Directed Evolution for Business Automation

Abstract

We need to move away from data-centric programming frameworks toward business-centric frameworks. Business programmers are saddled with a large amount of unnecessary record-keeping and data moving details that should be relegated to the run-time system. Our traditional algebra-based syntax is limiting our thinking and our productivity. Current XML-based notations are better suited to being object languages, not source languages. We need more expressive notations adapted to the business automation problem space. We need a smarter run-time that removes data handling from the problem space. This paper describes a prototype. The prototype is a test bed for notations and run-time concepts which address these goals.

Preview

A programming framework consists of a language (a coordinated set of notations) and a run-time (the thing we are programming). While we traditionally treat the tooling as if it exists outside the framework, there is strong coupling. When we direct the evolution of our programming frameworks, each evolutionary step needs to:
1. Make the programming notations more expressive.
2. Make the run-time we are programming more powerful.
3. Make the tools smarter.
A breakthrough in productivity can result from the synergistic effect of addressing a number of aspects simultaneously.

To illustrate some of the evolutionary steps, this paper describes an experimental prototype. The prototype includes a business automation language consisting of five notations which address plans, procedures, dialogs, views, and ontology. The notations are not coupled to specific implementation technologies, but they do assume a run-time system which transparently coordinates parallel and distributed processing, automatic persistence, messaging, workflow management, and resource management. Record-keeping tasks are relegated to the run-time so that the programming effort is focused on describing plans and procedures. The development environment is focused more on the structure of the entire code base and less on the structure of the individual module.

When we move the focus from data processes to business processes there are consequences affecting the education and training of business programmers. Business programmers will need to know less about optimizing data processes and more about optimizing business processes. We can expect a greater separation between the knowledge and skills required for engineering software infrastructure versus the knowledge and skills required for optimizing a large scale business architecture.

This paper identifies some problems with current business automation frameworks and demonstrates how those problems might be eliminated via a more business-centric language and a more powerful run-time. We illustrate solutions using a hypothetical framework called "Hum." Hum exists as a partially implemented prototype. It is not a product. It is a test frame for experimenting with various engineering hypotheses.

How do we make a programming framework more productive?

How does software scale?
- Is there a Moore's law for software?

Moore's law for hardware capability is based on the number of components or bits can be packed onto a surface. Is there an capability model for software? There seems to be a kind of diminishing returns with today's software technologies. I don't know of any equivalent for software, but the way that software gains capability is probably not as wonderful as Moore's law. [conjecture] On the other hand, it is probably better than the cube law governing a pile of sand. If you have a pile of sand and you need another pile twice as deep, you will need eight times as much sand.

Lacking a capability model to direct our framework design efforts, we can nevertheless employ some obvious strategies. We simply have to do without a strong model to predict the impacts. The Hum prototype attempts to create a more capable framework and a more productive programming environment by employing three relatively obvious strategies.
1. Make the programming notations more expressive.
2. Make the thing we are programming smarter.

3. Provide automated assistance.
The next three subsections discuss how each strategy is applied to the business automation context.

Make the notations more expressive.

As a college student, I accumulated six years of formal training in applied mathematics. I am quite comfortable with mathematical notations. In spite of that, I believe that using algebra as the base syntax in languages such as C, C++, Java, and Python creates ergonomic hazards that limit creativity and reduce the expressiveness of those languages.

In the 1970's, I adopted the goal of enabling natural language as the base syntax for a programming language. I felt that using natural language as the base would reduce the barriers between analysis, design, and code. My first attempts were like pseudo-code. A few experiments showed me that the productivity was only a little better than COBOL. I was looking for at least a factor of three, so this first attempt failed to make the grade.

In the 1980's, I attempted a more object-oriented approach and invented something like a text version of James Martin's Action Diagram notation. [Martin01] I was also inspired by Smalltalk's ability to obviate persistence by storing the data in a virtual memory. I built a prototype interpreter using Turbo-Pascal, but the result was no better than Action Diagrams and therefore not a break-through.

In the 1990's, I experimented with diagram-driven approaches. I found that diagrams lack the band-width to be productive notations. A diagram by itself is just a "stage prop." It may be a useful communication device, but it is not an engineering document until you add substantial text or code to describe the intended functionality. I decided that it would be more productive to derive a diagram from statements rather than attempting to derive statements from a diagram. This led to experiments with simulated annealing as a technique enabling automatic layout.

Around 2001, I discovered a way to synthesize several ideas into a working whole. The resulting framework allows one to author complex systems using written English as the base syntax. [international] The natural language statements become executable utilizing a simple trick.

The simple trick is to recognize that the nouns in a natural language statement represent variables. When a statement is interpreted by another person, that person will associate a real-world entity with each noun. Thus, a natural language statement is like a method signature with the nouns indicating the arguments. A software system can be organized to interpret natural language statements as if it were working with method signatures and method calls with the arguments indicated by the nouns.

In the case of a software system, we need to assure that the association between nouns and data elements (entities, attributes, categories) is highly predictable. While we can use written natural language as the base syntax, we still have the constraints seen in any programming environment. A noun must be defined before it can be used. A noun's data value must be populated before it is used as an input.

My own experience is that the guidance of a software assistant is all but essential to programming in a natural language. While I can simulate the assistant in my head while writing programs in an ordinary text editor, real productivity requires a tool with capabilities like the smart editors in Eclipse and equivalent development environments. [IDE]

Make the thing we are programming smarter.

If we make the system that we are programming smarter, we can create more business capability in less time. For example, Microsoft Access provides a transaction-processing framework. This framework provides a kind of automated persistence. It automatically produces the actions connecting the user interface to a database. The programmer does not need to specify the details. A transactional application can be built with a fraction of the lines of code.

The Hum framework incorporates similar knowledge concerning persistence. It knows what to persist, how, and when. Persistence occurs automatically and transparently when certain business events occur at run-time.

