(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
6 April 2006 (06.04.2006)

PCT

(10) International Publication Number
WO 2006/035448 A2

(51) International Patent Classification: Not classified

(21) International Application Number:
PCT/IL2005/001056

(22) International Filing Date:
29 September 2005 (29.09.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/613,942 29 September 2004 (29.09.2004) US

(71) Applicant and
(72) Inventor: ATLAN, Moshe [IL/IL]; 156, 90641 Kokhav hash'har (IL).

(74) Agent: APPELFELD ZER LAW OFFICE; 29 Lilinblum, 65133 Tel-aviv (IL).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

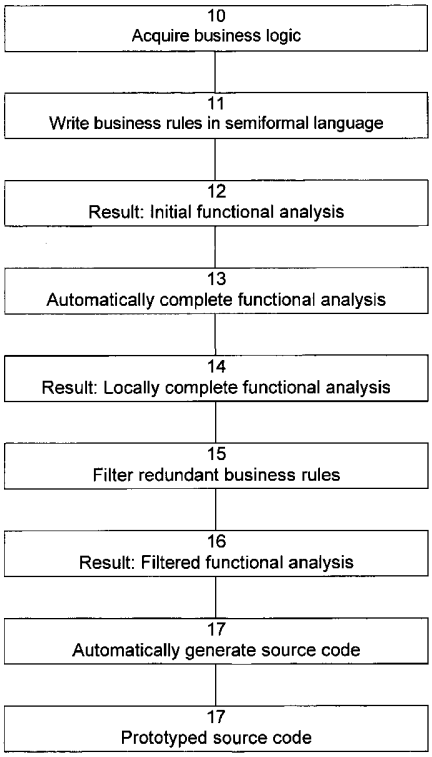
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: DYNAMIC FUNCTIONAL PROGRAMMING



(57) Abstract: The present invention provides a user-centric method and system for programming software modules based on automatic source code prototyping directly from functional requirements. It enables, before producing any source code, distribution of implemented business rules among predefined software modules. The invention is based on the formal characterization of the properties intrinsic to functional analysis and independent of any implementation techniques, conditioning the possibility of implementing business rules as a textually contiguous set of elementary instructions. The invention starts with an initial functional analysis wherein the functional requirements of a system are expressed in terms of rules. An automatic functional analysis completion algorithm produces a locally complete functional analysis from the initial functional analysis. Next, a filtered functional analysis presents the computational properties of a correct implementation from the locally complete functional analysis. Finally, the filtered functional analysis can be automatically translated into object oriented program source code.

WO 2006/035448 A2

DYNAMIC FUNCTIONAL PROGRAMMING

FIELD OF THE INVENTION

The present invention relates to software program development. More particularly it relates to a method for optimizing the modularity of object oriented programming.

BACKGROUND OF THE INVENTION

Reduced programming effort is an important goal of software engineering. It is commonly accepted that programming effort is strongly correlated to source code modularity, which enhances understandability and facilitates maintenance and reusability [10].Particularly, in areas with high process variability, such as enterprise resource planning (ERP), there is strong economic interest in finding methods for optimizing source code modularity[5].

Object Oriented Programming (OOP) techniques play a central role in modern software engineering because they enable a high degree of modularity. However, the full potential of OOP has yet to be realized. Indeed, there remain tangling code and cross-cutting concerns that break the modularity of source code. These issues are expressed by source code segments that are redundant and mix various concerns. They considerably increase the programming and maintenance effort because they require taking into account many concerns simultaneously and considering many potential consequences in other parts of the source code during a local modification.

Aspect oriented programming (AOP) techniques have been designed to overcome limitations of basic OOP techniques regarding the separation of concerns [7]. They allow aggregation of elementary instructions related to the same concern regarding to tangling and cross-cutting source code. A major implementation of the AOP paradigm is AspectJ [1][6][8], which is based on the abstraction of the functional interaction between classes or objects into Aspects, which are themselves Java classes. Aspects contain meta-programming rules that modify, either statically during compilation or dynamically at run time, the implementation of class or object behaviors.

Nevertheless, AOP does not have any methodology for determining objective conditions that justify Aspect implementation in a particular case [3]. Indeed, AOP tools extract the relevant Aspects from implemented source code. Thus, the Aspect structure is an emergent property of source code at a given implementation stage. Therefore, it is always possible to reject an Aspect model since it might result more from a bad object implementation than from intrinsic properties of the underlying functional analysis or OOP technique's limitations. Thus, AOP is still based on empirical observations and on a fundamental hypothesis according to which it is not always possible to implement the separation of concerns with basic OOP techniques. The lack of a theoretical basis for AOP prevents leading ERP programmers from developing AOP projects on an industrial scale.

Program slicing, which was introduced by Weiser in the 1980s [13] for procedural programming and extended to OOP by Larsen and Harrold in 1996 [9], is a powerful tool for giving a modular view of an isolated program's statement. The original research of Weiser was motivated by the need for tools facilitating the debugging operation. The slice of a program with respect to a set of program elements S is a projection of the program that includes only program elements that might affect (either directly or transitively) the values of the variables used at members of S. Slicing allows one to find semantically meaningful decompositions of programs, where the decompositions consist of elements that are not textually contiguous in the

program source code. For broad survey on program slicing and its applications, see Binkley and Gallagher's work [4], Tip's work [12], and the Wisconsin program slicing project home page [14].

Program slicing is mainly aimed at software documentation improvement while Aspect Oriented Programming is a paradigm aimed at improving source code modularity itself. Therefore, the application of program slicing to AOP is a novel research topic [2][15][16]. In these researches, program slicing (or similar algorithms) is used to provide AOP with analytic method to justify and to analyze aspect composition.

Our invention is based on a similar approach in order to decide and to implement automatically the best separation of concerns strategy with respect of the desired functional composition of the different software modules.

The present invention provides an innovative algebraic theory of functional analysis, which we refer to as dynamic functional programming (DFP). DFP makes it possible, working from a functional analysis and before writing any source code, to decide the best separation of concerns strategy at the functional requirement level. The starting point, as in program slicing, is the determination of the reference variable set and of the influence variable set for each variable assignment statement. Then an aggregation algorithm decides which statements will be implemented as a textually contiguous block of instructions and which ones will be delocalized. Finally, decision rules are used in the automatic implementation of the aggregated statement by choosing the application of aspect oriented programming or polymorphism.

The present invention optimizes the modularity level that OOP techniques can implement thanks to the invention's novel analytic method.

The present invention has many applications in software development. For example, it can provide ERP solutions with highly adaptive features at low acquisition and integration cost.

In the present invention one can choose, before coding, the rules embodied in the functional description to be represented in the final source code by an aggregated sequence of elementary instructions, referred in the following as aggregated rules, that significantly decreasing programming effort and maximizing the reusability of programs' modules.

In this optimized source code, any elementary instruction related to a given aggregated rule is grouped only with other elementary instructions related to the same business rule. We refer in the present invention to this property of source code as "functional connectivity," which means a formal definition of separation of concerns at the business rule level. The present invention optimizes the functional connectivity of the source code, that is a formal definition of separation of concerns at business rule scale.

When source code has optimized functional connectivity, a programmer does not need to painfully navigate when implementing a new business rule. Also, there is no need, before modifying a given business rule, to carry out an impact analysis to find which source code segments will be impacted by the modification.

Other objects and advantages of the present invention will become apparent after reading the present specification and reviewing the accompanying drawings.

BRIEF DESCRIPTION OF THE INVENTION

The present invention optimizes the modularity of object-oriented programming. This in turn substantially reduces software development and maintenance costs and enhancing modules reusability, particularly in areas with a high level of process variability, such as enterprise resource planning (ERP).

