Force provides a radically different model of "programming"
  Web forms.
  Privilege-based access.
  Event-Condition-Action (triggers)
Force claims that "no programming is necessary."
What other alternatives are there to "programming"?
We presume that the goal of programming is to enable people to get work done. We presume that there is a very clear concept of what it means to get work done. We treat people and programs as equal partners in this process. This is a fundamental paradigm shift.
The policy/software divide
Management determines policies: how a business or other enterprise will operate. Engineers develop software coherent with policies. How do these entities communicate?
The COBOL language had some unique properties:
  Technical professionals write code.
  Code is readable to can be verified by a non-technical person.
  Thus, management can ensure that policies are obeyed... without even knowing high school algebra...(!)
There is value in a representation that is
  Crafted by technical experts.
  Verifiable by non-technical management.
There is value in a representation in which
  Human roles are represented along with program roles.
  Workflows can be represented, critiqued, and optimized.
IBM's dream

Management describes policies in a precise form.
Policies are converted directly into programs.
   (There's no programmer.)
Program infrastructure is self-healing.
   (There's no system administrator or database administrator.)
Steps toward the dream

Describing policies (BPMN).
Converting them to programs (BPEL).
Executing those programs: orchestration and choreography.
Two prevalent approaches to business modeling

**BPMN (Business Process Modeling Notation):** draw a process, irrespective of whether it's executable by a computer or not.
- Intrinsically, a diagramming method that cannot be converted into an executable program.
- People play roles in process.

**BPEL (Business Process Execution Language):** define an executable part of a business process, separate from the human parts.
- Intrinsically, a programming language for creating services that implement a business process.

Basic strategy:
- Managers create BPMN
- Programmers convert this to BPEL
- All else is done via magic.
  - Axis services.
  - Websphere services.
  - The plumbing to make them work together.
Business Process Modeling

Describes "how a business works".
Has human and IT elements.
Reference:

http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf
Business Process Modeling Notation (BPMN)

A way of graphically describing a business process. Depicts workflow.
Can be created via the Eclipse SOA Tools Project (STP).
File/New.../New project.../General.../Project
File/New.../Other/BPMN Diagram
Rounded boxes are processes:
Something that happens.
Can receive messages or send messages.
Can have sub-processes (that are sub-boxes).
Page icons represent data

![Page icon with handwritten data]
Connectors

Solid arrows represent precedence:

\[ \text{A} \rightarrow \text{B} \quad \text{means A is completed before B begins.} \]

Dashed arrows represent messages/data:

\[ \text{A} \rightarrow \text{B} \quad \text{mean A sends a message to be used by B.} \]

Dashed arrows with open arrowheads indicate inputs to a process.

\[ \text{A} \rightarrow \text{B} \quad \text{means A is input to B.} \]
Events

Wednesday, March 31, 2010
10:16 AM

Circles represent events

Start event: begins a process.  ○
End event: stops a process.  ○ hold
Intermediate event  ○ two circles

```
- - - > [ ] - - - >
"mail box"

---

- - - - >
"observable exchange"
```
Decisions are notated by diamonds:

Watch out:
Both sides of a fork have to have a symmetry
  If one side sends a message, the other one must as well.
  Or the recipient will deadlock.
Sub-processes

Can be described via nested boxes.
There are two entities. They exchange information. Exchange isn't modeled.
Processes are never this simple!

- Multiple participants ("pools").
- Timing requirements ("events").
- Loops ("negotiations").
- Exceptions and errors.
Pools represent participants (entities) involved in an event.

(Some authors call this a "swimlane diagram".)
Loops represent repeated tasks or negotiations

Try for appointment until you succeed!
Some helpful notations:

- A loop
- A time dependency (wait)
- An error event
Low and high-level depictions:

High level:

Low level:

Source:

http://www.bpmn.org/Documents/Introduction_to_BPMN.pdf
Why BPMN is important:
- Makes business processes explicit.
- Defines roles as pools.
- Depicts interactions between roles, and precedences between interactions.
- Exhibits the true complexity of processes.
- Documents failure modes.
- Can be used to analyze and tune processes.
The BPMN diagram tells us exactly what impact IT has upon the business. The diagram can be analyzed for the impact of software failures.
Process versus orchestration

BPMN: defines business processes
Orchestration: assigns (internal) processes to responsible parties.
Orchestration language: WS-BPEL
(Web Services Business Process Execution Language)
The dream of WS-BPEL

You define services via, e.g., AXIS (or by hand (gasp!)). This generates a WSDL file for each service, which contains Xschemas for input and output. BPEL "orchestration" defines how services interact, and which services are "bound" to each activity, by referencing (AXIS-generated) WSDL files for each service.

