"Skill-Driven" Software Design

By Avner Ben

Abstract: **"Skill-Driven" Design ("SDD") is a comprehensive computer-aided method of object-oriented software development, with these two distinct features: (1) It is a *language* that is *compiled* to visuals, rather than a visual editing system and (2) it is centered on *functionality* and functional decomposition, rather than the usual object model. SDD is meant to be an open-source project that will help professional OO software developers to construct software fast, allowing a greater portion of the time for design than has so far been common. This paper presents highlights of a useful case study, compiled with the pre-release support software.**

Software, Design, Object-Oriented, CASE, Open-Source

I. Introduction

"Skill-Driven" Design is a general-purpose-software development method developed and made open to the public by this author. A "Skill-Driven" design is made in the *Design* language, rather than in a binder of charts, as is now common. The (freely distributed) *Design* compiler reads the design and produces the charts, bound in a design presentation, plus code starters and exported databases, when these becomes necessary. "Skill-Driven" Design has been in the making for some 20 years, inspired by Structured Analysis and Design and facilitated by object-oriented programming. The format of the Design language is meta-object-oriented. The symbols used eventually generate code starters for such object-oriented programming entities as classes, methods and message passing occasions. The "Skill-Driven" Design paradigm is functional – it expresses mainly functionality and functional decomposition.

"Skill-Driven" design is an evolving project. It has been tested successfully in a number of commercial projects and is ready to be tested by the general software developing community. Specifying the entire *Design* language and its usage is beyond the present scope. This article describes the motivation for the method, regarding the present state of the art. It then demonstrates the usage of the method by highlights from a case study, annotated with brief definitions of the terms of the *Design* language.

Field code TLS.
Manuscript received on 20 November 2002.
Avner Ben has been programming for twenty years and has been teaching object oriented design and programming and designing object-oriented applications for twelve years. During the last decade, Mr. Ben has been working with the SELA group in Israel.
Avner Ben may be reached at avner@skilldesign.com.
The "Skill-Driven" Design site is at http://www.skilldesign.com

II. The Present State

"Programming" is a profession in the making. People have been working seriously with higher-level languages for less than 50 years (Fortran, 1957), which is nil compared with the experience, know-how and discipline that back such professions as civil engineering that has been with us for thousands of years or printing with movable type that has been with us for hundreds of years. Under these circumstances, anyone's guess on how a programming project should be conducted is equally good. In reality, most current programming efforts are conducted by trial and error. Whole books and articles are published by respectable professionals that show how code is written, fails, is improved, fails again, is improved again and so forth, until eventually it manages to do something and then, the motivation for the whole project is worked out from the solution.

Apparently, the quest of the emerging software industry for discipline has been bottom-up. Our success so far has been on the low level – the immediate implementation tools. In the beginning, programmers were occupied mostly with writing code – devising intellectually pleasing and machine-efficient "algorithms". However, in a very short time, "higher level" languages were offered to allow programmers to express problems and indulge less with the precise way the computer implements the solution. It is interesting to note that the "third generation" languages constructed in the 70's (e.g., C and Pascal, as well as improved versions of "second generation" languages as Fortran, COBOL and BASIC) still dominate the programming market, in one form or another.

So, we have become adept at *how* to do it. Still, we know little about *why* we do it in the first place. So far, all attempts to discipline the art of collecting requirements safely and ensuring their prompt execution into a comprehensive *method* have been a passing fashion. The 1980's saw the first wave of globally-acknowledged methods of structured software development – most notably "Structured Analysis and Design" ("SA/SD" - e.g., DeMarco[1]), SSADM, MERISE etc. These methods were acclaimed by the academy and embraced by the greater part of the industry for more than a decade and – puff! They are gone, almost without a trace.

What was the reason for the commercial failure of the SA/SD initiative? On the low-level, Structured Design was geared to generate strictly procedural code. On the high-level, the Structured Analysis language lacked the now-familiar object oriented constructs. So, with the revolution of object-oriented programming, the entire scheme became irrelevant. In

addition, the high-level, conceptual constructs of "dataflow" were downgraded by impatient implementation-minded individuals to *visual programming* and forced to imitate raw programming constructs. I believe this impatience to be the greatest obstacle in the way to making a respectable profession of out of programming. Apparently, the current wave of software design methods – centered around "UML", is heading in precisely the same collision course.

"Unified Modeling Language" ("UML") [2] originating with the Rational Company, probably in the context of its "Rational Rose" CASE tool and was later standardized by the Object Management Group ("OMG"). The UML concept is contradictory to everything that the preceding generation stood for. SA/SD was a comprehensive method. UML is a loose synthesis. Where SA/SD was based on a main notation ("Data Flow Diagram" – "DFD") with precise semantics meant to trigger fertile analysis, UML is a loose assortment of common notations imported from elsewhere, meant to invoke heuristics, if anything. Where SA/SD insisted upon a clear-cut design paradigm ("see the application from the viewpoint of the data items that flow through it") [1], UML is attempting to bridge among a number of software development disciplines developed elsewhere.

Finally, there is "Computer Aided Software Engineering" ("CASE") which originated in the 1980's to automate part of the SA/SD and – on the database development side - "Entity/Relationship" ("E/R") diagramming. The advent of UML has brought with it a revival of the CASE technology, this time, centered on the "Class Diagram" (A variant of E/R) and – in the embedded software industry - the hierarchical finite state machine. However, apart from the qualitative benefits brought by today's fast CPU's, color monitors and Graphic User Interfaces, the facilities are identical.

III. Problems with the present state

A. Problems with UML

UML has been a transitional-period measure and had better be left that way, until the correct alternative emerges. Its weaknesses as a lasting platform are too profound for time alone to mend: It is unreliable, economically inefficient, non-object-oriented and is improvised to begin with. Its wide acceptance by the industry at this very moment is poor excuse – we have seen such fashions come and disappear without trace – see the case of SA/SD.

1) Reliability

UML designs are heuristics that lack the integrity of the previous generation methods, which were built to reject error and draw attention to missing detail - "When a DFD is wrong – it is glaringly wrong!" [1] UML designs may or may not be wrong and may or may not contains all the information it takes – the only way to know for sure is to construct the code!

2) Support for the OO paradigm

Of the 10 odd notations making UML, only one ("collaboration diagram") is object-oriented (and is of limited use). The main notation used by most UML users (and used by CASE tools to generate code starters) is the class diagram – which is a just dialect of the Entity/Relationship diagram of old. In the second and final place comes the Use-Case diagram, which is the traditional flowchart with an added object dimension. Most CASE tools do not generate code starters from it. The embedded programming industry uses the hierarchical finite state machine model which owes nothing to UML and its association with objects is superficial. UML-based development often stresses Use-Case Driven Design[3] - a method of acquiring requirements *procedurally* (see for example its critique by B. Meyer[4]).

3) Methodical basis

At the bottom line, UML is an improvisation that lacks a sound theoretical basis. Backing the notations with Use-Case-Driven Design is a bottom-up approach that evades addressing the real problem domain.