The Hum framework also has a built-in knowledge of resource management and resource accounting. While most of this accounting occurs without programmer intervention, there are a small number of statement types that cue the resource accounting. The interactive development environment (IDE) will notify the programmer when it detects resource usage that requires some accounting but the resource usage measures have not been specified by a bill-of-material or the procedure.

I believe that the potential exists in every programming framework for each domain-specific component including the notation, the IDE, and the run-time to individually double programmer productivity. The total synergistic effect may be the product of these individual effects producing a total productivity improvement exceeding a factor of eight. I think that most of the productivity effects are enabled by making the run-time smarter. With a smarter run-time, the language notations become deeper. More capability can be invoked with fewer statements. With knowledge of the run-time, the IDE can also leverage the ontology to detect missing statements and potential contradictions. Diagrams designed to reveal the structure of the ontology and other visual aids become be more useful as they incorporate knowledge about the run-time into the visual cues and structures.

Provide automated assistance.

The evolutionary goal for Hum is to move toward an environment where instead of the IDE advising the programmer, the programmer will advise the IDE. The IDE coupled with the run-time will already know how to provide routine data processing services including record-keeping (persistence), messaging (actor-to-actor communications), and bookkeeping (resource accounting). The IDE will need no advice in the data processing domain. In the business domain, the IDE coupled with the run-time will already know how to implement standard business process patterns. It will contain as much or perhaps even more business knowledge than data processing knowledge. The programmer will advise the IDE about how a particular enterprise differs from the standard pattern.

I characterize a "4GL" as an environment that incorporates knowledge about data processing. I characterize a "5GL" as an environment that incorporates knowledge about the external world. At this time, Hum is only a "4GL" not a "5GL." But it could be a stepping stone toward a fifth generation environment. Hum provides the means to write ontologies where the operational meaning, not just the static meaning, of a vocabulary is captured. This enables smarter tools because it provides a deeper structure relating to the intent of the code base.

For example, a future evolution might add programming tools that enable a programming team to edit and extend the standard ontology with "deltas" to produce customized (specialized) solutions. When the base code is enhanced, the tool can reason about potential impacts on the deltas. Since the customization is represented as deltas, the usual problem where custom code freezes the base code can be mitigated.

Additional tools may address the problems inherent in merging and/or interfacing ontologies. Different communities will accidentally use the same nouns but with different operational meanings and new nouns, but with meanings that overlap existing nouns. While we can paper over these problems by writing "wrappers" or "adapters" that translate between vocabularies, we are treating the symptom, not the cause.

The near-term goals for the Hum IDE, the Programmer's Assistant, are relatively prosaic. All modern IDE's will identify statements that are suspect at a semantic level in the current program context. The Hum Programmer's Assistant assumes these capabilities also. For example, a noun value must be populated before it is used as input. The statement containing the noun may comply with the syntax rules, but the program will not be executable until some missing statement is added to populate the noun value.

A later section entitled "The Programmer's Assistant" describes specific features that support programming tasks in the Hum framework. These are vaporware features currently under development as part of my on-going experiment.

Coordinating a network of collaborating actors.

One of our three strategies is to make the programming notation more expressive in the business automation problem space. This section demonstrates notations that are business centric and based on natural language syntax.

Consider what we always need when we attempt to coordinate the actions of a large number of people. First, we need a goal. Given the goal, we need a plan. Given the plan, we need to assign specific actions to various people. Different people have different skills (role knowledge), so we can sort out the assignments according to the role knowledge that the people have. The role knowledge can be codified in standard work instructions (SWI).

Using our software vocabulary, standard work instructions are simply procedures. Procedures are sequences of statements that are interpreted (executed) by an actor that "understands" those statements. They are written in that actor's vocabulary. If the actor is a numerically controlled machine or robot, the statements might be in or translatable to an appropriate machine vocabulary. If the actor is a human, the statement must still be in a vocabulary that is appropriate to the person's training.

But first we need a goal and a plan. That means that we need a notation that expresses goals and plans. The next section demonstrates that notation and how it is executed.

Planning—Goals, Preconditions, and Plan Trees

One of the easiest ways to make a plan is to start with the *end goal*, identify the *preconditions* that are the prerequisites of that goal, and work backwards through the preconditions of the preconditions forming a *plan tree* as we go. Eventually we reach a point where all of the "leaves" of the plan tree have no preconditions. At that point, the plan tree is complete.

Task frames define the nodes of a plan tree.

Here is a relatively simple example. Imagine a regulated utility that provides electricity and gas services to consumers. Imagine what happens when a customer moves from one apartment to another and both apartments are inside the utility's service area. The utility needs to coordinate a number of actions involving several business processes, several internal organizations, and several external parties. [disclaimer]

The task frame below identifies the end-goal as a *post condition*. Each precondition indicates a *sub-goal*. The statement in the *action* section cannot be executed until all of the preconditions are met. After the statement in the action section is executed, the post condition is asserted. Since this particular task frame is at the root of the plan tree, when that post condition is asserted, the end-goal is met.

The task frame notation uses "Post" as a shorthand for "Post Condition" and "Pre" as a shorthand for "Precondition." Comments are enclosed in parentheses. Known nouns are underlined. [underlines]

```
Task: Move customer billing from old-residence to new-residence.
Post: Customer billing moved from old-residence to new-residence.
Pre:
. Old account closed for customer at old-residence.
. New account started for customer at new-residence.
Action:
. None.     (The preconditions define the post condition.)
```

To complete the plan tree we need to add task frames that show how the preconditions are satisfied. Before we start a new account we need turnover readings.

```
Task: Start new account for customer at residence.
Post: New account set up for customer at residence.
Pre:
. Meter readings posted for turnover-date at residence.
. Old account closed for prior-customer at residence.
Action:
. Mailer:  Send welcome-note to customer's new billing-address.
```

Similarly, to close an account we need turnover readings.

```
Task: Close old account for customer at residence.
Post: Old account closed for customer at residence.
Pre:
. Responsibility for residence assigned to landlord or new-tenant.
. Meter readings posted for turnover-date at residence.
Action:
. Billing:  Send final-bill for account at turnover-date
. . to customer's new-billing-address.
```

```
Task: Read meters for turnover-date at residence.
Post: Meter readings posted for turnover-date at residence.
Pre:
. None. (Metering will wait until the turnover date.)
Action:
. Metering: Read residence meters on the turnover-date.
```

Assumptions.