A BPEL "engine" (e.g., IBM Websphere, Apache ODE) executes the orchestration file and creates a website that supports your business process. With the proper BPEL, a web "solution" pops into existence, with interfaces for all involved people!
WS-BPEL

An "XML Programming Language"
An open standard
Derived from IBM and other process standards.
Executed by several engines.
Reference:
http://www.eclipse.org/stp/b2j/docs/tutorials/wsbpel/wsbpel_tut.php (I am fairly sure the example is incorrect!)
Contents of a BPEL description

Process descriptions within a pool in BPMN.
Bindings for particular services.
Note that unlike BPMN, BPEL has non-graphical components, including the service bindings themselves.

A BPEL process describes activities for one pool of a BPMN diagram.
Parts of BPEL

XSD (XML Schema Definitions) define types to be used. (internally, uses XSL and XSLT!) XPATH specifies data queries and boolean operators when choices are made.

WSDL (Web Service Definition Language) defines services you may have created (using, e.g., AXIS). BPEL (Business Process Execution Language) glues together WSDL's to create a business process.
A BPEL process

```xml
<process name='foo'>
    <!-- BPEL "activities" -->
</process>
```

An activity can be
- Receiving a message.
- Sending a message.
- Querying a service.
- A sequence of activities.
- A set of parallel activities.
- For, while, do... until loops.
- If-else constructions.
- Pick: switch on first message received.

Aside: BPEL is not based upon lambda-calculus, but upon an alternative pi-calculus, which is in turn based upon petri net semantics.

BPEL is an **orchestration** language:
- It says what components should do in order to interact.
Scope

By default, all variables are global to a process.
One can use `<scope>...</scope>` declarations to keep
variables, exception handling, etc local to a region.
Variables

Kinds of variables:

- **messageType**: appropriate for a SOAP message, declared in WSDL.
- **type**: appropriate for local processing, declared in XSD.
- **element**: a piece of XML.
Receiving requests from others

Wednesday, March 31, 2010
12:59 PM

<receive name="ReceiveRequestFromPartner"
    createInstance="yes"
    partnerLink="ClientStartUpPLT"
    operation="StartProcess" />


Taking this apart
createInstance: create a new process when this happens.
partnerLink: which partner sends you the message.
operation: what to do when you get it.
<reply name="ReplyResponseToPartner"
    partnerLink="ClientStartUpPLT"
    operation="StartProcess" ... />


Taking this apart:

  partnerLink: who to send it to.
  operation: what to do when it's done.
Two forms of valid requests:

a. Request/response:

```
<invoke name="RequestResponseInvoke"
   partnerLink="BusinessPartnerServiceLink"
   operation="RequestResponseOperation"
   inputVariable="Input"
   outputVariable="Output" />
```


Taking this apart

- partnerLink: who to contact.
- operation: what to do on the other side.
- inputVariable: what to send.
- outputVariable: where to put what you get back.

This is an RPC call: request/response.

Appropriate use:
- Query by key

a. One-way:

```
<invoke name="OneWayInvoke"
   partnerLink="BusinessPartnerServiceLink"
   operation="OneWayOperation"
   inputVariable="Input" />
```


Taking this apart:

- partnerLink: who to contact.
operation: what to do on the other side.
outputVariable: what to send.

Appropriate use:
Storing data.
Sequencing means doing things in order.

```xml
<sequence name="InvertMessageOrder">
  <receive name="receiveOrder" ... />
  <invoke name="checkPayment" ... />
  <invoke name="shippingService" ... />
  <reply name="sendConfirmation" ... />
</sequence>
```

Conditionals

<if name="isOrderBiggerThan5000Dollars">
  <condition>
    $order &gt; 5000
  </condition>
  <invoke name="calculateTenPercentDiscount" />
</if>  
<elseif>
  <condition>
    $order &gt; 2500
  </condition>
  <invoke name="calculateFivePercentDiscount" />
</elseif>
<else>
  <reply name="sendNoDiscountInformation" />
</else>
</if>

<while>
  <condition>
    $iterations < 3
  </condition>
  <invoke name="increaseIterationCounter" ... />
</while>


and

<repeatUntil>
  <invoke name="increaseIterationCounter" ... />
  <condition>
    $iterations > 3
  </condition>
</repeatUntil>


True use of iteration: evaluating multiple alternative services.