B. Problems with CASE

A CASE tool typically consists of a database with a graphic front end and a report printing service. Although the technology has been with us for some 20 years, it is still premature. The weaknesses of current CASE technology are so profound as to suggest re-thinking the entire application. These tools are uneconomical to use and often degrade the notion of design into little more than visual programming.

1) Graphic user interface

Present CASE tools insists upon WYSIWYG data entry, but the graphic user interface supplied is not sophisticated enough to support the claim. Entering data is tedious, erratic, sometimes irreversible and often uneconomical (compared even with a general-purpose vector graphic editor such as Visio or SmartDraw). The graphics produced by most CASE tools are below presentation quality.

2) Code generation

The common economical justification for using CASE technology is the ability to generate code starters. Unfortunately, the symbolic language offered (UML) is not on a sufficient level of abstraction to support the claim. To generate code, the designer must trade *design* (in the original sense of the word) for "*visual programming*" – writing the code in graphic symbols. The real design – e.g., understanding the business requirements – must be done intuitively elsewhere. The final urge to downgrade design to visual programming is "full-cycle" engineering – use a CASE tool to generate the code, maintain the code and even reverse the design from the code after the latter has been modified offline. This low-level scheme has so far been successful in certain niche applications (e.g., hard-core real-time and GUI database front-ends) where real design is not an issue. Such applications are concerned with careful attention to detail, reliable execution and fast code generation. Here again, we find CASE technology playing a negative role. It makes whole installations spawn almost identical code over and again, instead of seriously attending to reusing their present code (and *designing* it for reuse). The latter is perhaps the only

economical justification for going *object-oriented*!

C. Problems with UML and CASE

The combination of a notation on an uncertain level of abstraction with an insufficient graphic user interface creates an inexplicable gap between the languages of design and implementation. While this gap was natural in the old procedural world, it contradicts the very object-oriented purpose. Consequently, we find programmers driven to either of two equally bad extremes: they either break the connection between design and implementation or give up design for good.

1) Programming by trial and error

Some programmers believe that designs are general guidelines by nature and do not have to map systematically to implementation in the first place. They draw charts for the record, use CASE tools as forms that must be filled, pretend to follow a project management standard as so much red tape. Eventually, then write – or generate - the code and proceed to maintain the code, leaving the design charts to rot on the shelf. While this approach may have its merits, e.g., for small scale "explorative programming" and pilot projects, it is unacceptable in real-life commercial projects because it makes compliance with the requirements impossible to trace.

2) Visual programming

Importing visual programming to the mainstream of software design is an attempt to eliminate the gap between design and implementation by getting rid of the first. The CASE tool allows the programmer to draw the program with graphic symbols and leave the real design to intuition, or to a preliminary stage, mistaken for "analysis". Again, this practice is disastrous in real-life general-purpose projects, because it makes compatibility with requirements impossible to trace.

On top of all that, current CASE technology will not even allow full-scale visual programming in the first place! UML-based visual programming fails to copy in full the multi-paradigm power and complete syntax of modern languages such as C++, e.g., its support for functional programming. A single line of C++ code that makes use of standard algorithms may require a full page of sequence diagram to express. The results – if attempted – will be less readable than the code! Too often, the programmer is asked to give up the better part of his/her language to be able to use the design tool!

3) The chart lay outing economical problem

"One picture is supposed to be worth a thousand words!" Indeed? One picture, carefully laid out by a professional graphic artist, may do. On the contrary, the average design document may contain dozens (if not hundreds) of charts – class diagrams, sequence diagrams, state/transition diagrams – all very much alike, some redundant and all in poor presentation quality. A dozen words may do a better job than a sequence diagram that contains two lifelines and a single message or a class diagram with classes spread all over the page and arcs crossing each other. Is *this* the job of our senior programmers – to be part time graphic artists? A senior programmer, during the phase of design, using a modern CASE tool (or even a vector graphic editor) may spend between quarter to half of his/her time doing nothing but lay outing charts. The effort is un-proportional compared with the architecture or engineering professions were charting is a long established activity. Considering the price of a senior programmer hour, there must be a convincing reason for spending so much money on a single activity - and the reason must be better than presentation quality or mere aesthetics.

4) The chart lay outing challenge

Either many programmers are wasting an enormous amount of good time and money on activities that are just nice to have, or there is a hidden science of software design chart aesthetics whose study may promote the profession. For example, we tend to group classes horizontally or vertically in the class diagram, to stress hierarchy (of inheritance or containment). We place symbols near or far from each other to denote some domain semantics. Whether the lay outing effort helps the picture replace a thousand words during design review is an open issue. To really earn its pay, the lay outing effort should have a solid software engineering outcome, such as exposing a neglected fact about the domain, or clustering software components into a software assembly.

I personally believe that software design chart lay outing has a sound case. Unlike the CASE vendors, the conclusion I draw is that the tedious – and inefficient - Human lay outing job must be replaced by algorithms that would do the job efficiently, and that the logic behind the aesthetics deserves an honest analysis effort. Allowing your senior programmers to spend so much expensive time on manual lay outing is about as economically wise as to allow them to write the resulting code in assembly language!

D. The "Skill-Driven" alternative

With "Skill-Driven" Design, I propose a fresh start, addressing the needs of the industry in a top-down way. "Skill-Driven" Design takes the discipline and analytic approach from SA/SD, the programming constructs from object-oriented programming and whatever positive may be collected from the elements that make UML. "Skill-Driven" Design takes from SA/SD the idea of deriving the object model from the functional model, the hierarchical decomposition, the timeless exposition of processes and the specification of transforms (but it does not take the DFD notation). From object-oriented programming, it takes the encapsulation of skills in an entity. The "Skill-Driven" design project develops along these guidelines:

- The chart lay outing challenge. **Problem:** We do not yet understand the logic of complex chart lay outing. Manual chart lay outing consumes a forbidding amount of expensive senior programmer time. With the poor quality graphics that result from present-day CASE technology, we are not sure whether this much time and money is going to a useful purpose. **Solution:** The chart lay outing application is yet to be analyzed and its exact requirements specified. In the

mean time, only simple charts whose logic is fully specified will be generated. Creative artwork, where strictly necessary, is to be added manually.

- The CASE user interface challenge. **Problem:** We are not yet capable of a useful graphic user interface for the purpose of CASE. **Solution:** In the mean time, the most economical option is to use a programming-like language. When the CASE/GUI application is properly analyzed and a working design is proposed, it will be implemented as a layer above the *Design* language (in conformance with the "document/view" architecture).

Here are the main assets of the "Skill-Driven" Design method":

1) The "Skill-Driven" paradigm

The purpose of a "Skill-Driven" design is *to ensure that program features answer to business functional requirements and nothing else and that changes of requirements do not destroy this traceability.* Any single fact that the designer adds must be justified in view of that! It must be possible to stop any programmer, at any stage whatsoever and make him/her respond immediately to the question: "Exactly what original business requirement is promoted by what you are doing right now?" "I am writing a sort algorithm" is not an acceptable answer!

The art of "Skill-driven" design strives to be conclusive by insisting on a single textual notation – the *Design* language – generating diverse graphic and textual views. Everything is measured in terms of required skills. A "Skill-Driven" design declares the following facts (Fig. 1): *The user requires skills of the software. The software offers facilities to the user. Software facilities are backed by skills of the software. Software assemblies are made of skills.* This is a functional paradigm, based upon functional decomposition. It is based upon the assumption that software is commissioned to serve a function and must not be allowed to degrade to serving itself!