Some preconditions may be assumed, but we should still document those that are not future-proof. For example, we may have a precondition which can be assumed to be always true based on a business policy or some factor that might change in the future. The annotation "[assumed]" is attached to these preconditions so that the IDE knows that no task frame is needed, and future readers are reminded that the plan depends on that assumption.

Natural language can be our base syntax.

COBOL was designed to make programs easy to read by using a syntax based on natural language (English). While COBOL has many detractors, and many programmers find it tedious, it was for some time, the most successful programming framework because it modeled the record-oriented reality faithfully, and, when written carefully, it is readable and easily understood. A programming framework for today's business environment should also be based on natural language syntax. In fact, a simple trick makes it possible to create a syntax that is easily translated and more or less open-ended.

The simple trick is to ignore the structure of a sentence. For our purposes, we do not need to identify subject, verb, object, and qualifiers. The sentence, in its entirety, is a "method signature." We really only need to know which words are nouns because they identify the data. Once the nouns are identified, written natural language statements become machine-readable and we may use natural language statements as the base syntax in our programming notations.

The nouns in a statement identify variables to be instantiated at run-time. The values assigned to those variables are held on a *blackboard* associated with each run-time context. When messages are transmitted between actors, the noun values are populated from the blackboard. Each actor has its own blackboard containing the noun values for the current task.

Dictionary frames define nouns and their relationships.

The terms that are underlined in the various frames indicate the nouns that are known to the IDE. Nouns are identified via *dictionary frames* and some naming conventions. The definition of a noun may be the cumulative result of multiple dictionary frames. We do not need to define everything at once. The dictionary frame below partially defines this domain's nouns and their relationships.

```
Dictionary: site vocabulary.
. A residence is a site.
. A residence has a street-address.
. A site has a set of energy-services.
. A site has an owner.
. A site has a location.
. A site has a current-account.
. An account has a customer.
. An energy-service has a meter.
. A customer has a set of accounts.
. A customer has a billing-address.
. A landlord is a customer.
. A tenant is a customer.
```

By applying a naming convention, some terms are automatically identified as nouns of a certain type when a hyphenated term ends with a noun that is already defined. For example, "new-billing-address" ends with "address" which is a part of the base vocabulary. The terms "old-residence" and "new-residence" become defined when "residence" is defined. This convention avoids tedious declarations such as
`"A turnover-date is a date."` and
`"A new-residence is a residence."`

Procedures—Roles, Actions, Procedures, and Actors

Actions may be invoked by plans or dialogs or other actions. In every case, the action will be performed by an actor assigned to a role. In some cases, the actor is component of the system and the actor's action statements are predefined in the system's *base vocabulary*.

Planning is great, but we still need doing.

If you diagram a plan tree with the end-goal at the far end, you get a kind of precedence chart. Traditional planners will quickly relate to that idea. You can also think of a plan tree as a style of state-transition-diagram. It shows how the states are connected by transitional actions.

But how does the plan get executed? We build the plan "from the top down," but we execute it "from the bottom up." The branching of the plan tree means that we have a notation that explicitly represents parallel processes. The rendezvous conditions are expressed in the task frames. Transition actions are blocked until all preconditions in a task frame are satisfied.

We need a notation for specifying those actions. The next section demonstrates how action frames are used to define procedures. Action frames utilize a notation that includes the usual procedural facilities for conditional execution and iteration. Action frames also allow the current role to delegate a specific step to a collaborating role.

Role frames define the actions performed by a role.

The task frames in the prior section identify roles which we called Billing and Mailer. The Billing and Mailer roles were given responsibility for certain *actions*. One of the actions assigned to Billing was to "final bill" an account. The role frame below indicates the procedure for that action. This example is limited to a single action. A more typical role frame will define a number of actions.

```
Role: Billing.

Action: Send final-bill for account at turnover-date
           to customer's new-billing-address.
. Apply account's deposit, if any, to account's balance.
. Invoice account for usage up to turnover-date.
. Send final-bill to the customer's new-billing-address.

To: Send final-bill to the customer's new-billing-address.
. Generate final-bill for customer's account.
.  Mailer:  Send final-bill for account to customer
              at new-billing-address.
```

The set of action statements that a role supports defines its public interface. The "To: " label above identifies a private action that is not part of the public interface. The public interface should be stable while the procedures and private internal actions may vary over time. In fact, implementations are not necessarily constrained by the instructions given in the role frame. Those instructions represent a reference implementation. The only requirement is that the actor respond to the public interface defined by its action statements.

Delegation

Roles can delegate work to other roles. In the role frame above, the Billing role delegates a task to the Mailer role. At run-time, a component called the Resource Manager assigns tasks to actors that know the role. When actors report for work, they contact the Resource Manager and specify the roles that they can perform.

Polymorphism

The dictionary notation supports subtypes. The action notation supports polymorphism. Here is an example. The procedure for reading a meter depends on the meter's technology. The role frame below defines several actions that differ only in the subtype named. The run-time will call the procedure with the most precise subtype.

```
Role: Metering

Action: Read energy-service's manual-meter giving meter-reading.
. ...
Action: Read energy-service's telephone-meter giving meter-reading.
. ...
Action: Read energy-service's mesh-net-meter giving meter-reading.
. ...
```

```
Action: Read energy-service's meter giving meter-reading.
. ...
```

Invoking a Goal—Dialogs and Protocols

The client of a business process may be a human being or another business process. In either case, the client invokes the execution of a plan via some kind of dialog. As a consequence, we need a notation that defines such dialogs. The connection between the dialog and the plan becomes explicit.

