人工知能学会
第12回知識流通ネットワーク研究会
SIG-KSN-012-01

知識伝達媒体としてのライブプログラミング環境

小田 朋宏[1]　中小路 久美代[1]　山本 恭裕[2]

[1]株式会社 SRA 先端技術研究所
[2]東京工業大学 精密工学研究所

Live Programming Environments as Media for Knowledge Transfer

Tomohiro Oda[1], Kumiyo Nakakoji[1], and Yasuhiro Yamamoto[2]

[1] Key Technology Laboratory, Software Research Associates, Inc.
[2] Precision and Intelligence Laboratory,　Tokyo Institute of Technology

概要: ソフトウェア開発において複数人の開発者が即興的な体験を共有することで知識の伝達および共有が起こる代表的な例として，アジャイル開発で行われるペアプログラミング，プロトタイピング，ホワイトボードを使った議論に着目し，アイデアの発案と評価を行うためのライブプログラミング環境 SOMETHINGit を紹介する．

Abstract: Transfer and sharing of knowledge happen when sharing live performance of the task at hand. This paper describes pair programming, prototyping and whiteboard discussions as shared live performance for knowledge transfer and sharing, and introduces SOMETHINGit, a live programming environment.

1. はじめに

ソフトウェア開発では，開発における知識が一部の構成員に偏らず，共有されることが必要である．そのために多くの開発組織では，記述すべき開発文書を定め，開発に必要となる知識が共有されるように努めている．一方で開発においてはプログラミングでの暗黙知/経験知も存在している．ソフトウェア開発における暗黙知は，ソフトウェア開発の経験を共有することで伝達され共有される．

本稿では，ソフトウェア開発において即興的な体験を共有することで暗黙知を伝達している手法を概観し，ライブプログラミングによる知識共有を図るための開発環境 SOMETHINGit を紹介する．

2. ソフトウェア開発における即興的な実演による知識伝達

開発文書は外在化された形式知であり，一方で開発においてはプログラミングでの暗黙知/経験知も存在している．暗黙知/経験知は文書化できないため，その伝達のために導入教育やメンタリングなどが行われる．暗黙知は体験を共有することで伝達が可能になる．例えば，開発者コミュニティ参加のためにコミュニティの周辺から参加し，コミュニティとしての活動を既存メンバーと共に体験することで，その開発者コミュニティにおける暗黙知が伝達される．

本節では，ソフトウエア開発において現在用いられている知識伝達のための手法のうち，即興的な実演としての側面を持つものを挙げる．

2.1. ペアプログラミング

ペアプログラミング[1]は，2 人 1 組のプログラマが 1 台の計算機を共有し，協調してプログラミング作業を遂行する手法である．2 人のプログラマは同一のプログラムについて相補的な作業を行う．ペアプログラミングを提唱した Beck は，1 人のプログラマ（ドライバ）がキーボードとマウスを取ってプログラムを実装している時に，もう 1 人のプログラマ(ナビゲータ)はその作業をより戦略的に俯瞰する，としている．ドライバが置かれている状況をナビゲータが観察し共有することで，ナビゲータに暗黙知/経験知が伝達される．また，ナビゲータからの質問や指示を受けることで，ドライバはナビゲータが意思決定を下した状況を共有する．

ペアプログラミングにおける知識伝達は既に多くの研究が報告されている．Fronza ら[2]は，開発チームに新しいメンバーが加入する過程でのペアプログラミングについて，Initiation, Independence, Maturity, Integration の 4 つのフェーズを同定し，特に Initiation フェーズにおいて既存メンバーと新メンバーのペアプログラミングが多用され，Integration フェーズで

は新メンバーは既存メンバーと差異無くペアプログラミングを行う点を指摘した．開発チームにおける知識が Initiation フェーズでペアプログラミングを通じて新メンバーに伝達され，Integration フェーズにおいて新メンバーはペアプログラミングを通じて既存メンバーと一体となり，知識を共有する．