2) Economy

The use of a design language allows a professional programmer to clarify complex logical problems to an arbitrary level of detail without being irreversibly stuck with code. By the time the *final* code starter is generated, only the details remain to be fit in. No matter how lengthy and responsible the filling in of coding details may be, decisions made at this stage seldom require reevaluating the design!

The use of a design language has these main benefits:

- **Agile high-level development.** Build small-scale systems in record time, while allowing for at least half of the overall development time to careful design.
- **Steep learning curve.** Allow a proficient programmer to speak in a natural language (compared with drawing graphics). Generate most of the graphics for review.

3) Compatibility

Object oriented programming: "Skill-Driven" Design is a native object-oriented method. The design term "Skill" maps N:1 to the programming term "method" and the design term "entity" maps 1:1 to the programming term "class". On the contrary, the *Design* language (purposely) lacks syntax for expressing a data-structure without functionality or a time-ordered process.

Present-day paradigms: As novel as the "Skill-Driven" paradigm may seem, it is really a careful synthesis of a number of familiar design paradigms (see section IV).

CASE technology: Since one of the outputs of a "Skill-Driven" design is code-starters, the results of an object-oriented design may be fed to a visual-programming system (e.g., CASE tool) via an XML database. On the contrary, producing a *Design* source from the information in an UML-based CASE tool would be a major reverse-engineering undertaking, since the *Design* source is on a higher level of abstraction. When the requirements of lay outing UML-type charts are properly analyzed and specified, the *Design* engine will be adjusted to produce these charts as well.

IV. The "Skill-Driven" Synthesis

Object-oriented design is about understanding – as opposed to solving – problems. The origin of object-oriented programming is in simulation software. The object oriented designer constructs a working model of the problem domain, believing that solutions follow naturally from careful study of the problem and need not be invented. In the first stage, object oriented *"analysis"* activities are there to collect all known and relevant facts that may contribute to the construction of a domain model. Then, (top level) *"design"* activities arrange these facts into a model that may be constructed using existing technology. Analysis serves the business. Design is an internal activity of the software development team, meant to ensure that the requirements (that were set forth during analysis) are indeed feasible. Since the cost of design changes during implementation is prohibitive, it is essential for the design team to make as much iteration as necessary until persuaded that the requirements are indeed realistic and that they know how to implement them. Of course, there always are marginal cases that may not be fully designed in advance, due to unfamiliar implementation technology or evolving market. However, experience has been showing that the part of the unknown, when well isolated, is much smaller than feared at the start.

"Skill-Driven" design addresses this challenge by combining a number of more or less familiar design and analysis disciplines in a process that gradually leads to code production and testing - see Fig. 2. The backbone of the process is *functional*: functional analysis (revealing business entities defined by *"strategic-level" skills* - see section III) leads to functional design (specifying all *"tactical-level" skills* and arranging them under software *facilities* - see section VI), which leads to functional implementation *(code starters)*. The analysis is supplemented by transform-driven analysis (revealing business *entities* and *"tactical-level" skills* - see section IV) and use-case driven analysis (determining *facilities*

- see section V). Project management combines with functional design to sequence the system facilities in an evolving product (see section VII). Data-driven design is used to determine the exact nature of associations suggested among entities and resolve entity instances. These improve the quality of the generated code starters.

Since the present scope would not suffice to describe any of these disciplines in detail or how they interact, this article will do with pieces of a case study, highlighting the unique contribution of each of these methods that make a design "skill-driven". The information system to be constructed is the "Personal Billing" system[5], used by the author to keep track of his teaching and software development jobs with the intent of billing his clients. This information system is a relatively small (less than 30 major entities, of which about 10 are directly related to the business model), lacks intimidating object-oriented features (such as polymorphism) and is easy to explain. I wrote my first Personal Billing system some 15 years ago with a mainframe hierarchical database, then redesigned and implemented it with a PC application generator, then redesigned and implemented it in Microsoft Access. The present redesign effort is meant at a fresh implementation in C# with XML, which (at last!) enables a full-fledged object-oriented design. Since the business domain is well known and no implementation risk is expected, the design may take the lion's share of this one-man project, which favors "Skill-Driven" design. Since the domain is obvious, and since "Skill-Driven" design graphics are meant to be intuitive, then the following account of an information system *should be* self explanatory!

V. "Functional analysis" support

A. Terms

The *"Assembly"* is the basic project management unit of "Skill-Driven" design. It comprises the least amount of work that may be assigned to a single developer to design, implement and test in full. In addition, assemblies also make semantic and reuse units. An *"entity"* is anything on which information is managed and to which responsibility may be delegated. The conceptual term "entity" maps 1:1 to the object-oriented implementation term *"class"*. *A "skill" is an ability that is manifest in software and which may be traced to a functional requirement from the software.* The common case of "tactical-level" skills will be presented later. A *"strategic-level"* skill, like those in Fig. 3, does not *necessarily* suggest a discrete procedure. It may consist of many activities that are executed on different occasions, still grouped under one functional unit. *When put together in hierarchy, strategic skills reconstruct the original requirements of the system.* Each strategic skill gives the reason for the existence of exactly one entity (or assembly), thus giving a functional definition for what would otherwise be mistaken for a mere data record or routine library.

B. Purpose

The "strategic skill" tree (see Fig. 3) displays the division of the Personal Billing system to assemblies and then to entities. Each assembly and each entity are defined by a single "strategic-level" skill. Some entities serve as "façade" for their assembly - a single entry point for all other objects which are hidden inside. Entities (and sometimes assemblies) are arranged hierarchically for the sake of readability[1].

The purpose of this chart is to redefine the top-level functional requirements of the system in practical terms. (All top-level requirements are constrained 1:1 to software entities and these are constrained N:1 to assemblies).

VI. "Transform-Driven Analysis" Support

A. Terms

"Transform" is a Human language sentence where the subject is a message-sending object, the verb suggests the operation required and the object is the receiver of the message[2]. A transform may be followed by a comment detailing the reason for the message and whoever else is expected to be involved in method of its execution. *"Transform sequence stage"* is a sequence of transforms that either describe a real use case or summarize the necessities of all use cases of a kind. A transform sequence stage may be headed by an informal account of the sequence. *"Transform sequence"* is a sequence of transform-stages, where the division to stages suggests significant gaps, e.g., the lapse between the storage of data and its possible use by another process. See Appendix 1 for the Personal Billing example.

B. Purpose

"Transform-driven" design is an informal analysis method, inspired by Structured Analysis and Design, [1] that can teach a lot about the business domain in a short time and will proceed to inspire a more precise use-case-driven and functional analysis and design efforts.[3]

VII. "Use-Case-Driven Analysis" Support

A. Terms

"Functional use-case" is a grouping of automated facilities that are likely to be applied, in whole or part and in any order, by the user of the system, to achieve a single major business objective during a single session - see section VI. The facilities

[1] The strategic-skill hierarchy does not necessarily copy a programmatic inheritance or containment hierarchy. Its sole purpose is to reconstruct the functional requirements using programmatic entities. It is yet to be studied whether this for-now intuitive ordering does suggest a logic that merits formalism.