This is not a calculation language.
It is a glue language.
Best use of iteration: try a set of services until one answers.
<forEach parallel="no" counterName="N" ...>
    <startCounterValue>1</startCounterValue>
    <finalCounterValue>5</finalCounterValue>
    <scope>
        <documentation>check availability of each item ordered</documentation>
        <invoke name="checkAvailability" ... />
    </scope>
</forEach>
Parallel execution

<flow ...
  <links> ...
  <documentation>
    check availability of a flight, hotel and rental car concurrently
  </documentation>
  <invoke name="checkFlight" />
  <invoke name="checkHotel" />
  <invoke name="checkRentalCar" />
</flow>

During a parallel computation, you may want certain phases to wait for others. A "link" is a boolean variable that becomes true under specified conditions. This controls whether a parallel process can continue or not.
<flow ...>
  <links>
    <link name="request-to-approve" />
    <link name="request-to-decline" />
  </links>
  <receive name="ReceiveCreditRequest"
    createInstance="yes"
    partnerLink="creditRequestPLT"
    operation="creditRequest"
    variable="creditVariable">
    <sources>
      <source linkName="request-to-approve">
        <transitionCondition>
          $creditVariable/value &lt; 5000
        </transitionCondition>
      </source>
      <source linkName="request-to-decline">
        <transitionCondition>
          $creditVariable/value &gt;= 5000
        </transitionCondition>
      </source>
    </sources>
  </receive>
  <invoke name="approveCredit" ...>
    <targets>
      <target linkName="request-to-approve" />
    </targets>
  </invoke>
  <invoke name="declineCredit" ...>
    <targets>
      <target linkName="request-to-decline" />
    </targets>
  </invoke>
</flow>
Taking this apart
request-to-approve: a flag that is true if the request should be approved.
approveCredit: a service call that will be invoked if request-to-approve is true.

A link is a variable that is set entirely for its side-effects.
The side-effect of a link being true is invocation of a specific service.
Join conditions

Join conditions: operation is invoked if one target is present:

```xml
<flow ...>
  <links>
    <link name="request-to-approve" />
    <link name="request-to-decline" />
    <link name="approve-to-notify" />
    <link name="decline-to-notify" />
  </links>
  <receive name="ReceiveCreditRequest"
    createInstance="yes"
    partnerLink="creditRequestPLT"
    operation="creditRequest"
    variable="creditVariable">
    <sources>
      <source linkName="request-to-approve">
        <transitionCondition>
          $creditVariable/value < 5000
        </transitionCondition>
      </source>
      <source linkName="request-to-decline">
        <transitionCondition>
          $creditVariable/value >= 5000
        </transitionCondition>
      </source>
    </sources>
  </receive>
  <invoke name="approveCredit" ...>
    <source linkName="approve-to-notify" />
    <targets>
      <target linkName="request-to-approve" />
    </targets>
  </invoke>
  <invoke name="declineCredit" ...>
    <source linkName="approve-to-notify" />
  </invoke>
</flow>
```
Either decline-to-notify or approve-to-notify gets set. When either one is set, notifyApplicant gets called.

There is no sequence to whether decline-to-notify or approve-to-notify gets set first.
The whole interaction is asynchronous:
   The event notifyApplicant gets called when the arrival event is approve-to-notify or decline-to-notify.
Fault Handling

<faultHandlers>
    <catch faultName="BookOutOfStockException"
           faultVariable="BookOutOfStockVariable">
        ...
    </catch>
    <catchAll>...</catchAll>
</faultHandlers>

Scopes

Can use scopes to create local:

Variables

Fault handlers

```xml
<scope>
  <faultHandlers>
    <catch faultName="xyz:anExpectedError">...</catch>
    <catchAll><![-- deal with other errors --]>
      ...
    </catchAll>
  </faultHandlers>
  <sequence> <![-- or flow -->]
    <![-- do work -->]
  </sequence>
</scope>
```

Waiting for mutually exclusive events

<onMessage partnerLink="buyer"
operation="inputLineItem"
variable="lineItem">
<!-- activity to add line item to order -->
</onMessage>

<onMessage partnerLink="buyer"
operation="orderComplete"
variable="completionDetail">
<!-- activity to perform order completion -->
</onMessage>

<onAlarm>
  <for>'P3DT10H'</for>
  <!-- handle timeout for order completion -->
</onAlarm>

Taking this apart:
  onMessage: first one that is received is processed.
  onAlarm: handles timeout events: 3 days 10 hours.
Critique

Much slower execution time than Java glue...
But glue program here is shorter!

Very general:
Responses can be whole web pages (servlet invocations).
Requests can be AJAX.

But: there is a very serious limit
BPEL isn't "reusable".
The concept of a subroutine does not exist.

Using BPMN, one can define a process.
Using BPEL, one can refine a pool into a program.
This is the policy/program divide in action.

Solution: look at interactions as subroutines, rather than steps. => Web Service Choreography.