ソフトウェア設計のための 文脈によるプログラミング言語処理系

森下 敦司†

ContextWorkbench は，文脈指向を実現するプログラミング言語処理系である．そして，本論文は，ContextWorkbench の新機能の報告である．新機能の目的は，ソフトウェア設計における試行錯誤を容易にすることである．この新機能のために，セッションと呼ばれる機構を実装した．この機構を使用すれば，異なる複数の案を並行して設計出来る．

Context-oriented Programming Environment for Software Design

Atsushi Morishita†

ContextWorkbench is a programming environment for context-oriented approach. And this paper is a report about new features of ContextWorkbench. The purpose of the features is to simplify complicated operations for trial and error in software design. The mechanism called “session” was implemented for the features. By using the mechanism, several different ideas can be designed in parallel.

1. はじめに

ContextWorkbench は筆者が開発した文脈によるプログラミング言語処理系である [1]．文脈によるプログラミングとは，制御構造やデータ構造の記述と，それらを構成する関数や変数の実装を切り離し，その結合を実行時に行うことで，プログラムの実行内容を動的に決定，変更するプログラミングである．そして，ContextWorkbench は，この仕組みを文脈機構として提供する点に特徴がある．

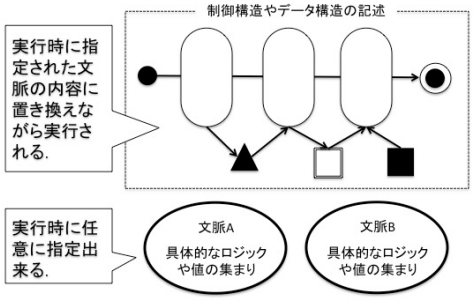

図 1 文脈

†株式会社 NTT データ
NTTDATA CORPORATION

本稿では，ソフトウェア設計の試行錯誤や分担作業を実施可能にするために実施した，ContextWorkbench の拡張について報告する．

2. ソフトウェア設計に関する考察

2.1 モデルを活用した開発の理想

ソフトウェア開発における上流工程では，情報の構造と値の変化の観点で対象領域の整理，把握を行う．この作業はモデリングと呼ばれ，その作業結果はモデルと呼ばれる．なお，モデルを中心とする代表的な開発手法はオブジェクト指向による開発である．オブジェクト指向による開発では，オブジェクト指向プログラミング言語を使用することで，上流工程で作成したモデルをプログラムの構造に適用する．これにより，オブジェクト指向を採用することで，シームレスな開発が可能になると言われている．

2.2 モデルを活用した開発の実際

しばしば，オブジェクト指向による開発は，理想通りの成果をもたらさない．筆者は，その原因の１つは，上流工程のモデリングがプログラミングと類似した素養を必要とすることへの理解不足であると考えている．なお，オブジェクト指向は，そもそもプログラミン

グ技術から生じたものであり，オブジェクトであっても，データの操作という点で，本質的には，計算機におけるメモリ操作と同様の論理性を必要とする．つまり，データのスキーマが計算機のメモリであるか，バーチャルなオブジェクトであるかの違いにすぎない．しかも，上流工程のモデリングでは，プログラミングのように動作結果として作業結果を確認することが出来ない．従って，欠落や矛盾を含んだモデルが作成されてしまう．

また，速度性能などの観点で，上流工程で作成したモデルを設計に適用出来ない場合も多い．この様な場合，構造を変換する必要が生じるが，それは，意味の誤解や情報の欠損等のミスを招く．また，上流工程でモデルを作成することの動機を損なう原因にもなる．

これらは，筆者が ContextWorkbench を開発した主要な動機である．ContextWorkbench は，実行可能な要件や仕様の記述を目指すとともに，文脈機構によって，異なる構造に対して共通に適用可能な記述を可能にすることを目指している．

3. 課題

3.1 上流工程におけるプログラミングに必要な機能

上流工程のプログラミングは，仕様を実現するのではなく，不明確なアイデアやイメージを形にして仕様を明確にするために行うものである．従って，筆者は上流工程におけるプログラミングには，次の特徴が必要であると考える．

① 対話性
② 記述の容易性
③ 後付けによる意味の明確化

そして，これらは，ContextWorkbench において以下の機能として実装されている．

① Smalltalk を模したユーザインタフェース
② 日本語シンボルを用いた記述
③ 実行時に動的に型評価を行う機能
④ 動的な意味の指定を可能にする文脈機構