The present invention is a new separation of concerns strategy, referred to as dynamic functional programming (DFP), which provides the basis for an original automatic programming engine optimizing the source code modularity. The present invention substantially reduces programming effort by translating functional requirements expressed under the form of a business rule set into modular source code. In the generated source code, any elementary instruction related to a given rule is textually contiguous only to other elementary instructions related to the same rule. We refer to this property of the source code as functional connectivity.

In order to analyze the formal conditions of functional connectivity conservation in a program source code, we have designed an algebraic theory of functional analysis, which we refer to as Dynamic Functional Programming (DFP).

We define the implementation of a functional analysis in the context of a causal analysis of asynchronous state machine transitions.

We formulate the Causal Equivalence Principle (CEP), which provides the basis of an automatic functional analysis completion algorithm.

We prove the Reciprocal Cross Disaggregation (RCD) theorem, establishing the formal conditions under which it is impossible to aggregate the elementary instructions of two distinct business rules.

The RCD theorem, combined with CEP, provides the basis of an automatic software engineering process that is suitable for automated, highly-modular, source code prototyping.

DFP allows bypassing the negative effects of the RCD theorem. DFP can be advantageously integrated into existing off-the-shelf object designing and programming tools.

The added value of such a process is particularly high in areas with a high level of business process variability, such as ERP. For example, software customization no longer requires parameter setting, rather preprogrammed software module recombination. Such a software customization process, called Software Profiling, implies the best compromise between development, maintenance end integration costs [5].

BRIEF DESCRIPTION OF THE FIGURES

FIG. 1 is a generalized flowchart of a preferred embodiment of the present invention.

FIG. 2 is a diagram of a generic exception management process for eliminating tangled code coming from transactional logic management.

DETAILED DESCRIPTION OF THE INVENTION

Programming is a translation task that takes a functional description of a system's logic and culminates in elementary instructions in programming language syntax.

A common format for functional descriptions, particularly in ERP applications, is as a set of business rules. This disclosure of the present invention is based on a system functional description expressed as a set of business rules. It should be noted that from the I/S (Information System) perspective, a business rule pertains to the facts of the system that are recorded as data and to the constraints on changes to the values of those facts. The business perspective of business rules involves the behavior of people in the business system, where business should not be understood in a restrictive way: business refers to human activity in general.This detailed description starts with an overview of the present invention, followed by a description of the theoretical basis of the invention, and examples.

FIG. 1 summarizes the major steps by which the present invention converts the logic of a given system into modular source code:

Step 10 Acquire business logic: Acquire the logic of a system that is to be implemented into source code. This is usually provided by an expert in the domain of the system. For example, a business expert could describe in free text the parts of his business operation (such as billing) that he wants implemented in an ERP application.

Step 11 Write business rules in semiformal language: A designer (who is not obliged to have any programation skill) expresses the acquired system logic in terms of rules Ri for each variable Vi where a rule comprises a causal part and a functional part, as follows:

Causal part:

Precondition variables PRi: comprise those variables the recalculation of which causes recalculation of Vi (the Reference set in Program Slicing). For example, a variable *value-added-tax* is a precondition variable for a variable *subtotal* because a

change in the value of the former causes recalculation of the latter.

Postcondition variables POi: comprises those variables for which a recalculation of Vi causes their recalculation (the Influence set in Program Slicing). For example, *subtotal* is a postcondition variable of *value-added-tax*.

Functional part:

The formula according to which Vi is calculated. For example, *subtotal* = (1+*value-added-tax)* (before-taxes-price)

A given rule Ri is a set of elementary rules, each elementary rule being defined as comprising Vi and either a precondition variable PRi or a postcondition variable POi. Therefore, for a given rule Ri, the number of elementary rules equals the total number of its precondition and postcondition variables.

It will be noted that the designer is free to express each rule Ri in the manner he finds most convenient: by listing post conditions POi, preconditions PRi, or a combination of them.

The result at this point is initial functional analysis 12.

Step 13 automatically complete functional analysis: This step is performed automatically by a program implementing this part of the method of the present invention. When the designer in step 12 has expressed rule Ri of var Vi in terms of a set of preconditions PRi and a set of postconditions POi, then a completion algorithm, based on the causal equivalence principle (CEP) (defined later in this disclosure), adds Vi to the postcondition set of the rule of each of these PRi and adds Vi to the precondictions set of the rule of each of these POi.

Thus, each elementary rule is expressed twice in the locally complete functional analysis.

The result at this point is locally complete functional analysis 14.

Step 15 Filter redundant business rules: Filter the locally complete functional analysis according to the "reciprocal cross disaggregation" theorem (RCD) (defined later in this disclosure) by choosing, for every pair of equivalent elementary rules, one of them to implement. This task is performed automatically by a program implementing this part of the method of the present invention.

Filtering can be performed according to various strategies, according to the desired format for the final program source code. For example:

A implement the elementary rules involving preconditions and remove the elementary rules involving postconditions

B implement the elementary rules involving postconditions and remove the elementary rules involving preconditions

Strategy A or B will usually lead to redundant method calls in the final program source code. These redundant method calls, referred to as crosscutting instructions in the context of aspect oriented programming, break the source code modularity. Therefore, one could choose a different filtering strategy that optimizes the modularity (minimizes crosscutting):

C implement the elementary rules involving preconditions from precondition variables sets containing more than one variable and remove the equivalent elementary rules involving postconditions. After such a strategy has been applied to all the rules, implement the remaining elementary rules involving postconditions.

Strategy C presents the advantages of minimizing cross-cutting concerns and being entirely automatic but does not guaranty the desired

distribution of business rules among the software modules that we call functional distribution. In order to obtain precisely a given functional distribution, one has to choose the rule to be aggregated by hand :

D for every pair of equivalent elementary rules, choose by hand the one to implement. Thanks to the "reciprocal cross disaggregation" theorem (RCD) (defined later in this disclosure), the consequences of such an operation, on the aggregation of the other rules, is immediately knowable.

The result at this point is filtered functional analysis 16.

Step 17 Automatically generate source code: Generate the final program source code from the implementation results by translating the filtered rules into the programming language syntax. This task is performed by the automatically by a program implementing this part of the method of the present invention.

The final result is prototyped source code 18.

We now present a more detailed description of the underlying principles of the present invention.

In terms of object programming, a class is a set of ***n*** attributes, each of them being defined by its functional interactions with other attributes of the same or other classes. A class instance, an object, is obtained by assigning values to each of its attributes.

As a consequence, a class can be defined as an ***n*** dimension asynchronous automat.

This formalism of state automata presents the advantage of being dense in the programs space (since it allows defining all the basic logic operations of a Turing machine) and allows functional programming, which is very close to functional analysis, from a syntax point of view.

We can define a class C as the following system:

$$\begin{cases} s_1 = f_1(E) \\ \quad \vdots \\ s_i = f_i(E) \\ \quad \vdots \\ s_n = f_n(E) \end{cases}$$

where s_i is a state variable (property) and fi is its state transition function (method), which depends on the global system state E (state of the runtime environment).

E is defined by

$$E = \begin{pmatrix} \overrightarrow{s_{1,1}} & \cdots & \overrightarrow{s_{i,1}} & \cdots & \overrightarrow{s_{p,1}} \\ \vdots & & \vdots & & \vdots \\ \overrightarrow{s_{1,n_1}} & \cdots & \overrightarrow{s_{i,n_i}} & \cdots & \overrightarrow{s_{p,n_p}} \end{pmatrix}$$

where $\overrightarrow{s_{i,j}}=(s_{i,j,1},\ldots,s_{i,j,k},\ldots,s_{i,j,n})$ is the state vector (class's attributes values) of an instance ***j*** of the class **Ci**, written **Cij** and $s_{i,j,k}$ is the attribute ***k*** of the instance ***j*** of the class ***i***.