[2] Oddly enough, "transform-driven" design appears to support "use-case-driven" design. Each transform sequence may be freely adapted to a *sequence diagram* and possibly inspire - or validate - a use case. Still, the translation is by no means 1:1. A transform sequence is a use-case scenario – it details a sequential flow of events that took place – however, its lines refer to entities, rather than instances!

[3] The "transform sequence" is a far relative of the familiar Data Flow Diagram ("DFD"), but with an object-oriented stress.

may be ordered hierarchically for the sake of readability. See Fig. 4 for the Personal Billing example.

B. Purpose

"Functional" use-case driven analysis is a restricted form of "use-case driven design"[3]. A use-case specification becomes functional when it has time ordering successfully removed. Time ordering is well known to be detrimental to correct software design. The exact reason is beyond the present scope - see for example B. Meyer's classical account[4].

The purpose of the "functional" use-case is to offer automated facilities to the user in an attractive way. This procedure guarantees that all the system services are indeed useful and that operations that must be manually sequenced are considered in advance. An obvious use of "functional" use-case analysis is to inspire the system's graphic (or other) user interface - e.g., a menu system.

VIII. "Functional Design" and Implementation Support

A. Terms

A *skill* is considered *"tactical"* when it may be implemented in a procedural language by a single procedure, block or expression. A *"Facility"* is a tree (or forest) of tactical skills and thus is effectively "object-oriented Pseudocode". *"Reliance"* is a binary directed association between two skills, where one skill requires another. (E.g., in order “to format Billing Report”, one must also be able “to format Work” many times.) Whether the same person does both jobs, or whether the person responsible for the first job delegates responsibility to another person does not change the very fact of *reliance*. Most reliances are implemented simply, in a procedural language, as function call, block nesting or expression nesting. (For example, compare the exploded "facility" of Fig. 5 with the generated code starter of Appendix 3.) The *"functional closure"* of a skill is all its reliances. I.e., to satisfy a skill, all skill in its closure must be satisfied, subject to reliance conditions. A *"facility"* is exploded for presentation by expanding the functional closures of its roots.[4] Software facilities often feature single root and explode to a tree. Forest facilities describe an asynchronous – or “event-driven” - design where a number of skills are required to realize a skill, but they have to be coordinated from outside. (E.G., the “Report Production” facility relies both “to initialize Billing Report” and “to print Billing report”. The first skill does not chain the latter and the latter skill does not wake the first. It is expected of whoever presses the “Print” button to make sure that the report is indeed initialized. Otherwise, the results are undefined.) A detailed account of a non-trivial event-driven design may be found in my “Skill-Driven” account of the Model/View/Controller architecture. [6] A facility root may itself be a branch in the explosion of another facility. Cutting to size is either the result of use-case analysis (see section V), but just a matter of convenience.

"Coupling" is an asynchronous binary directed association between two skills where the first skill requires a resource that only the second skill can produce, but it cannot control it. E.g., reading data that has (hopefully) been written on time or waiting for an event to fire. Couplings are synchronization information that suggests the time ordering of reliances in the same closure (which are, by default, arbitrarily ordered).

Another means for understanding the implicit order between skills is the analysis of their contract. Each skill is defined by *pre-conditions*, *post-conditions*, *axioms* and *failure conditions*. Matching preconditions to post-conditions may help to analyze the implicit order. The exact syntax of the *"TLDBC" (Top-Level "Design by contract")* is inspired by B. Meyer's work on the Eiffel language [4]. The syntax of the TLDBC language is beyond the present scope - see appendix 3 for an example.

"*Visibility*" between entities is implied when an entity relies upon another entity. "Visibility" between assemblies is implied by visibility between entities of different assemblies. A major objective of the "Skill-Driven" designer is to minimize visibilities among entities and in particular, among assemblies. Visibilities among assemblies limit the modularity of the design and complicate project management[5].

B. Purpose

The main activities in the "Skill-Driven" Design process is ordering tactical skills in facility explosions and watching the result from various angles. Normally, the process is "top-down", starting from facility roots and proceeding down to detail when its time is ripe. When ready for "low level" - or "detailed" - design, the designer enters an iterative process of adding detail such as function names and argument types, generating code starters and occasionally stopping to correct the design itself.

C. Visual language

Since functional design is the backbone of "Skill-Driven" Design (as well as its birthplace), the latter proudly offers an abundance of graphic representations and levels of summary for depicting it. Fig. 5 features a facility explosion in an hierarchical view - the form that appeals most to the procedural programmer. This simple form resembles a procedure call tree, however, as the *"code starter"* of Appendix 3 shows, the mapping is definitely not 1:1. This code was generated from the information in the *Design* program database, reflecting mostly the facility of Fig. 5. The code is a "starter" only, waiting for a proficient programmer to expand it to a working form. The objective of code starter generation is to allow the programming-minded designer to understand the implications of the design in code as many times as needed during the design process, before committing

[4]For the sake of information hiding, facility explosion is cut at assembly boundary.

The analysis of visibility is also where the functional and data-driven paradigms meet. "Navigability" over an association between two entities is a special case of visibility. In "Skill-Driven" Design – as in SA/SD before it - the object model is expected to result from careful analysis of the skill/reliance model.

to the final code.

The *"wire chart"* in Fig. 6 is a graphic representation of the same facility that is exploded in Fig. 5, but with a profound object-oriented stress. The wire chart is sorted by entity, so that time ordering is made (intentionally) impossible. The call-tree ordering is implicit. For those who must chart the control flow within the reliance lattice, the applet that actually displays wire charts in the *Design* output (which is on an Internet page) kindly supplies a chart animation service. The purpose of the wire chart is to show the collaboration among the participants in detail. Although the wire chart contains the same amount of detail as the respective facility explosion, its timeless format stresses the density of the reliance lattice and makes recurring patterns of control the more apparent and shows immediately how elegant - or how clumsy - the design is. The wire chart format especially excels in exposing couplings (charted on the *right* margin). On the contrary, it is practically impossible to show coupling information effectively in a time-driven format such as call tree, flowchart or sequence diagram.

Since all functional decomposition details are in an automated system, the next step is to provide the designer with higher-level summaries. Fig. 7 features an entity-level wire chart of the facility explosion in Figs 5-6.[6] This view level is beginning to be of interest to the real design-minded designer. Finally, the project-management-minded designer is offered a system-wide visibility summary on assembly level – see Fig. 8. It is well known that the schedule of a software project is determined almost entirely by the dependency (i.e., visibility) among the software assemblies. Therefore, the ability to assess the assembly-level visibility lattice at any time is guaranteed to drive the designer to limit the dependency among assemblies to only where strictly necessary!