Dialogs—Responding to humans

The basic programming concept behind the dialog notation is stimulus-response. The basic syntax is similar to a stage play. When "he says this," then "she says that." In fact, the the dialog notation was inspired by the AIML chat-bot notation.

Here is an example of how the dialog notation works. Imagine that the utility business in the scenario given earlier enables the consumer or the call-rep as the proxy for the consumer to set things in motion via the following dialog. Assume that we have already authenticated and identified the customer so that we already know the current residence and current billing address.

```
Dialog: Customer is moving.

Context: Customer is moving.

U: * moving *.
U: * new address *.
S: We need to know your future address and when you are moving.
   Here is a form where you can fill in that information.
. Ask new-residence.
. Goal: Move customer billing from current residence to new-residence.
```

A dialog frame contains one or more *contexts* and each context contains one or more *vignettes*. Each vignette starts with a set of *patterns* representing alternative forms of the user's request. Each pattern is prefixed with "U:" to indicate that it is the user's part of the dialog. Each set of user patterns is followed by a single *system response*, which is prefixed with "S:" to indicate the system's part of the dialog.

The system is represented by a role called "Speaker" (the "speaker" for the system). The system response may be a mix of speech and action. The action statements are limited to the Speaker's public interface and the public interfaces of the roles in our current *ontology*.

In the dialog frame above, the "`Ask new-residence.`" instruction invokes an action statement that is one of the actions defined for the Speaker role. The Speaker will display a pane containing a fill-in-the-blanks form defined by a view frame called New-Residence. When the user releases (submits) the data, the new-residence data required by the next instruction will be on the dialog's blackboard.

The next instruction:
`Goal: Move customer from current residence to new-residence.`
indicates that a plan tree will be executed with the given goal. The customer must be identified before the dialog invokes this goal. If the customer entity is identified, the customer's attributes may be referenced. That allows "residence" to be referenced in the goal.

Protocols—Responding to robots

In an automated process, some clients will be robots. In this example, the client needs a cross-town delivery.

```
Dialog: Cross-Town Delivery.

Context: Request for proposal.

U: Please bid delivery of package from pick-up-location to drop-off-location
     between pick-up-time and drop-off-time.
S: (Need to compute a bid before we reply.)
. Scheduler: Compute most-reliable-promise to deliver package
             from pick-up-location to drop-off-location
             between pick-up-time and drop-off-time.
. If bidding:
  Our most reliable bid is pick-up between: [expected-pick-up-period]
    with drop-off between: [expected-drop-off-period].
. else:
  Sorry, we have no capacity available that meets your constraints.

U: Okay. Your bid is accepted.
S: Thank you. We are reserving that resource.

U: Sorry. We did not accept your bid.
S: Thank you. We are releasing that resource.
```

Sensors—Responding to transducers

One of the characteristics of event-driven user-interface coding is that it is very technology specific. One finds technology-specific events, such as "mouse-over," "key-down," "click," and "gaining-focus" driving the user interface response. As a result, we may need to write additional responses as new sensors are introduced.

We avoid this kind of technology-dependent code in the dialog frames. Our strategy is to introduce an actor (in the Scribe role) to translate sensor outputs into user speech acts. When the user's keyboard or mouse or joystick produces a signal, the Scribe interprets the gesture and the equivalent speech act is given to the dialog in place of the original signal.

In the example below, imagine that the user is operating a flight simulator and the sensor is a joystick. The Scribe will translate the stick angles into pitch-rate and roll-rate. If some other sensor such as a mouse, iPhone, or iPad is to be accommodated, the Scribe may be extended, but the dialog remains untouched.

```
Dialog: Space Flight Simulator.
Context: Pitch and roll rate.
U: Make the pitch and roll rates P and R degrees per second.
S: (Don't talk. Just do it.)
. Pitch-Thruster: Produce P degrees per second change in pitch.
. Roll-Thruster: Produce R degrees per second roll rate.
```

The dialog notation provides a level of abstraction that removes the coupling of the dialog with a specific interface technology. The resulting dialog may then be rendered in a web application, a web service, a VRML receptionist, a client-server framework, or any desired implementation of the Speaker and Scribe roles.

Showing Data, Asking for Data—Views and Forms

View frames provide fill-in-the-blank forms.

Dialogs are executed by an actor assigned to the Speaker role. The Speaker role contains a small set of action statements that are defined in the system's base vocabulary. If an instruction in a dialog frame does not specify a role, it is directed to the Speaker that is executing the dialog.

"Show view-frame." and "Ask view-frame." are two of the instructions that the Speaker understands. A view frame may be invoked from a dialog vignette

in read-only mode by the instruction "`Show view-frame.`"
and in data-entry mode by the instruction "`Ask view-frame.`"

In the dialog frame above, the system response includes the instruction "`Ask new-residence.`" The effect of that instruction is to display a view pane that provides a fill-in-the-blank form. When the user releases (submits) the data, the new-residence data required by some following instruction will be on the dialog's blackboard.

```
View: New-Residence.
! New Residence
  Please enter the street address for your new residence.

    Street: [street]
 Apartment: [apartment]
      City: [city]
     State: [state]
       Zip: [zip]
```

The view frame syntax is derived from wiki mark-up syntax. I chose wiki syntax over XML-based syntax because it is less intrusive, more concise, and produces an effect that is somewhat closer to what-you-see-is-what-you-get (WSIWYG). I will not attempt to describe the entire view frame mark-up.

The basic idea is that a view frame is used to view or enter data for an entity. The entity is named on the title line. The fields are assumed to be attributes of that entity. They do not need to be qualified with the entity name.