The iteration of the state variable $s_{i,j,k}$ is written $\omega_{i,j,k}$. The increment of $\omega_{i,j,k}$ triggers a transition of $s_{i,j,k}$ according to $f_{i,\bullet,k}(E)$ $\left(f_{i,j,k} = f_{i,l,k}\ \forall(j,l)\right)$.

We define the postcondition at attribute $s_{i,j,k}$ level as the attributes set $s_{l,m,n}$ that has to be recalculated consequent to a modification of the value of $s_{i,j,k}$. The postcondition of $s_{i,j,k}$ is written $\psi_{i,j,k}$ and is defined by :

$$\psi_{i,j,k} = \left\{s_{l,m,n}; \delta s_{i,j,k} \neq 0 \Rightarrow \omega_{l,m,n} = \omega_{l,m,n} + 1\right\}$$

With additive constraints on indexes ***i, k, l, n,*** one distinguishes class postcondition from attribute postcondition. For instance, if one forces only $k \neq n$, one defines the postcondition down to the lowest level between two attributes of the same class. By forcing $i \neq l$, one defines the postcondition between two different classes.

In order to build a virtual machine providing an execution environment for DFP, it is necessary to consider the class instance. However, such a consideration remains outside the scope of the present work, which considers only source code and not the execution environment. Thus, in order to simplify we will not continue to consider the instance index.

From here on, we will consider only the attribute ***i*** of the class ***j ;*** $s_{i,j}$ and its postcondition $\psi_{i,j}$. On the other hand, ***E*** will no longer be the set of instances state vectors in the execution environment, but rather the set of classes attributes, $E = \left\{s_{i,j}\right\}$.

The postcondition rate of $s_{i,j}$ is $\varepsilon_{i,j} = Card\left(\Psi_{i,j}\right)$.

By symmetry, the precondition of $s_{i,j}$ is given by the set of attributes $s_{k,l}$ whose transition implies the recalculation of $s_{i,j}$. It is written $\Omega_{i,j}$, and defined by:

$$\Omega_{i,j} = \{s_{k,l}; s_{i,j} \in \Psi_{k,l}\}$$

The precondition rate of $s_{i,j}$ is $\rho_{i,j} = Card(\Omega_{i,j})$.

In terms of AOP, the set $\Omega_{i,j}$ is the cross-cutting set induced by the attribute $s_{i,j}$. Indeed, without an aggregation strategy, the set of attributes $s_{k,l}$ whose transition implies the recalculation of $s_{i,j}$ is expressed in the source code by redundant calls to the calculation method of $s_{i,j}$ each time the value of an attribute $s_{k,l}$ changes.

The full business rule of the attribute $s_{i,j}$ is defined by:

$$M_{i,j} = (f_{i,j}; \Omega_{i,j}; \Psi_{i,j}).$$

where the calculation method $f_{i,j}$ constitutes the functional part of the rule, and where the precondition state transitions of $s_{i,j}$: $\Omega_{i,j}$. and its postcondition state transitions: $\Psi_{i,j}$. together constitute the causal part of the rule.

The complete functional description, called the functional analysis F, of an implementation ***Imp*** is the set of full business rules of all the attributes $F(\mathrm{Im}\, p) = \{M_{i,j}\}$. Since the expression for each attribute $s_{i,j}$, of the two sets $\psi_{i,j}$ and $\Omega_{i,j}$ describes completely and locally $M_{i,j}$, such a functional analysis is called locally complete.

We now describe the aggregation of the causal part of the business rules, putting aside the aggregation of the functional part. The aggregation conditions of the functional part depend more on considerations related to implementation techniques, such as transactional logic management or the localization of all the information necessary to compute locally a transition (see later discussion analyzing sources of tangled code).

Let the filtering rule operator be:

$R^{o,p}(M_{i,j}) = \{f_{i,j}; \Omega_{i,j} \cap o; \Psi_{i,j} \cap p\}$ where $(o,p) \in (P(E))^2$

which produces a sub rule of $s_{i,j}$, $M_{i,j}^{o,p}$ from its full rule $M_{i,j}$, considering only a portion of the consequences on the environment and a portion of the causes coming from the environment.

In order to simplify the representation of rules, we can rewrite some rules as follows:

- An unspecified precondition rule of $s_{i,j}$ comes from $o \neq \phi$ and $p = \phi$, it is written $M_{i,j}^{o,\bullet}$. It is elementary if $Card(o) = 1$.
- The aggregated precondition rule of $s_{i,j}$ comes from $o = \Omega_{i,j}$ and $p = \phi$, it is written $M_{i,j}^{\Omega}$.
- An unspecified postcondition rule of $s_{i,j}$ comes from $o = \phi$ and $p \neq \phi$, it is written $M_{i,j}^{\bullet,p}$. It is elementary if $Card(p) = 1$.
- The aggregated postcondition rule of $s_{i,j}$ comes from $o = \phi$ and $p = \Psi_{i,j}$, it is written $M_{i,j}^{\Psi}$.

We can now express the causal equality (indicated by $\overset{c}{=}$) between two unspecified elementary rules as:

$$\forall(i,j,k,l)\ \ M_{i,j}^{\{sk,l\},\bullet} \overset{c}{=} M_{k,l}^{\bullet,\{si,j\}}$$

Indeed:

$M_{i,j}^{\{sk,l\},\bullet}$ is the elementary precondition rule of $s_{i,j}$, which expresses the effect of a transition of $s_{k,l}$ on the transition of $s_{i,j}$.

$M_{k,l}^{\bullet,\{si,j\}}$ is the elementary postcondition rule of $s_{k,l}$ which expresses the effect of a transition of $s_{k,l}$ on the transition of $s_{i,j}$.

Thus these two distinct elementary rules of two distinct attributes are causally equal.

The disaggregation of a rule into a set of sub-rules coming from itself is called a homogeneous disaggregation.

Let the homogeneous disaggregation operator X of an unspecified rule $M_{i,j}^{o,p}$ be :

$$X\left(M_{i,j}^{o,p}\right)=\left\{R^{\{q\}\in o,\phi}\left(M_{i,j}^{o,p}\right)\bigcup R^{\phi,\{r\}\in p}\left(M_{i,j}^{o,p}\right)\right\}$$

which decomposes $M_{i,j}^{o,p}$ into the set of its $Card(o)$ elementary precondition rules $M_{i,j}^{\{q\},\bullet}$ and $Card(p)$ elementary postcondition rules $M_{i,j}^{\bullet,\{r\}}$.

From causal equality, we can deduce the causal intersection written $\overset{c}{\bigcap}$ of two unspecified rules by applying the homogeneous disaggregation operator:

Since $\left\{M_{i,j}^{\{sk,l\}\bullet} \in X\ \left(M_{i,j}^{o,p}\right)\right\} \overset{c}{=} \left\{M_{k,l}^{\bullet,\{si,j\}} \in X\ \left(M_{k,l}^{q,r}\right)\right\}$,

thus $M_{i,j}^{o,p} \overset{c}{\bigcap} M_{k,l}^{q,r} = \left\{M_{i,j}^{\{sk,l\}\bullet} \in X\ \left(M_{i,j}^{o,p}\right)\right\}$

We can now formulate the causal equivalence principle (CEP) according to which a precondition rule $M_{i,j}^{\Omega}$ exists, in a locally complete functional analysis, in at least two equivalent forms: its natural aggregated form and its disaggregated postcondition form written $M_{i,j}^{\Omega\to\Psi}$:

$$X\left(M_{i,j}^{\Omega}\right) \overset{c}{=} M_{i,j}^{\Omega\to\Psi} = \left\{R^{\bullet,\{si,j\}}\left(M_{l,m}^{\Psi}\right); s_{l,m} \in \Omega_{i,j}\right\}$$

Proof :

$$\left\{R^{\bullet,\{si,j\}}\left(M_{l,m}^{\Psi}\right); s_{l,m} \in \Omega_{i,j}\right\}$$

$$= \left\{M_{l,m}^{\bullet,\{si,j\}}; s_{l,m} \in \Omega_{i,j}\right\}$$

$$\overset{c}{=} \left\{M_{i,j}^{\{sl,m\}\in\Omega_{i,j},\bullet}\right\} = X\left(M_{i,j}^{\Omega}\right)$$

By symmetry, the CEP gives the disaggregated precondition form of an integrated postcondition rule:

$$X\left(M_{i,j}^{\Psi}\right) \overset{c}{=} M_{i,j}^{\Psi\to\Omega} = \left\{R^{\{si,j\}\bullet}\left(M_{l,m}^{\Omega}\right); s_{l,m} \in \Psi_{i,j}\right\}$$

In contrast to homogeneous disaggregation of a rule into a set of elementary rules coming from itself, heterogeneous disaggregation disaggregates a rule into sub-rules coming from other rules according to the CEP.