IX. Data-Driven Design Support

A. Terms

"Class" is the programmatic representation of "Skill-Driven" Design's "entity". *"Association"* represents permanent visibility between two entities. *"Navigability"* is the fact of the permanent visibility - a path is made possible between any instances of two classes, allowing whoever got to that point (e.g., a method executing over an object) to navigate on. In "Skill-Driven" design, navigability is a restricted case of visibility and visibility is just the fact of reliance of one entity on another (to promote the provision of some facility). The decision that a visibility is permanent - and thus merits an association in the class diagram - is largely manual. The default association is *"aggregation"*. It simply states that an object of a type is aware of the existence of an object of another - or the same - type and no more. *"Composition"* is a special case where the object, in addition to being aware of the other object's existence, also controls its life span, and thus effectively "contains" - or "composes" - it. While it is not essential for the composing object to create the composed object, it is essential that it will clean it up in time and will not allow another object to endanger its existence. Objects of each type in the association have a "role". The *"quantity"* of a role is the number of objects of a type that are allowed for it, given one object, playing the other role. Role *"cardinality"* is an array of quantities, where each consecutive pair suggests either a range or an alternatives. A quantity of zero makes the role "optional".

B. Status

Fig. 9 features the *"class diagram"* of the "business" assembly (i.e., the "document") of the Personal Billing system.[7]

The graphics at Fig. 9 were rendered using SmartDraw 6. The "Skill-Driven" Design support for *data-driven* design is currently under definition. This important design discipline has been deliberately left to late at the Skill-Driven Design project. The challenge of combining functional design with data-driven design (without degrading to visual programming) is not a small one and requires much attention to detail.

X. Project Management Support

A. terms

Assembly *"evolution"* is a linear plan for implementing a assembly. Assembly *"generation"* is a stage in a assembly evolution, scheduling a number of skills (of this assembly) for implementation. The entire closure of these skills will be implemented in this stage, except for skills that have been explicitly scheduled for a later generation. The assembly evolution defines a product that is growing in functionality with each successive generation, until reaching maturity in the last generation, which installs the entire functional closure of the assembly. Like a assembly, the very system evolves too, but the system has no skills of itself. The *"milestones"* - generations in the system evolution – assemble a number of assembly generations and thus schedule a system-wide product. Like a assembly evolution, the system evolution describes the gradual construction of the system product from limited to partial to full-fledged functionality. The final milestone exhausts all generations of all assemblies in the system. See Fig. 10 for a Personal Billing system example.

The system Gantt (see Fig. 11) charts the successions of assembly generations, ordered horizontally according to their inherent visibility constraints (analyzed from skill/reliance). The duration of each assembly generation is computed from the number of skills it involves. The unit used to measure duration in a "Skill-Driven" project is the *"EM"* - *"Entity Measure"* - the average number of useful skills per a useful

[6] This format is a cross between functional design and the object model. It shows net *visibilities* - specifying collaboration - among objects and skips the procedural details. Thus, it establishes the background for constructing "class diagrams".

[7] The Class diagram, which is part of the UML, is a derivative of the long established "entity/relationship" model.

entity in the system. Skills and entities are *"useful*" if they are relied upon by someone. When computing the time left for working over a assembly generation, it is assumed that the implementation will take an equal time to that invested in pure analysis and design (unless another ratio is specified), the percentage of design completeness and evaluation factor of the assembly and developer, if specified. To convert the duration from EM to days, an EM is considered to represent 3 days (unless another ratio is specified).

B. Purpose

In "Skill-Driven" design, from the moment a significant number of skills and reliances is accumulated, the project Gantt materializes and proceeds to accompany the designer at any step, ensuring that a realistic evaluation of resources and development time is always at hand, based upon precise data. Although the assertion of "three days per class" (or "100 classes per man-year") may seem arbitrary at first sight, it has been well proven in hard-core object oriented projects, involving above 30 major business entities. In projects that give reason to believe that the pace should be faster or slower - adjust the EM accordingly. To be on the safe side, state the percentage of completeness for each assembly generation, to account for additional design work that may lie ahead.

XI. The status of the "Skill-Driven" project

The *Design* language was specified and implemented in late 2000. It was used in a small (5000 useful C++ code lines) commercial project in 2001. The language was completely restructured by end 2001. During 2002, the *Design* compiler was used in a commercial project to generate C++ code starters for code that eventually weighed some 30,000 useful C++ lines.

The functional support is the pillar of the entire system and is well defined and functioning, with the exception of "virtual" skills" that are under construction. The project-management support was added during 2002 and used successfully to schedule a commerical project. The use-case-driven and transform-driven support layers are conceptually important but structurally superficial and are currently under construction. Two major tasks are still ahead: The data-driven support will boost the usefulness of the generated code starters. However, I had rather plan this layer with much caution, to prevent the peril of degrading to visual programming. Finally, a teamwork support will be added to allow a team of developers to use the same design repository concurrently in an intelligent way. This final task has more to it than just client/server architecture of the *Design* compiler. It involves an analysis of the teamwork design application and the addition of interaction among *systems* to the *Design* language. Until then, the *Design* compiler will safely remain a single user tool that accommodates a single system.

The *Design* compiler currently weighs some 15000 useful Python lines plus some lines of Java and Javascript. It may be downloaded freely from the "Skill-Driven" design site[7] in either raw Python or in Microsoft Windows executable form, together with an HTML presentation-support library. Although the Design code has - itself - been designed in the *Design* language, it is currently under construction and is likely to be adjusted to meet users' demands. When release 1.0 of the *Design* language and compiler will is ready, I will place both final code and its "Skill-Driven" design at the "Skill-Driven" Design site for the public to use under an open-license. I expect that to happen by end 2003.

Appendix

This section features three textual samples from the Personal Billing case study whose careful reading will enhance the pictorial samples brought in the figures.

Appendix 1. A comprehensive "transform sequence" traversing the Personal Billing system. This is the story of a course, from Service to Contract to Account to delivery of Events to billing.

1. *We add a "Design" course to our services.*
    a. The Accountant *inserts* a **Service**, (Service type "Course", default Account type "Class").
2. *A client takes interest in the new course. We make a contract for 25 Gold an hour for teaching and 10 Gold for checking exams. By default, classes begin at 09:00 and last 8 hours.*
    a. A Client *opens* **Contract** (for Service).
    b. Contract *opens* **Tariff** (for Skill. Sequence repeated twice).
    c. Service *retrieves* **Skill** (For validation).
    d. Tariff *opens* **Tariff Version** (the default).
    e. Tariff Version *opens* **Hour Rate** (starting at current date).
3. 3. *We are commissioned a design class by the client, Client code 125, allowed 1-hour commuting time. A Class is scheduled every Thursday, from October 8, 5 times in a row.*
    a. A Contract_*retrieves* **Tariff** (for Skill).
    b. Contract *opens* **Account** (external id set to 125, default commuting time set to 1).
    c. Account opens Work (for Tariff Version).
    d. Work retrieves Contract (through Tariff, through Tariff Version, to validate against Contract through Account).
    e. Calendar *opens* **Event** (for Work. Sequence repeated 5 times, dates computed.)
4. *October 15 is moved to previous Wednesday.*
    a. A Schedule Row View *modifies* **Work Schedule Cell View**. (Update done by dragging cell in the schedule editor.)
    b. Schedule Cell View *modifies* **Event** (date set to 14).
5. *The teaching event of October 14 indeed takes place. It is checked, less half hour, due to our delay. In addition, two exams are checked, lasting an hour.*
    a. A Schedule Cell View *modifies* **Event** (changing duration to 7.5, status to True).
    b. Schedule *opens* **Work Schedule Row** (for same Account and Skill "Exam").
    c. Account *retrieves* **Work** (for Skill "Exam" through Tariff through Tariff Version. Assume not found - proceeding to open).
    d. Contract *retrieves* **Tariff** (for Skill "Exam", to validate).
    e. Account *opens* **Work** (for same Account and Tariff).
    f. Schedule View *opens* **Schedule Row View** (for the new Schedule Row).

g. Schedule Row View *activates* **Schedule Cell View** (duration 1 hour).
h. Calendar *opens* **Event** (for Work, duration 1).