The data-type of each attribute is defined in a dictionary frame somewhere in the ontology. The rendering logic selects the appropriate widget based on the data-type. For example, in the view frame above, the `[state]` field references an attribute which is known to name a *category* defined by an enumeration containing the names of states. The field will be rendered using a drop-down list.

Because an attribute of an entity may also be an entity with its own attributes, one view frame can invoke another. The layout convention is to simply inset the second pane and provide a thin border. The left and top margins of the inset pane are aligned where the left and top margins of a simple text field would be.

The layout wizard follows simple and predictable rules to assure that a form is user friendly. For example, the layout wizard expects labels and fields to be paired as seen in the view frame above. When it sees that pattern, it will right-justify the labels and left-justify the fields. The width of each field is determined by the width required to enter 98% of the data without scrolling.

If we needed a layout that used less vertical space, we can put some fields on the same line, eliminate the title, and drop the white space. The result is shown below.

```
View: New-Residence.
Please enter the street address for your new residence.
Street: [street                                   ]  Apt: [apartment]
  City: [city                         ] State: [state        ] Zip: [zip  ]
```

The white space padding the field names above indicates the programmer's field-width preferences. If the programmer had not indicated a preferred field width, the layout wizard would have used a width determined by the data's statistical profile or the enumeration's maximum.

View Frame Summary

View frames are invoked by "Show" or "Ask" instructions in dialog frames. View frames may be nested when an attribute is also an entity. The view frame notation allows the programmer to specify the conceptual layout of the view while leaving details that can be easily automated to the layout wizard. At run-time, the dialog's blackboard persists the data entered. View frames are typically used to gather the data needed to fully specify a goal.

Relegate standard solutions to the run-time.

Business process automation involves a number of problem-space patterns that are highly predictable. The corresponding solution-space design patterns are also well known and equally predictable. We should not burden our programmers with work that is so obviously ready for automation. These tasks should be relegated to the run-time.

The following sections demonstrate how the Hum run-time obviates common programming tasks including:
Persistence—All business events are automatically recorded.
Messaging—Actors communicate transparently.
Job Accounting—Work assignments are work orders.
Resource Management—Business processes consume and create resources.
Trouble-Shooting—Handle business process exceptions automatically.
Dashboards—Monitor the business process.

Persist now, purge later.

In traditional frameworks, programmers spend too much time deciding what to persist, when to persist it, and how. This time can be put to better use. Programmers should be able to largely ignore the recording mechanisms while automating a business process. The solution is simple but radical.

Persist everything now. Decide what to purge later.

Change the problem from thinking about persistence to thinking about record retention rules. These tend to be relatively simple. For example:

- Asset records are retained for the life of the asset.
- Financial transactions are retained according to tax rules.
- Customer history is generally only relevant for a certain time.
- Certain employee events may be purged after some time.

Record retention rules belong to the enterprise world-base configuration. They don't belong to any particular application.

How might we completely automate persistence? The needed records are implicit in the business process description. In addition, there are design patterns (such as transaction logging) that must also be present to enable diagnosis, testing, and simulation. The following list outlines the tactics involved.

- When a post condition is asserted, it is an event. Record the event, including the state of the entities referenced by the nouns.
- When an actor delegates work to another actor, the caller may go to "sleep" until the delegated action completes. Persist the state of the caller during the sleep.
- When a blackboard is updated, record the update. Blackboards are typically updated by user interfaces.
- Record all updates to the world-state in a world-base. A world-base is a kind of temporal database that remembers the history of each entity until that history is purged.

- The world-base may be distributed and replicated. However, distribution and replication should be transparent to the business process. The persistence mechanism simply calls a World-Base actor with all world-base updates.
- Record all message traffic. This obviates local logs and enables tests of new processes with "live" data.
- A world-base belongs to a community. In a business context, that community is generally a business unit that can define its records retention rules subject to industry regulations. The record retention rules belong to the community not to a specific application.

Messaging is built in, not on.

The mechanics of sending a message to an actor should be a background process relegated to the run-time. The business programmer should not need to think about it. In fact, the programmer should not need to know if the actors involved are on the same host or even the same planet.

How might we automate messaging? The following list outlines the tactics involved in implementing the Messenger.

- Recall that actors are assigned at run-time. So message routing cannot be static; it must be dynamic.
- All messages between actors follow an asynchronous design pattern. There is no shared data.
- Every actor has a network address (URI). The Messenger (actor) knows if the communicating actors are co-located by comparing the host part of their network addresses.
- Every message may be recorded. This enables a replay if there is a serious system failure. It also allows us to determine how a modified business process will perform by running a simulation based on detailed historical data.
- The messages in-transit and the messages recorded must be encrypted for privacy. The only question is the level of encryption that should be applied to the message content. Policies for this may be declared at the entity level and the message level.

Every message is a job ticket.

In most of today's distributed business systems, resource accounting is an afterthought. We retrofit work-in-progress tallies and exception backlog histograms after we have written the "mainline" logic. Cost accounting resembles a kind of data mining performed after the fact. This is doing it the hard way and tends to produce untimely results of relatively low quality.

The required improvement is obvious when you consider that each message is, in effect, a kind of work order. Whenever a first actor delegates work to a second actor, the message is treated as a job ticket. When the second actor delegates work to a third, the run-time creates a subsidiary job ticket. In other words, all messages between actors are treated as job tickets. The job tickets are built into the messaging system so that the mechanics are transparent to the programmer.

As tasks are completed, the costs, quantified by the resources used, are "rolled up" and accumulated at various levels. This gives a detailed account of the resources used at every level. The top-most level maps to the goal at the root of the plan tree. Pricing the resources to monetize the metrics is done with a reference implementation of the Bookkeeper role. The reference implementation will generally be replaced by an organization's own.

Resource management is fundamental.