This is precisely the form of disaggregation (or inversely, aggregation) that induces constraints that condition the conservation of source code functional connectivity.

Indeed, when a programmer tries to produce the most modular source code possible, he attempts to aggregate elementary instructions according to the cardinality of the precondition and postcondition sets. In other words, he tries to conserve the functional connectivity by implementing the locally complete functional analysis. However, such an implementation is necessarily redundant according to the CEP and furthermore breaks the program consistency.

As a consequence, during programming, the programmer propagates in his source code some disaggregation constraints that force him to reject some previous implementation options and he is obliged, at the same time, to give up the functional connectivity.

We will now formalize these disaggregation constraints.

Let $\mathrm{Im}\, p(F)$ be an implementation of a functional analysis $F = \{M_{i,j}\}$:

$$\mathrm{Im}\, p(F) = \left\{M_{i,j}^{o,p}\right\}$$

It is consistent (not redundant) if:

$$\forall\ \left\{M_{i,j}^{o,p}, M_{k\neq i,l\neq j}^{q,r}\right\} \in \mathrm{Im}\, p(F),\ X\left(M_{i,j}^{o,p}\right) \overset{c}{\bigcap} X\left(M_{k,l}^{q,r}\right) = \phi$$

It is complete if:

$$\forall M_{i,j} \in F,\ \exists\ \left\{M_{k,l}^{o,p}\right\} \subset \mathrm{Im}\, p(F)\ \textit{such that}\ :\ \left\{X\left(M_{k,l}^{o,p}\right)\right\} \overset{c}{=} X\left(M_{i,j}\right)$$

We write a consistent and complete implementation of F , $\mathrm{Im}\, p_{cc}(F)$.

The reciprocal cross disaggregation (RCD) theorem asserts that it is impossible to aggregate at the same time the postcondition part of a rule $M_{i,j}$ and the precondition part of another rule $M_{k,l}$ in a consistent implementation, if a transition of the attribute $s_{i,j}$ implies a transition of the attribute $s_{k,l}$:

$$\left\{M_{i,j}, M_{k\neq i,l\neq j}\right\} \in F \quad \textit{and}\ s_{i,j} \in \Omega_{k,l} \Rightarrow \left\{M_{i,j}^{\Psi}, M_{k,l}^{\Omega}\right\} \notin \mathrm{Im}\, p_{cc}(F)$$

Proof :

$$X\left(M_{k,l}^{\Omega}\right) = \left\{M_{k,l}^{\{so,p\} \in \Omega k,l,\bullet}\right\}$$

since $s_{i,j} \in \Omega_{k,l}$

thus $M_{k,l}^{\{si,j\}\bullet} \in X\left(M_{k,l}^{\Omega}\right)$.

Since $M_{k,l}^{\Omega} \overset{c}{\bigcap} M_{i,j}^{\Psi} = \left\{M_{k,l}^{\{si,j\}\bullet} \in X\left(M_{k,l}^{\Omega}\right)\right\}$,

it works out that: $M_{k,l}^{\Omega} \overset{c}{\bigcap} M_{i,j}^{\Psi} \neq \phi$

The RCD theorem provides the constraints to be satisfied by the aggregation algorithm that produces a consistent implementation from the locally complete functional analysis. The different aggregation strategies, also called filtering strategies, are presented above as strategies A, B, C and D.

Once the aggregation strategy has been designed at the level of the causal part, we still have to decide the implementation techniques for merging the elementary instructions in the source code. We identify three such techniques:

- Postcondition aggregation. This is the most natural way to program. It doesn't imply any particular object programming or aspect techniques. It is enough to merge at the same place, in the source code in the attribute's transition method definition, all the method calls triggered by the transition of the considered attribute.
- Polymorph precondition aggregation. In some cases, aggregation can be implemented by polymorphism, merging the method calls of an attribute's precondition rule. This aggregation technique is possible only if all the attributes of the considered precondition set are inherited from a unique attribute of a common ancestor class.

  Indeed, in this case the cross-cutting induced by a precondition set containing more than one element is an artifact coming from inheritance. It is therefore natural to solve it with a polymorphism aggregative mechanism.
- Aspect precondition aggregation. In cases where precondition aggregation cannot be implemented by polymorphism without altering the business logic contained in the class model, it is necessary to implement aggregation using AOP techniques.

Now that the problem of aggregation of the business rule's causal part has been studied, it remains to analyze the functional part's aggregation, which conditions the emergence of tangling code. Like cross-cutting concerns, tangling code breaks the source code modularity. The functional part of a rule $M_{i,j} = (f_{i,j}; \Omega_{i,j}; \Psi_{i,j})$ is $f_{i,j}$.

Tangled code occurs when the attribute's value calculation is delocalized from the place where the value is set.

The analysis of the tangling code emergence conditions at implementation level make appear two main sources of tangled code:

- The first source comes from response time optimization.
- The second source comes from transactional logic management.

In both cases, automatic aggregation of the source code remains possible since the necessary information is included in the dynamic functional program.

The first source of tangled code, response time optimization, appears when the programmer is led to delocalize the calculation of an attribute value. This happens when the necessary information for optimal calculation is not available locally, where the value is set.

For instance, for optimization reasons one can choose to implement an attribute's transition in an incremental way at the place where a differential appears at the place of a precondition attribute's value calculation. We find this case in the following example: an invoice header needs to recalculate the total weight of the group of delivered products included in the invoice lines, each time one of these products or its delivered quantity changes. This calculation could be implemented very simply and locally as the sum of every line's weight. But in order to optimize this calculation by avoiding the need to load every invoice's line each time one of them changes, the programmer

may be led to implement it in an incremental way in the invoice's line when either the article changes or the quantity changes. Such an implementation is optimal in terms of calculation time, but clearly induces tangled code because the calculation of the invoice header total weight is delocalized among the invoice lines.

A good compromise allowing simultaneous optimization of the calculation time and avoidance of tangled code is to conserve in each object the differential value (or previous value for symbolical attributes) of every attribute at any time. Then, it is possible to implement the calculation of every transition in an optimal and local way by passing to the method a reference of the object where a differential appeared.

Since all the necessary information is naturally contained in the dynamical functional program, it is easy to perform the automatic translation from absolute formula to incremental formula at the source code prototyping stage.

The second source of tangled code is the transactional logic management, which aims to guarantee data coherence between various interactive objects. As in the first case, this leads the programmer to delocalize the calculation instructions from where the attribute's value is set. Such a delocalized implementation is motivated by the precedence constraints arising from the need to first perform all the calculations and then verify the data relevance, while setting the values is postponed until after the verification.