なお，対話性は①，記述の容易性は②および③，後付けによる意味の明確化は④によって，それぞれ実現されている．

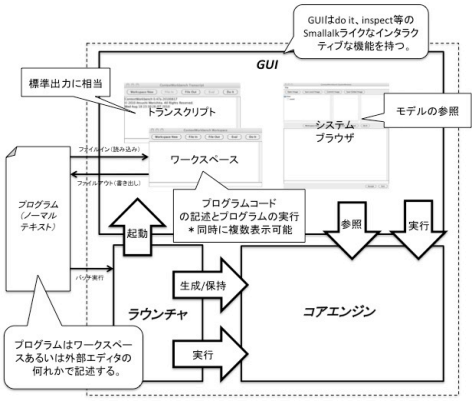

図 2 Smalltalk を模したユーザインタフェースと全体アーキテクチャ

3.2 試行錯誤を支援する機能の必要性

筆者は，上流工程におけるプログラミングについて，前節の内容に加えて，試行錯誤を支援する機能が必要であると考える．なお，ContextWorkbench では文脈機構を用いることで，その目的を達することが出来る．しかし，複数の案を検討する際に，それぞれの案に固有の文脈が混在すると作業が混乱する可能性がある．また，不要な案を削除する際に，誤って残すべき案を削除してしまう可能性もある．筆者は，これらの問題を解決し，試行錯誤の容易性を高めるには，次の要件を満たす必要があると考える．

① 複数の案を並行して作成出来るように，案毎に区画を設けて文脈を隔離する．また，ユーザインタフェースにおいて案毎に可視を制御する．
② 案毎のベースラインを統一するために，案毎に文脈を隔離するだけでなく，異なる案の間で共通の文脈を扱えるようにする．
③ 特定の案を採用するために，案毎に隔離された文脈を共通の文脈に反映出来るようにする．
④ 複数の案に共通の文脈に対する更新の競合を解決する．

なお，これらの要件は，既存の構成管理ツールやテキスト処理ツールを組み合わせるこ

とでも機能的には実現出来るが，構成管理ツールへの投入はベースラインを対象とすべきであり，試行錯誤中のプログラムを対象とすべきで無いと考える．また，試行錯誤のためにプログラムコードの差分検出や結合処理を細かく制御することは，煩雑に過ぎると考える．

4. 課題の解決

4.1 セッション機構

ContextWorkbench における文脈は，唯一のルートを起点とする木構造を構成する．なお，記号の探索は，対象とされている文脈からルートの文脈へ向かって行われる．そして，最初に発見されたものが，探索の結果となり，ルートへ至っても発見されなかった場合，その記号はリテラルと見なされる．

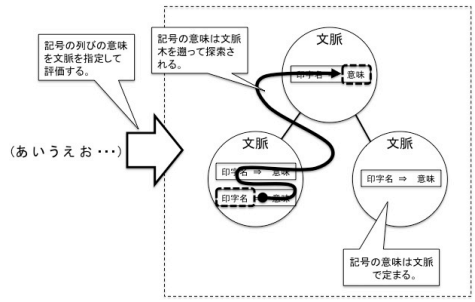

図 3 記号の意味の探索

筆者は，目的の実現のために探索機能の拡張を行った．この拡張において導入された機構をセッションと呼ぶ．セッションは文脈木への修正を保持するオブジェクトである．複数のセッションを作成することによって，１つの文脈木に対する複数の修正案を同時に作成することが可能になる．

なお，ContextWorkbench のプログラムは，必ず何れか１つのセッションを対象として実行される．なお，ContextWorkbench は常に１つのデフォルトのセッションを持ち，起動時は，そのデフォルトのセッションがプログラムの実行対象とされている．ただし，ユーザは任意にセッションを追加し，プログラムが実行対象とするセッションを変更することが出来る．ユーザは，セッション毎に別々のシステムブラウザを用いて，そのセッションで利用可能な文脈木の参照や操作を行う．ContextWorkbench のシステムブラウザの機能は，Smalltalk のシステムブラウザと同等であり，文脈木と文脈に固有の記号の意味を参照可能にする．ただし，ContextWorkbench の場合，ユーザは，同時にセッション毎のシステムブラウザを表示することで，セッション間の違いを比較しながら検討を進めることが出来る．

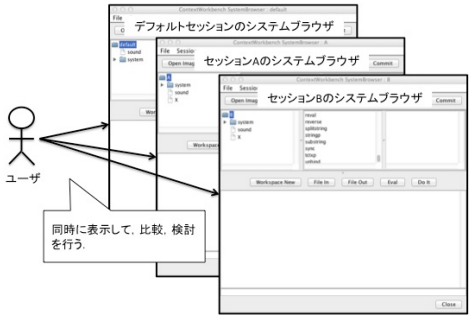

図 4 システムブラウザ

また，セッションの管理を行うために，新たにセッションブラウザを実装した．セッションブラウザは，全てのシステムブラウザから起動することが出来る．セッションブラウザを用いることで，ユーザはセッションの追加や削除を行うことが出来る．ただし，プログラムからも，それらを行うことが出来る．