A system component, the Resource Manager, dispatches tasks to actors and accounts for their usage. The Resource Manager knows the actors' availability, so that the work-in-progress dashboard is a simple by-product. When other consumable resources and rentable resources are used, the Resource Manager is called to record the usage and update inventory records.

The programmer must assure that nouns representing resources have an appropriate resource class as a super-type. This informs the IDE so that it can remind the programmer if a resource is mentioned, but usage is not properly quantified. It is assumed that a rentable-resource is rented for the duration of an action. A consumable-resource must be directly quantified. The action notation provides a small set of action-statements that support the needed resource accounting. For example:
`Add measure of resource to container.`

Business Exceptions are routed to humans, mostly.

This section is about business process exceptions, not programming exceptions. It turns out that there are standard solution patterns for detecting and dealing with business process exceptions and the run-time can provide the needed exception handling. The programmer should rarely need to deal with this complication. The following subsections discuss typical causes and the needed responses.

In most cases, some human intervention will be required. The human "troubleshooter" may reassign people, warn the client, abort the plan. A human troubleshooter is a key component. We cannot automate everything.

If the action that detects the exception is assigned to a human, the trouble-shooting might also be assigned to the same human. The exception is still registered so that we know which actions are delayed and we learn about the process problems that may cause delays. It is worth noting that this idea implies that the human is in frequent or continuous contact with the Resource Manager.

The following subsections discuss the nature of different kinds of detectable business process exceptions. Four cases are discussed:
1. No actor available.
2. Actor disabled.
3. Running out of stock.
4. Actor blocked.

It turns out that every case involves situations that cannot or should not be handled automatically. They must be dispatched to a human troubleshooter. Finally we discuss how people are assigned to troubleshooter roles.

No actor available.

When an action becomes executable, the Coordinator asks the Resource Manager to dispatch the action to an actor that knows the required role. If no actor is currently available, the action is blocked. The duration of the blockage can be predicted to some level of reliability based on the cycle-time statistics for the tasks that currently occupy the available actors.

If no active actor knows the role, the action may be blocked indefinitely. The predicted duration of the blockage depends on the concept that some actor which knows the role may eventually come online. The calculation of the blockage duration requires that the actors provide a future availability schedule.

In both of these cases, the duration of the blockage may be predicted to some level of reliability. The predicted blockage duration may then be compared to a threshold value defining what might be acceptable. If the blockage is not acceptable, the human troubleshooter must be informed.

Actor disabled.

The actor that is executing an action may become disabled. Actors may be humans, robots, or business processes. A human may get sick or injured. A robot may be damaged or its safety mode may be triggered. The vehicle transporting the human or robot may break down. A business process may become disabled by financial conditions (nonpayment) or a change in policy.

Actions are considered "atomic" and generally should not be re-assigned automatically in these circumstances. As a consequence, these exceptions are always dispatched to the human troubleshooter to assure that appropriate judgement is exercised.

Running out of stock.

Some actions will consume material resources. While modern inventory systems can prevent most stock-out situations, they depend on historical demand patterns. If the current demand rate rises sufficiently above the historically determined confidence interval, a stock-out becomes likely or actual.

This is another situation that is sufficiently complex that we should dispatch it to a human troubleshooter. It may be feasible in some future world to dispatch this kind of exception to an artificial intelligence, but we do not yet live in that future world. The troubleshooter may have access to supplies and suppliers that were not given to the inventory system.

Actor blocked.

An actor might be blocked by some physical obstacle. For example, an actor may be locked out of a location. A change in access policy may prevent a needed communication.

Almost any step may be blocked. Sometimes the needed intervention is simple and sometimes not. The basic problem here is that we cannot predict all of the failure modes. Again, by default, we dispatch this exception to a human troubleshooter.

How are troubleshooters assigned?

Troubleshooters are assigned at run-time. A troubleshooter is an actor and has a network address (URI). Troubleshooters may be assigned exceptions according to the source of the exception in the sense that problems detected by a role may be dispatched to a troubleshooter associated with that role. Similarly, resource exceptions may be dispatched to a troubleshooter associated with a resource subtype.

Dashboards and Drill-Downs are built in.

The Resource Manager and the Bookkeeper operate in near-real-time. This provides the opportunity to display process statistics and business metrics on a dashboard that is also near-real-time. What might be displayed? The run-time operates in simulation mode or in real mode. The dashboard looks the same in each mode except that the simulation mode contains a progress bar.
Various widgets indicate:

- progress on the simulation's timeline (The progress bar is omitted in real mode.)
- simulation (or real) date and time (date-time widget)
- number of threads currently executing (active actors)
- frequency of events (Pareto graph)
- material resources consumed (Pareto graph)
- actor activity (Pareto graph)
- wait time by resource (Pareto graph)
- current exceptions (Pareto graph)

The Programmer's Assistant

I call the IDE the "Programmer's Assistant" or just "Assistant." You can think of the Assistant as a software actor that watches keystrokes and responds to what it sees in the resulting program statements. If you use a modern IDE such as Eclipse, Visual Studio, or Squeak, you expect the IDE to provide a smart editor with automatic formatting and color-coding. You also expect the IDE to provide diagrams that indicate the organization of your code modules. Those diagrams are derived from the code.

The Assistant is working in a context that I call an *ontology*. Basically, an ontology defines a kind of namespace. Each ontology is a collection of frames. The dictionary frames provide the kind of declarative information that we expect in an ontology notation such as OWL. However, in this framework, the ontology includes executable frames also. The executable frames provide the operational meaning and where-used context for the nouns.

While the code editor provides the usual visual cues linked to the code, the Assistant also provides diagrams (graphs) that indicate the organization of the entire ontology. The Assistant differs somewhat from other IDE's in that it is more focused on the total code base and less on the individual modules.

The Assistant's Assistants

All of the components in the Hum system are actors. Several actors provide the functions in the Programmer's Assistant.