This second source of tangled code can be eradicated by a generic exception management procedure (FIG. 2) implemented on an application server 30 using a data base management system (DBMS) 60 and guaranteeing both data coherence and synchronization between DBMS 60, the dynamic memory of the application server, and graphical user interface (GUI) 50. (See also call to such a procedure in sample code implementation of rule no. 6 provided in discussion of TABLE 3 later in this description.)

The procedure is as follows: state transition is induced 40 at object level A from GUI 50, which opens a transaction T. A causal chain is triggered in the transaction T from object A to object C. When an exception 34 is raised at object level C, it cancels the transaction T by a rollback mechanism 32 at DBMS 60 level, and then reloads 36 the objects in transaction T in the application server's dynamic memory, and finally propagates a reload signal to the opening transaction point at GUI 50 level.

The transactional structure is naturally represented in the dynamic functional program's causal graph as the different paths from root nodes (attributes $s_{i,j}$ with an empty precondition set, $\rho_{i,j} = 0$) to terminal nodes (attributes $s_{i,j}$ with an empty postcondition set, $\varepsilon_{i,j} = 0$). Thus, it is possible to automatically generate instructions requesting the activation of an application server's synchronization mechanism at source code prototyping stage.

Such a process allows the programmer to ignore transactional logic management and lets him aggregate freely the implementation of the functional part.

At this point, we have:

- An aggregating algorithm that rearranges the causal parts of the business rules contained in a given functional analysis, in order to

optimize the functional connectivity of the implementation by removing cross-cutting concerns.

- Decision rules to guide the choice among different implementation techniques to implement the aggregation strategy produced by the aggregating algorithm.
- Heuristic rules for tangle code sources identification, leading to the aggregation of the business rules' functional part.

Thus we are ready, using the class model and dynamic functional program, to automate the prototyping of the source code that implements the best possible level of separation of concerns, on the scale of business rules.

We now provide a simple billing example to illustrate the aggregation algorithm. This example is based on a few business rules related to various bills such as invoices, orders and delivery bills with product movement in the warehouses.

TABLE 1 (see Appendix) is an example of a look-up table mapping the logic of business rules describing a business system and expressed in natural language (functional analysis) (right column) into a dynamic functional program (DFP) (left column) comprising an initial functional analysis, in accordance with a preferred embodiment of the present invention.

In the DFP column, after the attribute and its transition function declaration, its postcondition set is given (shown in braces – {}) according to the attributes set to be recalculated after the transition. The table also includes validity constraints (shown in brackets – []). This last feature will allow us to illustrate the abstraction of the transactional logic at application server level used to separate functional concerns from non-functional concerns in order to avoid tangled code.

At the dynamic functional program level, it is clear that no programming option has been performed yet. At this level, one finds only the pure functional description of the problem. It is still possible to implement it in many different ways.

Still, it is possible to forecast from this functional description, the cross-cuttings and tangling code problems the programmer will have to deal with at the programming stage.

TABLE 2 (see Appendix) is the locally complete functional analysis (described by a causal graph giving, for each attribute: its postconditions and preconditions) produced by local completion of the initial functional analysis by application of the CEP (an attribute of the precondition or postcondition set is prefixed only if its class is different from that one of the attributes affected by the business rule).

The table is a representation of the bi-directional transition. It immediately makes apparent the attributes inducing cross-cuttings as well as the disaggregation constraints.

For instance, rule no. 5 states that modification of a bill header's after-tax price (*at*) does not imply any other transition, it is a terminal node of the graph. However, every modification of the header's VAT (*vat*) or before-tax price (*bt*) implies the calculation of the new after-tax price (*at*). Thus, the management of this attribute's value is a cross-cutting source since it contains more than one element in its precondition set (VAT and before-tax price).

To eradicate this cross-cutting source, it is necessary to implement this business rule according to its precondition aggregated form. At the same time, in order to satisfy the RCD constraints, the equivalent elementary subrules set - that is, the occurrences of the attribute *SaleBilHeader.at* in the postconditions set of attributes *vat* and *bt*- in the rules no. 3 and 4.

Since the postcondition sets of the attributes VAT (*vat*) and before tax (*bt*) (rules no. 3 and 4) contain only one element (the after-tax-price: *at*), the disaggregation of the postcondition parts of rules no. 3 and 4, by removing an attribute from them, conserves the functional connectivity.

Thus, the local completion of functional analysis by applying CEP and filtering under RCD constraints eradicates the cross-cutting source coming from the management of the after taxes (*at*) attribute of the class bill header.

TABLE 3 (see Appendix) is the filtered functional analysis, which exhibits the computational properties (consistency and completeness) of a correct implementation of the initial functional analysis. This filtered functional analysis results from filtering under RCD constraints, according to the filtering strategy C as referred above, of the whole set of locally complete business rules presented in TABLE 2.

One can verify this implementation is complete and consistent. That is to say, it contains one and only one occurrence of each business rule present in the original functional analysis.

First of all, all cross-cutting has been eradicated; each of the attributes of the postcondition sets column occurs only one time. This will yield a source code without multiple calls to the same method, which characterizes cross-cutting. Second, every conserved postcondition or precondition rule has been conserved according to its aggregated form.

By means of this case study we have demonstrated our main result: the aggregation algorithm converges toward the most modular way of implementing a given functional requirement.

It remains only, for each rule, to choose a practical technique for implementing the produced source code aggregation strategy and to

generate the corresponding source code according to the syntax of the selected programming language. As mentioned earlier in this disclosure, aspect-oriented, polymorphic, and basic object-oriented programming techniques for doing so are well known to those skilled in the art.

The following samples (with reference to TABLE 3) illustrate the produced source code implemented in Java (trademark of Sun Microsystems, Inc.) programming language.

The following is a precondition aggregate example implementing rule no. 6. In this case, the aggregation of the causal part of the rule is implemented in the most natural way by calling the methods updating the line's before-taxes-price and the header's weight just after the update of the line's quantity.

```
public class SaleBilLine {
    Public void setQ (BigDecimal q) throws Exception {
        This.DeltaQ = q – this.q;
        this.q = q ;
        try    {getSaleBillHeader.updateWeight (this);}
        catch  (RollBackException)                                    e)
        {getSession().rollback(e.getTransaction); throw e}
        this.computeBeforeTaxesPrice();
    }
}
```

This simple code example also demonstrates the aggregation of the functional part of the rule managing the header's weight thanks to differential calculus and to the transactional logic abstraction (*catch* etc.) at application server level as described previously (with reference to FIG. 2).

Each time a line's quantity is updated, the quantity's differential is also updated. The header weight's method receives a reference to the line itself that calls it. Then the weight update at header level is performed in an incremental way, without requiring loading of all the lines, and locally (see the method's call above and the method implementation below).

Moreover, the validity constraint on the header weight is managed according to transactional logic thanks to the exception propagation mechanism described previously (with reference to FIG. 2) without any tangling code. This mechanism refers, in the previous code example, to the rollback invocation inside the try-catch block. The following code example shows the incremental calculus of the header's weight attribute and the origin of the exception propagation mechanism by throwing a RollBackException.

```
public class SaleBillHeader {
        void updateWeight (SaleBilLine sbl) throws Exception {
            weight += sbl.getDeltaQ () * sbl.getArticle.getUnitWeight();
            if (weight.doubleValue() > maxW)
                throw new RollBackException (this.getTransaction());
        }
}
```

This example is a good illustration of how the information contained in the PFD allows simultaneous aggregation of the causal part of rule no. 6 and of the functional part of rule no. 2. This is accomplished while also optimizing the computational time and satisfying the transactional logic thanks to its abstraction at application server level.

