor,
Expose things you're sure of.
Strategically don't expose things that need improvement.
(Expose features for a fee).
Exposed: very powerful drag-and-drop web controls.
You specify where controls are on a page and what they should do.
Framework builds both client-side (user interface) components and server-side (service) components in tandem (yet another flavor of Jackson's duality principle; this time, between client and server).

Unexposed: nature of REST protocols between client and server, including
Reserved names.
Name mapping between REST interface/variables/methods.
Caveat: it is very risky to try to use ASP.NET and your own REST.
ASP.NET concepts

**control**: a multi-purpose object that is a building block for web-based forms. Can have:
- a user interface component.
- a data model component.
- a protocol component between user interface and data model.

**binding**: the process of associating a control with the resources it controls, e.g., a database table.

**configuration**: the process of specifying the fine-grained behavior of a specific control instance.
A binding is an association between a program object and an external resource. In ASP.NET, one can "bind controls to database tables". This allows the control to edit the object, and to do whatever you wish to allow the user to do.

Supports full CRUD model of database access:

Create
Retrieve
Update
Delete

Result:

a guestbook is one non-custom control.
a shopping cart is one non-custom control.
Configuration

The process of specifying what capabilities a control should exhibit.

Example: the table control:
  Binds to a database table.
  You specify (via configuration)
    Which columns are shown.
    Which columns are editable.
    Which columns have custom presentations (programmed by you).
    The visual style of the table object on the screen.
    Interactions with other controls.
  The control does everything else.
The power of the table control:
The table object is a very complex piece of software.
You customize that software's behavior, but do not touch its code.
Thus, your task is greatly simplified, according to any measure of complexity.
Session objects

ASP.NET is designed for web interaction. There are many transactions within one user "session" (loosely defined as use of one browser on one machine until the browser is restarted).

A cookie in the browser keeps track of each session.

- Instantiated when the browser first comes to the application.
- Erased after a timeout or upon explicit logout.

Controls -- and you -- may keep track of anything you wish in a "Session" object:

- Contents stored on server.
- An associative array (strings -> strings)
- Persists between calls to the server.
- Objects are serialized before being stored in it.

Examples:

- The user's name.
- The user's authorizations and privileges.
Good news and bad news

Monday, December 03, 2012
9:53 AM

Good news: in manipulating and writing controls, you can store whatever you want in the session object.
Bad news: the exact mechanism by which the session data is stored is **unexposed**; you cannot (safely) use your own cookie scheme.

Good news: in manipulating tables, the table control is very powerful.
Bad news: if you want to do anything that it won't straightforwardly do, you more or less have to write your own.

It's easy to swim with the flow, and very difficult (and expensive) to swim against it.
## Frameworks and Capabilities

<table>
<thead>
<tr>
<th>Framework</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP.NET</td>
<td>Easy manipulation of database tables.</td>
<td>Single-vendor, services not reusable in other contexts.</td>
</tr>
<tr>
<td>Ruby on Rails</td>
<td>Highly tested MVC environment</td>
<td>Difficult to extend with new capabilities.</td>
</tr>
<tr>
<td>Symfony</td>
<td>Highly inclusive MVC environment; couples to all known utilities.</td>
<td>Difficult to validate environmental configurations of supporting software.</td>
</tr>
<tr>
<td>Google Apps</td>
<td>Horizontally scalable front-end; cloud-based, highly available backend</td>
<td>Single vendor, not portable to other cloud providers.</td>
</tr>
<tr>
<td></td>
<td>supporting JDO.</td>
<td></td>
</tr>
<tr>
<td>Force</td>
<td>Canned solutions to all major business needs; rigorous security model</td>
<td>Very difficult to extend, costly to operate.</td>
</tr>
</tbody>
</table>
A radical departure from any programming paradigm you have seen before. Not really "a programming language" at all. Instead, a "rule engine".

You specify the rules in Event-Condition-Action format:
- The **Event** to watch for.
- The **Condition** to check for (when the event is seen).
- The **Action** to perform when the event happens, if the condition is true.
ECA modeling

Begin with an object model of **business data**. Define **profiles**, **roles**, and **users** that describe what data state transitions **can** happen:

- **user**: an entity who can modify data.
- **profile**: a set of access privileges for data. what someone "can" do.
- **role**: a business role, mapped to profiles by inheritance. what someone "should" do.

Then define **workflows** that describe what data state transitions **should** happen.