Smart Editor
The smart editor signals the other actors when a statement is entered or modified.

Token Marker
The Token Marker marks each token in a frame. The type and attributes of the token determine how it is rendered by the Pretty Printer.

Pretty Printer
The Pretty Printer renders each frame with colors and icons that provide cues to the reader about the structure and status of the statements in the frame.

Statement Flagger
The Statement Flagger puts "flags" on each statement to indicate the status of the statement. A statement with no definition will have an "undefined" flag. If a local instruction is not recognized, it will be flagged.

Code Critic
The Code Critic provides suggestions that might improve the readability or structure of the frames in the system. If the Code Critic has something to say about a statement, the critic icon will appear next to the statement. If you want to know what the critic has to say, touch the icon and the critic's comment will

appear in a balloon. If the critic's audio voice is turned on, you will also hear the comment.

Librarian
The Librarian provides versioning services. One can "turn back the clock" and view the ontology as it existed at various times in the past. The librarian will also prevent accidental duplication or redundant definitions of a statement or dictionary entry.

Simulator
You may use the Simulator to test a program or to tune a business process. The Simulator will run the system with *sim-bots* (simulated actors) in place of real actors. If any undefined statements are encountered, the simulation will halt. One can "step over" undefined statements to continue the simulation. The simulator uses inputs that were recorded in previous simulated or real mode runs of the system. The simulator can also generate events and data.

Speaker
The Speaker provides the text-to-voice outputs from the system. The Programmer's Assistant provides different voices for each of the Assistant Assistants. You can turn off the Speaker's audio output when it is not useful or appropriate. When the audio is turned off, you can view the output as text instead.

Scribe
The Scribe provides the voice-to-text inputs to the system. The system expects verbal commands in the form "Scribe, do something." Each command can also be entered from the keyboard. Instead of verbalizing "Scribe, do something." you can enter "\do something." The Speaker provides the text-to-voice outputs from the system.

These diagrams are views, not models.

The Assistant produces diagrams that indicate the structure of the ontology. The diagrams are derived from the ontology. This is the opposite of some model-driven development environments. In the Assistant, the code precedes the diagram, the diagram does not precede the code.

The diagrams indicate the structure of the entire ontology. They provide visual cues when elements are missing or when the structure includes an illegal cycle (loop). The programmer may navigate to related frames by clicking diagram nodes and connections.

Plan Tree
The Plan Tree diagram resembles a precedence chart that reveals how preconditions link the task frames together. The programmer can examine the diagram to discover missing frames and illegal cycles.

Role Relations
The Role Relations diagram shows the calling relationships between roles. Since one role may delegate work to another, this diagram shows the degree of interdependence that may result. Cycles are allowed but flagged because they require careful attention.

Word Relations
The Word Relations diagram shows how nouns are related. Entities are connected to their attributes. Subtypes are connected to their super types. A visual cue identifies nouns that are predefined by the base ontology.

Dialog Structure

Dialogs contain contexts, contexts contain vignettes. A vignette may prioritize a different context. A vignette may invoke a view frame. The Dialog Structure graphs these relationships.

Simulations and Emergent Behavior

The Simulator supplies sim-bots in place of actual actors and runs on a scaled-time base where, for example, each millisecond simulates one minute of duration. That scale would enable a full year of 525,600 simulated minutes to run in less than 10 minutes. While using scaled time is not the most sophisticated approach, it allows one to use the same software in real mode and simulation mode with minimal adjustments and alterations. [precaution]

Each sim-bot selects simulated cycle times (action durations) from probability profiles provided by the Bookkeeper. The Bookkeeper derives the probability profiles from the data that it collects from job-tickets. When no history exists, the simulator may use cycle-time distributions provided in annotations that are appended to action statements or instructions.

A simulation may be driven by requests recorded in the message logs. When no history exists or we wish to create a new demand pattern, the simulation may be driven from the user interface with conventional load-testing tools.

Because the simulator is running the actual process and software rather than a model twice removed, we hope that it will produce relatively realistic behaviors in terms of resource consumption and resource-limited through-put. Of course, there are many assumptions involved in this hope. The key assumption is that cycle-times are not auto-correlated.

Merging Ontologies

You may encounter several hazards when you merge ontologies. The Programmer's Assistant includes a tool that will attempt to alert you to situations where the an ontology that you are attempting to merge will need some translation or "re-factoring" before you can safely merge it into your ontology. Specifically, the tools will warn you when the ontology you are importing contains:

- a matching entity, but with different attributes
- a matching entity, but with different super types
- roles that might be the same based on a community thesaurus
- roles that have the same name
- statements with similar noun patterns

When you merge statements from another ontology, the primary strategy is to replace nouns in the incoming ontology with equivalent, or nearly equivalent nouns from your ontology. The merge tool facilitates this replacement operation while watching for complications.

The Community Library

A programming language is a medium where authors may publish captured knowledge. Hum is open source and presents a number of opportunities where it may be enhanced or extended by open source and proprietary solutions.

Standard Components

Hum includes reference implementations of the Resource Manager, Bookkeeper, Messenger, Blackboards, World-Base, Speaker, and Scribe. These may be replaced by domain specific or proprietary implementations.

Plug-Ins and Adapters

Actors may be extended to facilitate interoperation with other technologies. For example:
1. The reference implementation of the Messenger implements only one communication stack. The community may extend it to provide adapters to other communication technologies.
2. The reference implementation of the World-Base is a "no SQL" implementation. Some community members may wish to extend it with a SQL interface or replace it with a relational equivalent.

Business Models

Hum contains a reference model of the order-to-delivery business process in the form of a skeleton implementation. The model facilitates industry-specific specialization.

Coordinated Ontologies

Industry-specific business models imply extensions to the base ontology.