The following is an Aspect-based precondition aggregate example implementing rule no. 5. The condition of Aspect requirement is satisfied (the

two attributes VAT (*vat*) and before-taxes price (bt) don't come from a common ancestor).

```
public aspect SaleBilAfterTaxesPrice {
    pointcut calculationPredicate () :
        call (void SaleBilHeader.setVAT ()) ||
        call (BigDecimal SaleBilHeader.computeBeforeTaxesPrice ()
    ;
    after () : calculationPredicate () {
        SaleBilHeader.computeAfterTaxesPrice ();
    }
}
```

The following is a polymorph precondition aggregate example implementing rule no. 13. The condition of Aspect requirement is not satisfied (the two attributes *SaleDeliveryBLine.q* and *SaleInvoiceLine.q* come from one common ancestor *SaleBilLine.q*). Thus this aggregate can be conserved by polymorph implementation.

```
public class SaleDeliveryBLine extends SaleBilLine {
    /* Delivery Bill specific operations and SaleBilLine abstract methods implementation */
}
public class SaleInvoiceLine extends SaleBilLine {
/* Invoice specific operations and SaleBilLine abstract methods implementation */
}
public abstract class SaleBilLine {
```

```
public void setQuantity (BigDecimal q) {
        deltaQ = q - this.q;
        this.q = q;
        String className = this.getClass().getName();
        Warehouse warehouse = getWarehouse;
        If (className.compareTo ("SaleDeliveryBLine")     == 0 ||
            className.compareTo ("SaleInvoiceLine")       == 0
        )
            warehouse.updateQuantityOut (deltaQ);
}

}
```

In conclusion, the DFP theory of the present invention makes clear the major source of pain in object programming. If the programmer doesn't analyze the disaggregation constraints that are self-contained in functional analysis, prior to coding, he discovers these RCD constraints while coding and is continuously led to reject previous implementation options.

Thanks to DFP, it is now possible to anticipate these constraints prior to producing any source code, which sets the basis of a new methodological framework which embeds separation of concerns techniques such as aspect oriented programming.

It is possible to consider two different approaches exploiting the DFP Theory in order to bypass the negative effects of the RCD Theorem on functional connectivity.

The first approach is automatic source code prototyping, which maximizes the separation of concerns from functional analysis.

A DFP graph, which is a graphical designing tool, with its integrated source code prototyping mechanism could be mounted on existing UML object modeling tools. For the business rules that not have been implemented as aggregated set of instructions, the program slicing tools may provide the textually contiguous view of them in the source code. This can provide a business-rule-driven source code browser for object programming frameworks.

The second approach is a runtime environment for DFP aimed at making functional concerns the central concerns of the software engineering process, moving aside the technical considerations, and providing a user-centric way of programming software.

Since a dynamic functional program defines the functional requirements with a set of formalized and local business rules, it is possible to design a virtual machine that is easily programmable considering only functional requirements.

The programming language could be produced by hybridizing a functional language such as LISP or PROLOG and an object language such as C++ or Java.

By reducing the required programming skill, the methodology and associated production tools of the present invention make business expertise play a more important role in software engineering processes. This increases quality, decreases production and maintenance costs.

It should be clear that the description of the embodiments and attached Figures set forth in this specification serves only for a better understanding of the invention, without limiting its scope as covered by the following Claims.

It should also be clear that a person skilled in the art, after reading the present specification could make adjustments or amendments to the attached Figures and above described embodiments that would still be covered by the following Claims.

REFERENCES

[1]AspectJ home page. http://www.aspectj.org.

[2]Balzarotti D. and Monga M. 2004. Using program slicing to analyze aspect-oriented composition. In Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004.

[3]Bakker J. and Tekinerdoğan B. *2005. Characterization of Early Aspects Approaches*. In *Proceedings of the International Conference on Aspect-Oriented Software Development, 2005, March 14-18, Chicago, USA*. 2005.

[4]Binkley D.W. and Gallagher K.B. 1996. Program Slicing. Advances in Computers, Vol. 43:pp. 1–50.

[5]Bouaziz P. and Seinturier L. From Software Parameterization to Software Profiling. 2001. In Awais Rashid and Lynne Blair (eds.). Proceedings of the International Workshop Lancaster University, UK. pp 8 – 12.

[6]Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J. and Griswold W. 2001. An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Lecture Notes in Computer Science.

[7]Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.M. and Irwin J. 1997. Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Lecture Notes in Computer Science. pp 220-242. Springer.

[8] Laddad R. 2003. AspectJ in Action. Manning. ISBN 1-930-11093-6.

[9] Larsen L. and Harrold M.J. 1996. Slicing object-oriented software. In Proceedings of the 18th international conference on Software engineering, Vol. 21, pp 495-505

[10] Parnas D. 1972. On the criteria to be used in decomposing systems into modules. In Communications of the ACM 15, 12. pp 1053-1058.

[11] Rumbaugh J., Jacobson I. and Booch G. 2004. Unified Modeling Language Reference Manual. Addison-Wesley.

[12] Tip F. 1995. A Survey of Program Slicing Techniques. Journal of Programming Languages, Vol.3, No.3, pp.121-189.

[13] Weiser M. Program Slicing. 1981. in Proceedings of the Fifth International Conference on Software Engineering, pp 439-449.

[14] Wisconsin Program-Slicing Project home page. http://www.cs.wisc.edu/wpis/html/#disclaimer.

[15] Zhang C. and Jacobsen H.-A. 2003. A Prism for Research in Software Modularization Through Aspect Mining. Technical Communication, September 2003, Middleware Systems Research Group, University of Toronto.

[16] Zhao J. 2002. Slicing Aspect Oriented Software. In Proceeding of the 10th IEE International Workshop of Programming Comprehension. Pages 251 - 260, June 2002.

APPENDIX

TABLE 1

DFP	Business rules in natural language
SaleBillHeader	**Sale bill header**
customer; {vat}	• If the customer changes, update the VAT.
weight = Sum line $_{i = 1, NbLines}$.(q * art.weight); {}; [weight < MaxWeight]	• The total weight is the sum of line's quantity multiplied by the line's unit article weight over the lines; Updating the total weight does not affect other attributes; The total weight should be less than the parameter MaxWeight value.
vat = customer.vat; {at}	• By default, the VAT is the customer's VAT; If the VAT changes, update the after-taxes-price (at).
bt = Sum (line $_{i = 1, NbLines}$.bt); {at}	• Before-taxes-price (bt) = sum of lines bt prices; if the bt changes, update the at price.
at = ht * (1 + vat / 100);	• After-taxes-price = bt * (1 + vat / 100) ;
SaleBillLine	**Sale bill line**
q = 1; {bt, header.weight}	• By default, the quantity is 1. If it changes, update the bt price and the header's weight.
art; {q}	• At article setting, initialize the quantity.
bt = q * art.price; {header.bt}	• bt = quantity * article unit price; If the bt price changes, update the header before-taxes-price.
SaleOrderLine extends SaleBillLine	**Order line** (is a particular case of the general sale bill line)
q; {warehouse.qRes}	• If the quantity changes, update the reserved quantity of the associated warehouse.

TABLE 1

DFP	Business rules in natural language
SaleDeliveryBLine extends SaleBillLine q; {warehouse.qOut}	**Deliver bill line** (is a particular case of the general sale bill line) • If the quantity changes, update the quantity that went out of the associated warehouse.
SaleInvoiceLine extends SaleBillLine q; {warehouse.qOut}	**Invoice line** (is a particular case of the general sale bill line) • If the quantity changes, update the quantity that went out of the associated warehouse.
WareHouse qIn ; {q} qOut ; {q} qRes ; {qAvl} qAvl = q – qRes ; q = qIn – qOut ; {qAvl}	**Warehouse** • If the entered quantity changes, update the current quantity. • If the quantity that went out of the warehouse changes, update the current quantity. • If the reserved quantity changes, update the available quantity. • Available quantity = current quantity – reserved quantity. • Current quantity = entered quantity – left quantity; if the current quantity changes, update the available quantity.