図 5 セッションブラウザ

セッションの生成や指定を行うプログラムの例を次に示す.

```
00: do ( defsession 'A )
01: do ( session=A bind 'x 1 )
02: do ( defsession 'B )
03: do ( session=B bind 'x 100 )
04: do ( session=A printcr x )
=> 1
05: chgsession B
06: do ( printcr x )
=> 100
```

defsession は指定された名前のセッションを生成する関数である．従って，この例では，A と B の２つのセッションを生成している．そして，セッション A では x の値を 1，セッション B では x の値を 100 に設定し，printcr 関数によって出力している. この時, session=A のように括弧内でセッションを指定する方法と，chgsession B のように，括弧の外でセッションを指定する方法がある．この点について，筆者は後者の方が汎用性の高いプログラムを実現出来ると考えているが，プログラムの始まりでは無く，プログラムの内部で細かく実行対象のセッションを切り替える必要性を考慮して前者も実装した．なお，chgsession と do は, インタプリタに対するコマンドであり，do コマンドに続く括弧とその内部が実際のプログラムである．つまり，chgsession コマンドによって，プログラムを変更すること無しに，インタプリタに実行対象とするセッションの変更を指示することが出来る.

文脈木に対して修正を行う際，修正対象となった文脈の子を保持するオブジェクトが実行対象のセッションの中に作られる．このオブジェクトをセッション固有文脈と呼ぶ．一方，セッション間で共有される文脈を大域文脈と呼ぶ．大域文脈は，ContextWorkbench の起動時に存在していた文脈をベースとして，セッション固有文脈を反映することによって修正される.

大域文脈に対する実際の修正は，セッション固有文脈に対して行われる．つまり，修正の対象であった大域文脈自体は修正されない．ただし，その様な大域文脈には，セッション固有文脈の印が付けられる．この仕組みによって，記号の意味の探索において，探索対象の大域文脈にセッション固有文脈の印が見つかると，探索対象がセッション固有文脈に変更される． なお, セッション固有文脈は，修正対象となった大域文脈の代わりの文脈か，そのセッションに固有に作成された文脈の何れかである.

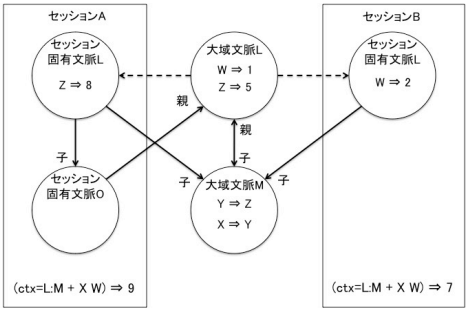

図 6　文脈木の拡張

4.2　コミット機能

セッション固有文脈を大域文脈に反映する機能をコミット機能と呼ぶ．特定のセッションに対してコミットを行うと，セッション固有文脈が大域文脈に反映され，コミットを行ったセッションのセッション固有文脈は削除される.

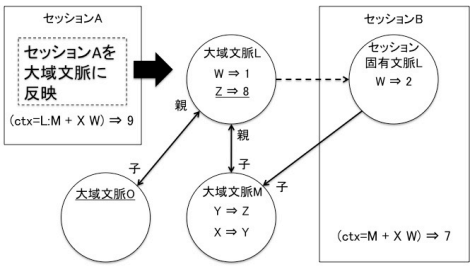

図 7　コミット機能

コミットは，システムブラウザあるいはプログラムから行うことが出来る．コミットを行うプログラムの例を次に示す.

```
01: chgsession B
02: do ( commit )
```

この例では，セッション B をコミットしてい

る．また，対象とするセッションの指定は，括弧の内部に session=B と記述することでも可能である．

なお，複数のセッションに対して繰り返しコミットが行われた場合，あるセッションによる大域文脈の更新が，別なセッションによる更新によって予期せず上書きされてしまう可能性がある．この問題を回避するために，カウンタによる簡易な制御を行った．具体的には，大域文脈にカウンタを設けて，セッション固有文脈の生成時にカウンタを加算するとともに，セッション固有文脈に、その値を保持するようにした. そして, コミット時に，セッション固有文脈が保持する値と大域文脈のカウンタの値を比較し，大域文脈のカウンタの方が大きい場合はコミット出来ない様にした．

4.3 シンク機能

カウンタによるコミットの制御では，本当にコミットを行いたい場合にも，コミット出来なくなる可能性がある．この問題を解決するために，シンク機能を実装した．シンク機能とは，コミット機能とは逆に，最新の大域文脈の内容を特定のセッションの内容に反映する機能である．ただし，修正対象となった大域文脈の代わりのセッション固有文脈に関しては，セッション固有文脈の内容が優先される．一方，そのセッションで修正を行っていない大域文脈については他のセッションをコミットした内容が反映される．