6. *Early November, an attendance report and a billing report are produced for October. October 15 gives two lines in the attendance report: The teaching event on October 15 adds 25 * (7.5 + 1) = 212.5 to client/class/teaching and to total for pay and is closed for modification. The checking event on October 15 adds 10 * 1 = 10 to client/class/checking and to total for pay and is closed for modification.*
    a. (Deliberately omitted.)

Appendix 2. Part of the Personal Billing *Design* code, containing the beginning of the Billing Report assembly, an entity, some skills and the explosion of the facility painted in Figs. 5-7 and coded in appendix 3. Each line is a separate command, starting with a keyword. Keywords have been capitalized automatically by the *Design* program. The *minus* symbol denotes a continuation line. The *item* symbol is for "user-defined" items. Such items are looked for by certain printouts (e.g., the *namespace* attribute, where available, is used by the code generator - otherwise it is generated). Other unknown attributes (e.g., "*invariant*" in the entity definition) are simply appended to the descriptor, when printed). Reliances are denoted by the keyword “RELIANCE” or by the paragraph symbol. The owner of the reliance is determined by the degree of indent. The *level number*s are generated for readability and may be removed. The grouping of *skill TLDBC* definitions under *entity* is generated automatically. When planning, the designer normally dictates skill definitions on the fly, while relying upon them for the first time (or accumulating definition parts later on). The clause “Façade” strategic skill defines the entity as the façade of the assembly. The “provided” clause specifies coupling: “to print Billing Report” relies upon data from “to initialize Billing Report”, but cannot oversee its production.

```
ASSEMBLY P-Bill Report Assembly ("pbl_report")
§ NAMESPACE rpt

  MAJOR-STRATEGIC-SKILL to bill Clients and report to them
  § IS to summarize Payable Events per time period and
  - Client and compute amounts payable

ENTITY Billing Report ("BillRpt")
§ IS the report of all Works within a period, with hour totals
- and amount payable
§ INVARIANT report ready for printout
  FACADE-STRATEGIC-SKILL
  MINOR-STRATEGIC-SKILL to export billing report
  SKILL to initialize Billing Report
  § IS the Billing Report constructor
  TRANSFORM OF report period
  TRANSFORM OF AND client list
  TRANSFORM TO raw report line sequence
  SKILL to format Billing Report
  TRANSFORM OF Schedule
  TRANSFORM OF AND client list
  TRANSFORM TO raw report line sequence
  METHOD format()

  - # (Remaining skills and entities deliberately omitted)

FACILITY Billing report production
  RELIANCE 1 to initialize Billing Report
    RELIANCE 2 to initialize the Schedule
    RELIANCE 2 to format Billing Report
      RELIANCE 3 MANY Work,OPTION Client in list:
      - to format Work for Billing Report
        RELIANCE 4 MANY: to summ Payable Event
          RELIANCE 5 to get hours payable
        RELIANCE 3 to summ Work amount due
          RELIANCE 4 to quote Hour Rate per Work
          RELIANCE 4 to compute Work amount due
          RELIANCE 4 to summ grand amount due
          RELIANCE 4 OPTION: to open Skill summary line
            RELIANCE 4 to sum Skill amount payable
  RELIANCE 1 to print Billing Report
  PROVIDED DATA FROM to format Billing Report
   - CONTEXT Billing report production
     RELIANCE 2 MANY: to print Report Line

   - # (Remaining facilities deliberately omitted)
```

Appendix 3. This C++ class code was generated from the facility explosion in Fig. 5. Net skills are preceded by "todo:". Reliance on skills that are implemented by a discrete method generates function call. Skill contract generates comment.

```
01: // Billing Report
02: // the report of all Works within a period, with
hour totals and amount payable
03: // INVARIANT: report ready for printout
04: class BillRpt
05: {
06:   //   visibilities   (suggested   member   data)
07: private:
08:   schd::Schedule aSchedule;
09:   std::vector<RptView> RptViews; // MANY
10: public:
11:   // "to initialize Billing Report"
12:   // Description: The Billing Report constructor
13:   // Transform of (a) report period and (b)
         client list into raw report line sequence.
14:   BillRpt() {
15:      aSchedule = new Schedule;
16:      format();
17:     // (implementing "to format Billing Report")
18:   }
19:   // "to format Billing Report"
20:   // Transform of (a) Schedule and (b) client
         list into raw report line sequence.
21:   void format() {
22:         // MANY Work,OPTION Client in list: "to
23:format Work for Billing Report"
24:      while (/* Work */) {
25:         if (/* Client in list */) {
26:            formatWork();
27:         }
28:      }
29:      // "to summ Work amount due"
30:      // Transform of (a) Work and (b) Work total
         hours.
31:      /*  unconditional block  */  {
32:         aWork.QuoteHourRatePerWork();
33:         // todo: "to compute Work amount due"
34:         // Transform of (a) total hours and (b)
            hour rate (amt) into Work amount due.
35:         // todo: "to summ grand amount due"
36:         // Transform of (a) Work amount due and
            // (b) accumulated total amount due into
               accumulated total amount due.
37:         // OPTION first Work of a Skill met: "to
               open Skill summary line"
38:         // Transition when first Work of a Skill
               met, leading to total Skill amount
               due zero.
39:         if (/* first Work of a Skill met */) {
40:            // todo: "to open Skill summary line"
41:          // Transition when first Work of a
                Skill met, leading to total Skill
                amount due zero.
42:         }
43:         // todo: "to summ Skill amount payable"
44:         // Transform of (a) Work amount payable,
               (b) accumulated total Skill amount
               due into accumulated total Skill
               amount due.
45:      }
46:   }
47:   // "to print Billing Report"
48:   // Transition when printing requested, leading
         to report visible.
49:   // - Transform of Report View.
50:   void print() {
51:      // MANY: "to print Report Line"
```

```
52:      while (/* unknown condition */) {
53:         aRptView.PrintReportLine();
54:      }
55:   }
56:   // "to format Work for Billing Report"
57:   // Transform of Schedule Row into raw report
         line.
58:   void formatWork() {
59:      // MANY: "to summ Payable Event"
60:      // ransform of (a) Payable Event, (b)
            accumulated Work amount due into
            accumulated Work amount due.
61:      while (/* unknown condition */) {
62:         // todo: "to summ Payable Event"
63:         // Transform of (a) Payable Event, (b)
               accumulated Work amount due into
               accumulated Work amount due.
64:      }
65:   }
66: }; // end class BillRpt
```

REFERENCES

[1] DeMarco, T. Structured Analysis and System Specification, Prentice Hall, Englewood Cliffs, NJ, 1978.