TABLE 2

	Business Rule	Postcondition	Precondition
1	SaleBilHeader.customer	vat	
2	SaleBilHeader.weight		SaleBilLine.q
3	SaleBilHeader.vat	at	customer
4	SaleBilHeader.bt	at	SaleBilLine.bt
5	SaleBilHeader.at		vat, bt
6	SaleBilLine.q	SaleBilHeader.weight, bt	art

TABLE 2			
	Business Rule	**Postcondition**	**Precondition**
7	SaleBilLine.art	q	
8	SaleBilLine.bt	SaleBilHeader.bt	q
9	SaleOrderLine.q	Warehouse.qRes	
10	SaleDeliveryBLine.q	Warehouse.qOut	
11	SaleInvoiceLine.q	Warehouse.qOut	
12	Warehouse.qIn	q	
13	Warehouse.qOut	q	SaleDeliveryBLine.q, SaleInvoiceLine.q
14	Warehouse.qRes	qAvl	SaleOrderLine.q
15	Warehouse.qAvl		qRes, q
16	Warehouse.q	qAvl	QIn, qOut

TABLE 3			
	Business Rule	**Postcondition**	**Precondition**
1	SaleBilHeader.customer	vat	
2	SaleBilHeader.weight		
3	SaleBilHeader.vat		
4	SaleBilHeader.bt		
5	SaleBilHeader.at		vat, bt
6	SaleBilLine.q	SaleBilHeader.weight, bt	
7	SaleBilLine.art	q	
8	SaleBilLine.bt	SaleBilHeader.bt	
9	SaleOrderLine.q	Warehouse.qRes	

TABLE 3

	Business Rule	Postcondition	Precondition
10	SaleDeliveryBLine.q		
11	SaleInvoiceLine.q		
12	Warehouse.qIn		
13	Warehouse.qOut		SaleDeliveryBLine.q, SaleInvoiceLine.q
14	Warehouse.qRes		
15	Warehouse.qAvl		qRes, q
16	Warehouse.q		QIn, qOut

CLAIMS

1. In a computer system having a processor and memory, a method for dynamically optimizing the modularity of object oriented implementation of functional requirements, said method comprising of:

- defining each class as asynchronous state automat, wherein state is the set of class's attributes' values;
- for each given attribute, in accordance with the functional requirements, defining the precondition set of all attributes wherein a change in their value affect the value of said given attribute;
- for each given attribute, in accordance with the functional requirements, defining the postcondition set of all attributes wherein a change in the value of said given attribute affect the value of attributes of the postcondition set;
- for each attribute defining a rule, in accordance with the functional requirements, said rule comprising the precondition set of said attribute, postcondition set of said attribute and the formula according to which said attribute changes its value.
- filtering each pair of elementary rules, said elementary rule comprising either a precondition of a given attribute or a post condition of another attribute, each expressing the same causality, where filtering comprises selecting whether to implement the precondition or the postcondition of the pair according to a given filtering strategy;
- generating the program source code implementing the selected postconditions and preconditions.

2 The method of claim 1 wherein the rule for any attribute $s_{i,j}$, wherein i is the class index and j is the attribute index, is defined as:

$M_{i,j} = (f_{i,j}; \Omega_{i,j}; \Psi_{i,j})$.

where $f_{i,j}$ is the transition method for calculating the new value of attribute $s_{i,j}$, $\Omega_{i,j}$ is the precondition set of $s_{i,j}$, and $\Psi_{i,j}$. is postcondition set of $s_{i,j}$.

3. The method of claim 2 wherein the calculation of a new value of $s_{i,j}$ is triggered by a change in a value of an attribute in the precondition set $\Omega_{i,j}$.

4. The method of claim 1 wherein the calculation of a new value of $s_{i,j}$ triggers a change in the value of each attribute in the postcondition set $\Psi_{i,j}$.

5. The method of claim 1 wherein the given filtering strategy comprises removing the elementary rules involving postconditions.

6. The method of claim 1 wherein the given filtering strategy comprises removing the elementary rules involving preconditions.

7. The method of claim 1 wherein the given filtering strategy comprises :

- removing the elementary rules involving postconditions equivalent to the elementary rules involving preconditions from precondition variables sets containing more than one variable.
- Removing the elementary rules involving preconditions from precondition variables sets containing only one variable.

8. The method of claim 1 wherein the given strategy comprises selecting from a graphical interface, the elementary rule to be implemented among each equivalent elementary rules pair.

9. The method of claim 2 wherein the generation of the program source comprises implementing the postcondition set and precondition set, of every attribute, remaining after filtering.

10 The method of claim 9 wherein the implementation of the postcondition set of a given attribute comprises merging in the given attribute's transition method source code, all the transition method calls of all the attributes in the post condition set, said calls triggered by a change in the value of the given attribute .

11. The method of claim 9 wherein the implementation of the precondition set of a given attribute comprises calling only one time the transition method of the given attribute, by means of either polymorph merging or aspect merging, after a value change occurs in anyone of the attributes of the precondition set.

12. The method of claim 11 wherein polymorph merging, which is only possible if all the attributes of the given attribute's precondition set are inherited from the same common ancestor, comprises calling only one time the transition method of the given attribute in the transition method of the common ancestor of all the attributes of the precondition set.

13. The method of claim 11 wherein aspect merging comprises calling only one time the transition method of the given attribute in the after() clause of an aspect pointcut containing a call statement for every transition method of every attribute of the given attribute's precondition set.

14. The method of claim 11 further adapted for managing generic exceptions by removing tangled code arising from transactional logic management, the method comprising providing a system comprising a graphical user interface for system management, an application server, and a relational database management module, the system configured so that a signal from the graphical user interface opens a transaction by calling a method of an object on the application server, the called method in turn calling other methods of other objects or of the same object, the sequence of method calls continuing up to a final call

determining the natural end of the transaction; the managing generic exceptions method further comprising:

upon an exception, at an object method level in a current transaction, canceling the transaction at the relational database management module level by a rollback mechanism;

reloading the objects involved in the transaction to the application server's dynamic memory from the relational database management module and back-propagating a reload signal to the graphical user interface, forcing the graphical user interface to refresh the displayed data according the state of objects in application server dynamic memory, after they have been reloaded.

15. The method of claim 1 implemented on a computer-readable medium.

16. A computer system for dynamic optimization of the modularity of object oriented implementation of functional requirements, said system comprising a processor and memory, the processor configured to:

- define each class as asynchronous state automat, wherein state is the set of class's attributes' values;
- for each given attribute, in accordance with the functional requirements, define the precondition set of all attributes wherein a change in their value affect the value of said given attribute;
- for each given attribute, in accordance with the functional requirements, define the postcondition set of all attributes wherein a change in the value of said given attribute affect the value of attributes of the postcondition set;
- for each attribute define a rule, in accordance with the functional requirements, said rule comprising the precondition set of said attribute, postcondition set of said attribute and the formula according to which said attribute changes its value.

- filter each pair of elementary rules, said elementary rule comprising either a precondition of a given attribute or a post condition of another attribute, each expressing the same causality, where filtering comprises selecting whether to implement the precondition or the postcondition of the pair according to a given strategy;
- generate the program source code implementing the selected postconditions and preconditions.

17. The system of claim 16 adapted for automated source code prototyping, the system further comprising a computing device adapted to provide a graphical designing tool with an integrated source code prototyping mechanism.

18. The system of claim 17, configured to allow a textually contiguous view of business rule implementations in source code, thereby providing a business-rule-driven source code browser for object programming frameworks.