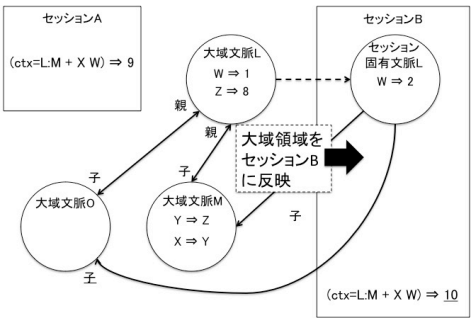

図 8 シンク機能

シンクは，システムブラウザあるいはプログラムから行うことが出来る．シンクを行うプログラムの例を次に示す．

```
01: chgsession B
02: do ( sync )
```

この例では, セッション B をシンクしている．また，対象とするセッションの指定は，括弧の内部に session=B と記述することでも可能である．

なお、シンクを行うと大域文脈のカウンタが更新され，セッション固有文脈にカウンタの値が反映されるため，シンクを行ったセッションはコミットが可能な状態になる．従って，シンク機能とコミット機能を連続して実行することでセッション固有文脈を大域文脈に反映することが可能になる．

4.4 ファイル入出力機能

コミット機能とシンク機能を用いて複数人で分担してプログラミングを行うために，セッションの単位でファイル入出力を行う機能を試作した． この機能を用いることで，共通の大域文脈を利用しつつ，分担してプログラミングを行うことが出来る．また，それぞれの作業結果をファイルに出力し，それを集めて新たな大域文脈を作成することが出来る．

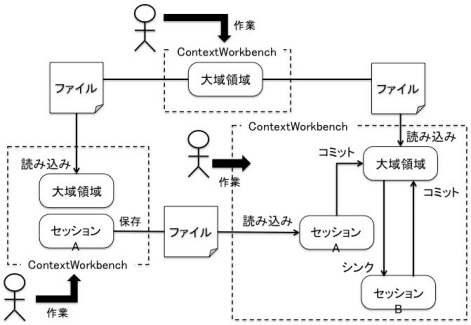

図 9 ファイル入出力機能

なお，この機能の実装には Java に固有のシリアライズ機能を利用した．従って，セッション固有文脈のみを出力するために，セッション内の大域文脈に対する参照を文脈木におけるパスに置き換える処理を必要とした．また，ファイルの入力にあたっては，逆に，文脈木におけるパスから大域文脈のオブジェクトに置き換える処理を必要とした．

5. まとめ

ContextWorkbench にセッションの機構を実装することで，次の通りに，複数の案を試行錯誤しながらプログラミング出来ることを確認出来た．

① セッション機構によって，複数の案をセッションとして並行に作成出来る．また，セッション毎にセッション固有文脈として文脈が隔離される．なお，セッション毎に別なシステムブラウザを用いることで，可視が制御される．
② セッション機構によって，大域文脈として複数の案に共通な文脈を扱える．
③ セッション機構のコミット機能によって，セッション固有文脈を大域文脈に反映出来る．また，ファイル入出力機能によって，異なる計算機上の ContextWorkbench で分担して行われた作業結果を取り込める．
④ セッション機構のシンク機能によって，大域文脈に対する更新の競合を解決出来る．

ただし，ファイル入出力機能については，Java に固有のシリアライズではなく，XML 形式の独自のシリアライズに変更する方針である．Java に固有のシリアライズを用いた場合，ContextWorkbench の内部実装の変更によってファイルの互換性が損なわれる可能性が有る．

なお，今後の方針としては，セッション機構の特徴を活かして，大域文脈に対する権限によるアクセス制御など，アプリケーションサーバやデータベースとしての利用を目的とした機能拡張を重点的に行う予定である．

参考文献

1) 森下敦司,“文脈によるプログラミング言語処理系”, 情報処理学会夏のプログラミング・シンポジウム報告集, 2011, Vol. 2010, pp. 111-116

質疑応答

Q コミットやシンクを行った結果を元に戻す（ロールバックする）ことは出来るのか？
A 現状では，その様な機能は有りません．ただし，データベースとしての利用を考慮するならば実装する価値はあると考えています．

Q シンクはどの様な単位で行われるのか？また，コミットした結果が通知無しに全てのセッションに反映されるのは危険ではないか？
A シンクの単位は文脈です．コミットの件については，ご指摘の通りかもしれません．ただし，オブジェクト指向DBMS などには，その様な動作をするものも有ります．また，セッション間の共通箇所の設計指針にも関係すると考えます．

Q 構成管理の観点で，大規模なソフトウェアの開発においては，関数の内部など，より細かい単位で制御出来る必要が有るのではないか？
A 経験上，少なくとも事務処理システムにおいては，プログラムコードの差分の検出や結合は関数よりも細かい単位では，あまり行われていないと考えています．なお，文脈の仕組みを用いることで，関数単位で実装が異なる文脈のバリエーションを扱うことは可能です．