[2] The latest UML standard may be found at the official UML site at http://www.uml.org/.

[3] Jacobson I., Object-oriented design, a use-case driven approach, Addison Wesley 1992.

[4] "Object-Oriented Software Construction Second Edition," Bertrand Meyer, Prentice Hall PRT, Upper Saddle River, New Jersey, 1997.

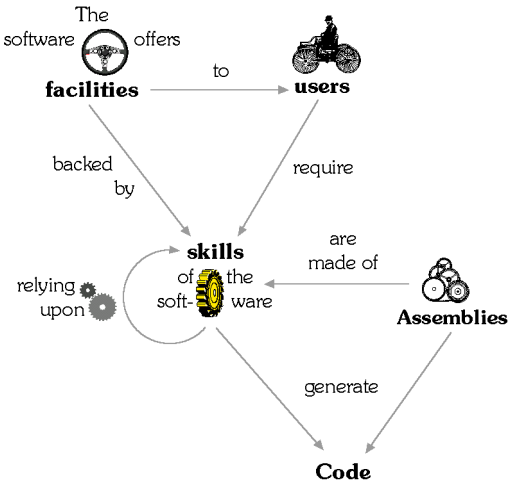
[5] The complete case study is listed at the "Skill-Driven" Design internet site at http://www.skilldesign.com/ Design/1/2/0.html

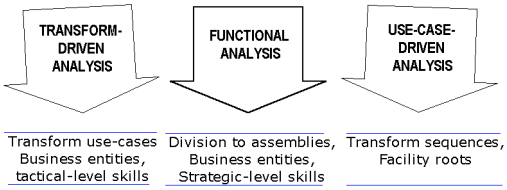
[6] Ben A., “The MVC Architecture – A Skill-Driven Account”, JOOP December 2000. See also http://www.skilldesign.com \articles\mvcintro\article.html

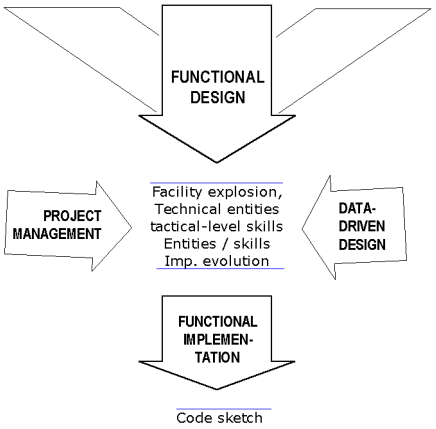
[7] The *Design* compiler may be downloaded from the "Skill-Driven" Design internet site at http://www.skilldesign.com/ Design/1/1/4.html

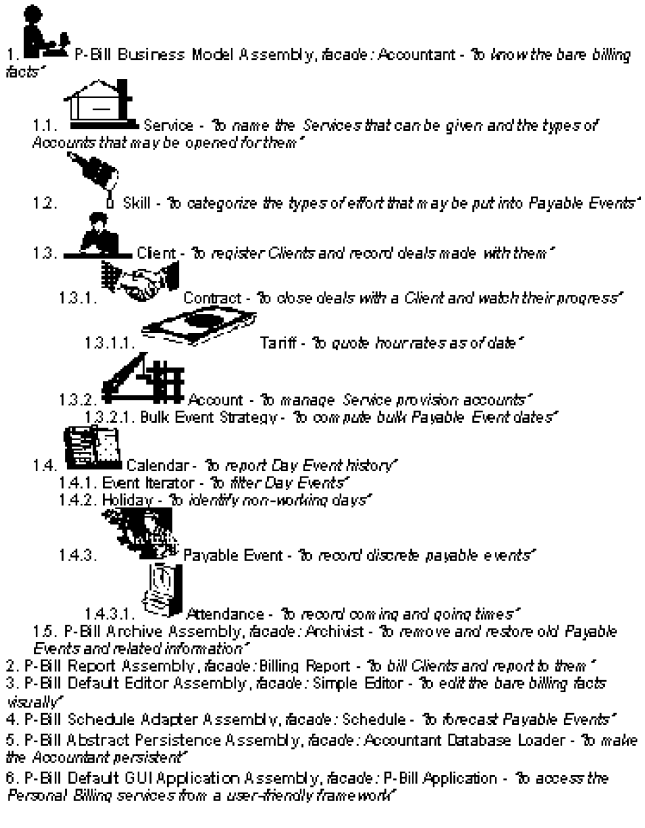
[8] List of figures

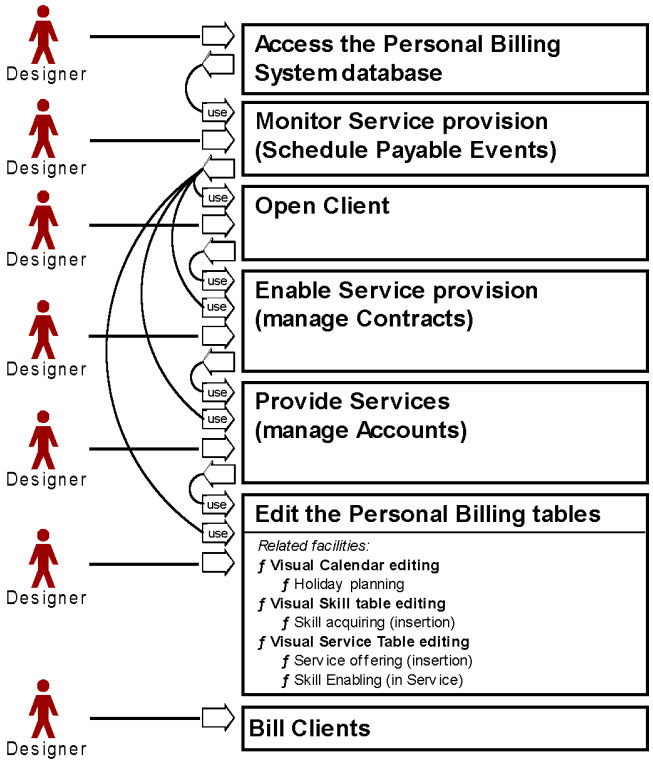
The "Skill-Driven" paradigm











to initialize Billing Report
(by Billing Report)
to initialize the Schedule
(by Schedule of schd)
to format Billing Report
(format)

Skill usage contract:

- *Transform of (a) Schedule and (b) client list into raw report line sequence.*

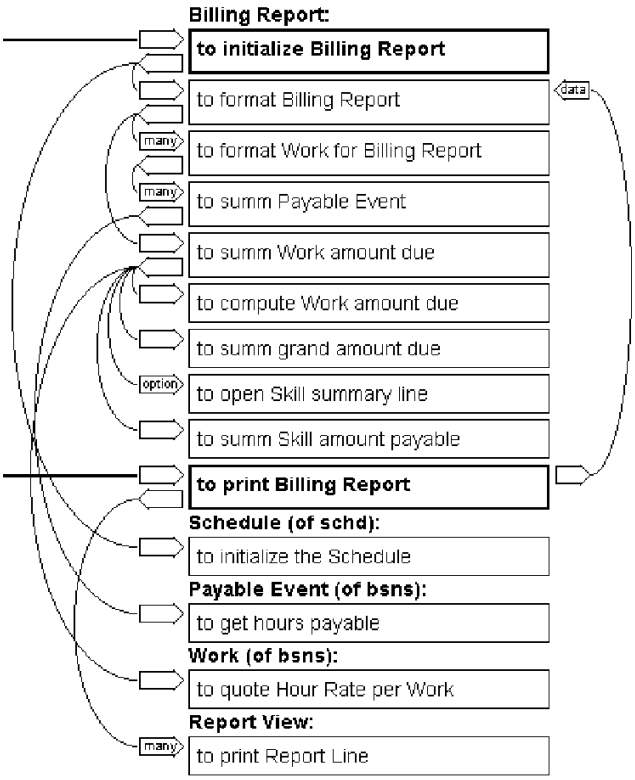
MANY Work,OPTION Client in list: **to format Work for Billing Report**
(format Work)
MANY: **to summ Payable Event**
to get hours payable
(by Payable Event of bsns)
to summ Work amount due
to quote Hour Rate per Work
(by Work of bsns)
to compute Work amount due
to summ grand amount due
OPTION: **to open Skill summary line**
to summ Skill amount payable

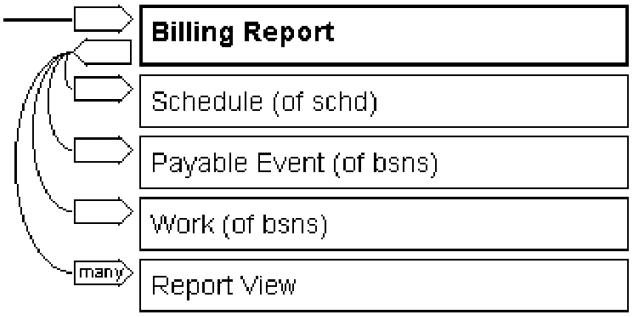
to print Billing Report
(print by Billing Report)

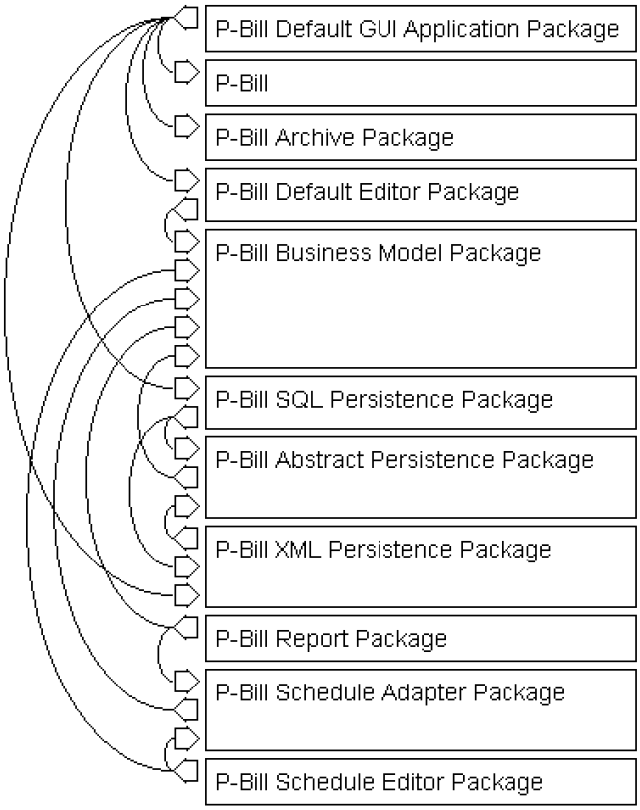
Skill usage contract:

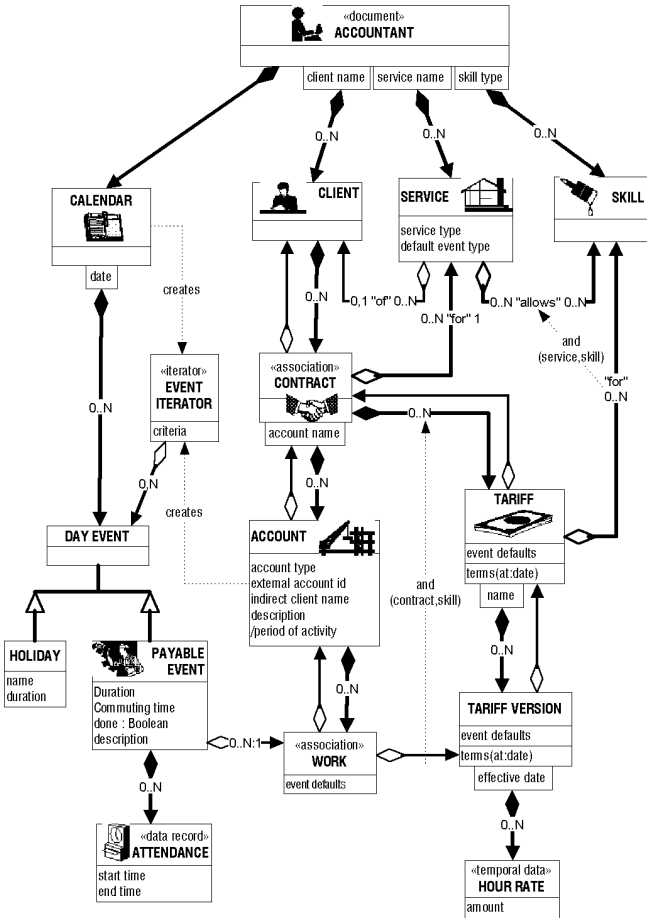
- *Transition when printing requested, leading to report visible.*
- *Transform of Report View.*

PROVIDED DATA FROM **to format Billing Report**
MANY: **to print Report Line**
(by Report View)









Project facts:

- The system has 36 useful entities with 190 specified useful skills (finally expected 453)
- *Entity Measure ("EM") computed for this system:* 12.58 skills (average number of skills per enti
- *Project constants:* EM lasts 3.0 days. The design takes 50 % of total effort.
- *Computed system development duration: 108.0 days linear (36 EM * 3.0).*

Generation 1 of 10: **"text table viewer"**

Package	Generation	Dev.	Fact	Duration EM	Design 50% EM	Impl. 50% EM	Remaining EM
P-Bill Business Model Assembly	1 (of 6) - "Table keeping Accountant"	Avner	-	0.6	0.3	-	0.3
P-Bill Abstract Persistence Assembly	1 (of 2) - "Read-only Accountant"	Avner	-	0.4 spec. 0.6 exp.	0.2 (75%)	-	0.3
P-Bill XML Persistence Assembly	1 (of 3) - "XML read-only Accountant"	Maurice	-	0.2 spec. 0.6 exp.	0.1 (25%)	-	0.6
			Total duration:	1.8			
			Total for developer "Avner":		0.5	-	0.7 (2.0 day
			Total for developer "Maurice":		0.1	-	0.6 (2.0 day