19. The system of claim 17 wherein the graphical designing tool is implemented on an object modeling tool.

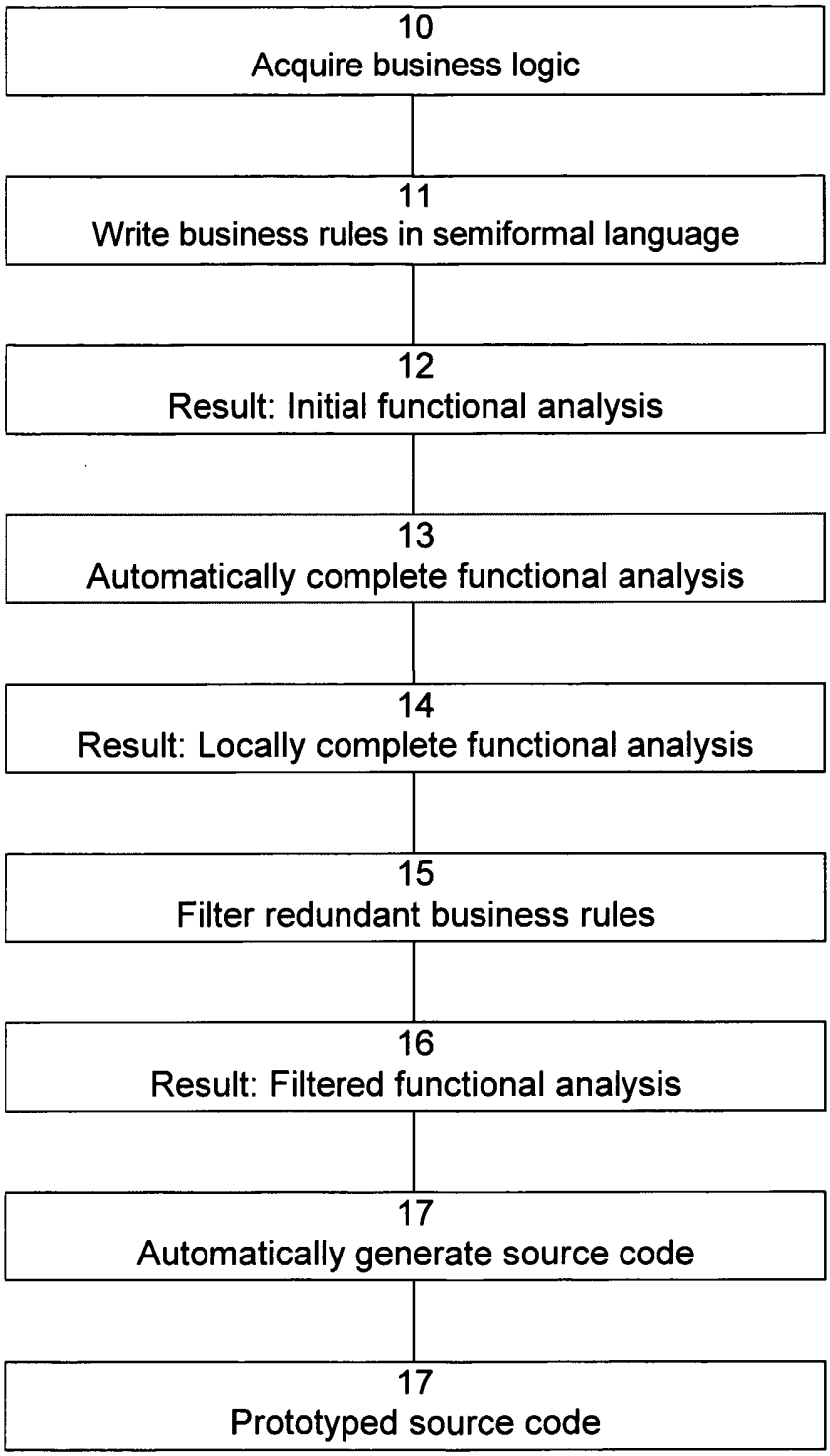

10
Acquire business logic
11
Write business rules in semiformal language
12
Result: Initial functional analysis
13
Automatically complete functional analysis
14
Result: Locally complete functional analysis
15
Filter redundant business rules
16
Result: Filtered functional analysis
17
Automatically generate source code
17
Prototyped source code

FIG. 1

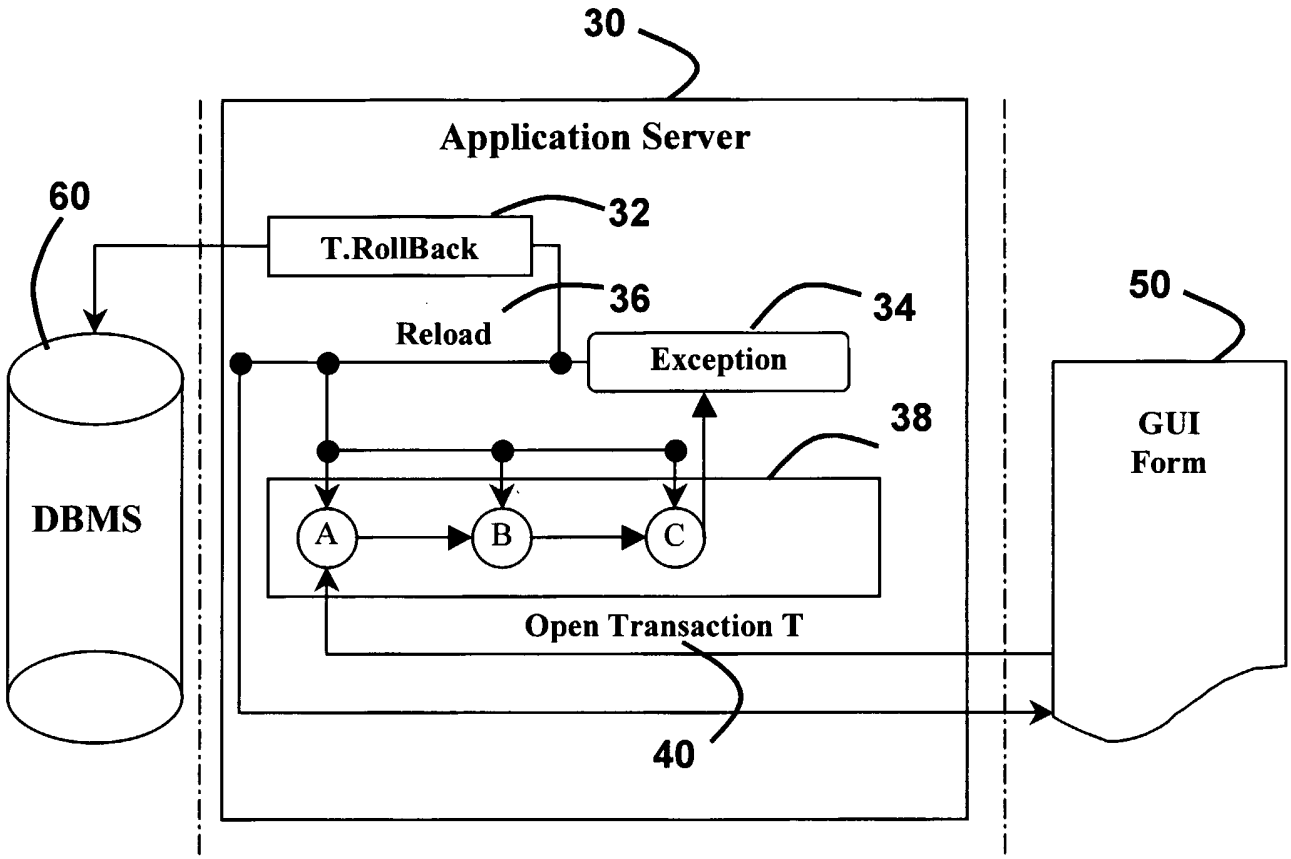

30
Application Server
60
32
T.RollBack
36
Reload
34
Exception
50
38
DBMS
A
B
C
GUI
Form
Open Transaction T
40

FIG. 2