**Caveat**: a workflow contains computational steps and people steps.
Profiles, Roles, and Users

A Profile is a set of user capabilities, e.g., "ability to modify this or that object"

A Role describes management hierarchy. Managers inherit the profiles of the people they manage.

Each User is assigned a (data access) Profile and (optionally) a (business) Role.

End result: hierarchy of responsibility, where upper objects are responsible for lower ones.
A simple picture

Responsibility
Manager
Role

People
Sales
Billing
Happy
Profiles

Instance of a class

A sale
An example of ECA design

Case: purchase of an item

Three entities: customer, billing department, shipping department.

An item can be
ordered (but not paid)
paid (but not shipped)
shipped
cancelled
People control states:
billing controls "paid"
shipping controls "shipped" and "canceled"
billing controls "canceled"
Data, profiles, and roles

Order record contains **state variables:**

- **paid:** 'true' if order is paid.
- **shipped:** 'true' if order is shipped.
- **returned:** 'true' if order returned.
- **canceled:** 'true' if order canceled.

Profiles: **customer, billing, shipping**

**customer** creates Order record, which is partially writeable to **billing** and **shipping**.

billing can change 'paid' or 'cancelled' to true in order record, but cannot change the order itself.

shipping can change 'shipped' or 'returned' to true in order record, but cannot change whether it is paid.

customer can only see **personal** records;

billing and shipping can see **all** records.
Note that state is "factored" into variables changed by different principals.
You might ask, why not create a variable "state" that can be "shipped", "paid", etc?
A: then that variable would be writable to all entities, even those that have no "business" writing it.
Sarbanes-Oxley act of 2002: businesses are accountable for enforcing regular business processes.
Factoring an object

Decompose the object into a state vector of state components that are writeable to different entities. Basic principle of factoring: a different role requires a different state component: do not allow two roles access to the same component.
Example: sales roles
Implementing ECA in Force.com

Create a data object containing appropriate state.
Set object to be created by customer, and viewed by billing and shipping.
Grant limited rights to modify state to billing and shipping.
Assign profiles to people who are customers, shippers, or billers.
Also need a "manager override" in case things go wrong.
A complete shopping application
Monday, December 03, 2012
5:20 PM

The last thing is to modify the peoples' views of objects.
  Shipping sees only things that aren't shipped.
  Billing sees things that aren't paid.
  When the state changes, the object disappears.

Thus, the whole ordering/billing/shipping was just programmed without writing a line of code.
Access rights in force.com

Each profile has overall access rights for each object. Each object field has access rights for each profile as well.
So, one can make an object viewable, and part of it editable, part invisible.
Watch out: access rights for objects are properties of the profile, while access rights for fields are properties of the object field itself, inside the object!
Profiles 1 and 2 share a state variable. They can be changed by the user and are linked together.
Events and actions
In Free Force.com, the only events we have are "workflows".
A "workflow" causes something to happen as a result of a user action, e.g., changing an object record.

Several kinds of workflow actions
Update field: set a field in an object to a new value based upon other changes in the object.
Send email.
Schedule a task (for a user).
Schedule an "approval" process.
A simple workflow

Event
Add new item
Condition: Always
Action: Update subtotal
A workflow is:

Roughly the same as a database trigger
....but for people!
If something happens, someone (or some group) in the organization either
gets a new task.
loses a task.
etc.
"We bring order to chaos"
  Tie events to responsibilities.
  "If someone buys something, someone else sends a bill."
  "If someone pays, someone else ships."
A queue is a list of tasks that are shared by a group of users (with the same profile).
Can assign a task to either a user or a queue.
The workflow for the example

Wednesday, April 20, 2011
5:10 PM

[Diagram showing workflow steps with labels for each step and person responsible]
Approvals

The most complex feature of Free Force.com: approvals

Often, business logic requires a complex set of approvals for a record.

E.g., for different levels of expenses.

Approval process requires definition of approvers for data in an object.

A potentially complex, multi-state system of approvals is possible.

Approvals are the only way to "lock" an object in Free force.com.

Approvals are not composite... they cannot be constructed from primitives already available in force.
The strengths of force

Security:
  Security model is based upon the role model, which is based upon corporate policies.

Multi-level programming:
  Corporate policies specify roles, via an interface accessible to managers.
A problem of levels

In regular web frameworks, security and integrity must be coded into the front-end code.
In force, security rules are coded into the back-end; the database model won't allow roles to do anything they are not allowed via policy;
It is not possible to violate company policy by screwing up front-end code.