Extensions for Virtual Worlds and Gaming

Business is a kind of multi-person game. Modern gaming software produces virtual world landscapes and avatars for virtual actors. While the dialog notation is designed to work with virtual world avatars, Hum contains no features that directly support virtual world landscapes. The concept of a landscape is not present in the language notations.

The Simulator is designed to run a "world" in scaled-time. However, the scale is adjustable so that the world may be running on a one-to-one scale. The Simulator may also be started and stopped. This enables some types of single-person games.

Consequences for Education and Training - What does the programmer need to know?

Today's business programmers, working in an environment such as the Java Enterprise Edition (JEE), need to know a great deal about relational database design, data manipulation language, object-relational mapping, at least one messaging protocol, the BPMN and BPEL workflow notations and engines, not to mention several alternative data structures for handling collections of objects, and at least one of several alternative frameworks for user interfaces. [JEE] Applying that knowledge characterizes a large part of the programmer's workload. The programmer is forced to work in the wrong problem space. They are working at the level of data processing problems, not business automation problems.

The framework described in this paper eliminates the data processing problem space by relegating it to the run-time. This moves the business programmer's focus and skill needs from data processes to business processes. There are consequences affecting how we train future business programmers.

Know less about data process, but more about business process.

Future business programmers will need to know more about the nature of large scale social systems. How is systems engineering applied to the enterprise? The skills needed for large scale automation have more to do with organizational structures and anthropology than data structures and algorithms.

**Know less about data representation,
but more about what the data represents.**

A careful examination of business reporting systems often reveals that the designers lacked a clear concept of how business processes are controlled and what business process metrics are appropriate. Future business programmers should, as a minimum, understand the fundamentals of accounting, finance, quality controls, and business statistics.

Working at enterprise scale typically involves facilitating the communications between several communities whose vocabularies are overlapping and inconsistent. The brute force approach is to demand or force-fit a standard vocabulary. "Let's have everyone learn to speak the same language." While there is a certain idealistic charm to that idea, it rarely works. The implication is that some care will be required to prevent mistakes and maintain an usable ontology. There is some risk that the word "spaghetti" which once characterized a certain style of programming may be applied to ontologies in the future. Of course, that is a word we wish to avoid.

**Know less about algorithms,
but more about optimization.**

Business management involves sub-optimizing a complex set of competing objective functions. The metrics may be slow-moving, auto-correlated, and noisy. Typically, the effect of an intervention may only be determined by a controlled experiment. Programmers will need to understand these concepts and the role that the business process metrics play in these contexts.

Competitive bidding involves a knowledge of bidding procedures, cost functions, risk assessment, and game theory. Dynamic scheduling involves another kind of bidding where the Resource Manager asks the actors to provide bids that identify costs and constraints. In both cases, we are dealing with a kind of constraint satisfaction problem under conditions that may contain uncertainty and noise.

**Know less about computers and networks,
but more about workers, robots, and organizations.**

Current procedures contain instructions to be executed or interpreted by a virtual machine, database engine, or user interface engine. Future procedures will be executed by robots (and avatars), human workers, and organizations. This implies that programmers will need to know how to author instructions to be interpreted by three different kinds of actors.

Summary

When we direct the evolution of our programming frameworks, each evolutionary step needs to:
1. Make the programming notations more expressive.
2. Make the run-time we are programming more powerful.
3. Make the tools smarter.
A breakthrough in productivity can result from the synergistic effect of addressing a number of aspects simultaneously.

To illustrate some of the next steps, this paper introduces a prototype business automation language consisting of five notations which address plans, procedures, dialogs, views, and ontology. Record-keeping tasks are relegated to the run-time so that the programming effort is focused on describing plans and procedures. The evolution of the IDE is focused more on revealing the structure of the entire code base and less on the structure of the individual modules.

When we move the focus from data processes to business processes there are consequences affecting the education and training of business programmers. Business programmers will need to know less about optimizing data processes and more about optimizing business processes. We can expect a greater separation between the knowledge and skills required for engineering software infrastructure versus the knowledge and skills required for optimizing a large scale business architecture.

Notes and Links

[conjecture]
My conjecture: The "capability" of a software architecture made up of some number of architectural layers is proportional to the product of the layers' capabilities. Each layer's capability is proportional to the logarithm of the number of fundamental statement types in the layer. For example, an instruction set architecture has a small number of instruction types. A procedural language has a slightly larger number of statement types. A library package has a somewhat larger number of method signature types. I use the term statement type and signature type because some subsets of the instructions, statements, and signatures are simply variations of the same concept.

[Martin01]
(reference the "Action Diagram" book)

[IDE]
My examples characterizing existing Java development environments reference Eclipse because I have more experience with Eclipse than I do with other Java tools. I believe that those Java tool characterizations apply to Sun's Net Beans, IntelliJ's IDEA, and Oracle's JDeveloper. I also believe that those characterizations also apply to the C# development environment in Microsoft Visual Studio.

[international]
I am not a linguist, but I have reason to believe that the notations can be translated to match the conventions of other written languages. The key concept is that statements contain readily identified nouns. The parsers amount to perhaps 200 lines of code, and are easily replaced.

[JEE]
I believe that it is equally difficult to develop enterprise-scale business automation ideas in Java, C#, and Python. The fundamental problem with each of those languages is that the algebra-based notation introduces an ergonomic hazard. In addition, the corresponding enterprise frameworks are more data centric than business centric. Hum is written in a dialect of Smalltalk. The Smalltalk notation is well-suited to expressing higher-level concepts in a readable form. However, the Smalltalk enterprise framework (in Visual Works) is also more data-centric than business-centric.

[disclaimer]
The code samples were written to demonstrate language features and bear no relationship to actual real-world code.

[underlines]
In the HTML version of this paper, hyperlinks in code samples link to this note. Normally, they would link to a data dictionary.

[precaution]
As a precaution, all messages generated in simulation mode are marked as simulated messages. The Messenger will not deliver them to any actor that is not a sim-bot.