ペアプログラミングが一人で行うプログラミング(ソロプログラミング)と異なる特徴として，即興性が挙げられる．ソロプログラミングでは，プログラマは作業結果をある程度のまとまりにしてチームに提供する．ペアプログラミングでは，作業中のプログラムがナビゲータから見られる状態が継続する．構文検査や型検査等に合格する品質に至る前のプログラムが共有され，ドライバは常にナビゲータに対して即興的なプログラミングを実演することになる．

2.2. プロトタイピング

プログラマという同種の技術者同士がペアを組むペアプログラミングだけでなく，プロトタイプを中心に置いて異なる専門性を持った開発者/利害関係者が議論をすることも，プログラミングの経験知を共有するものと考えることができる．

アーキテクトと顧客の間で，プロトタイプを使って要求を引き出すことがある[3]．プロトタイプはしばしば実行可能な形式を持ち，アーキテクトは顧客が求める品質に関してアーキテクチャがどのようにその品質を実現するのかを説明する．例えば，顧客がハードウェア障害に対する可用性の要求があり，その実現のために冗長性の高いアーキテクチャを適用する場合を考える．顧客が特定のコンポーネントを指定し，そのコンポーネントにハードウェア故障が起こった時の動作をアーキテクトが説明する．要望に従って開発者がプロトタイプを操作し，どのような動作が起こるかを確認し，開発者がその動作を説明し，顧客がその説明を受けて次の要望を提示する，という事が行われる．

顧客の要望に対してアーキテクチャの特性を説明するためにプロトタイプを即興的に操作する必要がある．また，アーキテクトは必要に応じて顧客の前でプロトタイプに小さな修正を施してプロトタイプを再構築し，顧客が要求する操作を実演することがある．これらの点から，プロトタイプによる説明は即興性が要求される．

2.3. ホワイトボード

開発者同士がホワイトボードを使って議論する時，ホワイトボードを通じて即興的な知識伝達が発生する．ホワイトボード上の記述そのものは外在化された形式知であるが，ホワイトボードという媒体を共有して行われる記述行為が経験として共有されることで知識の伝達が行われる．

中小路ら[4]はホワイトボードによる開発者同士の議論をビデオ撮影し特徴を抽出した．うち，特に即興性に関連するものを挙げる．

- 与えられた仕様にはない，新しく出てきた概念を扱う
- 共通理解を構築するための一時的な描画をおこなう
- 多様な記法を用いる
- UI を図的/言語的に詳細化する

表 1: ソフトウェア開発における 3 種類の即興的な実演

	ペアプログラミング	プロトタイピング	ホワイトボード
目的	製品の一部を完成させる	発案する，検討する	発案する，検討する
記述対象	製品プログラム	暫定的プログラム	暫定的表現
記述言語	プログラミング言語	プログラミング言語	フリースケッチ
成果物への厳密性	構文検査，型検査，単体テスト，コード規約	構文検査，型検査	仕様する記号や色等に関する規約
時間的制約	セッション中	要求された時点	アイデアが出た時点
決定事項	実装	疑問，要件，仕様	疑問，要件，仕様
実演事項	プログラムを記述する過程	プログラムが動作する過程	意思を決定する過程
失敗の深刻度	セッション中に修正	実演の失敗が不信を招く	セッション中に修正

- 「エウレカ」の瞬間を認識する
- 話すことと描くことの間に時間のずれが生じる

3. 知識伝達媒体としてのライブプログラミング環境に求められる特性

ライブプログラミング[5]は，アルゴリズムおよびその表現としてのプログラムを即興的に作成し操作する実演である．特に芸術的な実演として音楽や映像を生成するプログラムをその場で作成し実演することはライブコーディング[6]とも呼ばれる．ライブコーディングに用いられるプログラミング言語はScratch[10]等のグラフィカルなプログラミング言語が用いられることも，通常のテキスト形式のプログラミング言語が用いられることもある．

芸術的パフォーマンスとしてのライブコーディング以外にも，プログラミング教育において教師が生徒に対してプログラミングの実演を行う等，ライブプログラミングは芸術以外の領域においても行われている．ソフトウェア開発における即興的実演もライブプログラミングとしての要素を持つ．

前節で挙げた 3 種類の即興的な実演による知識伝達は，製品の実装を目的とするペアプログラミングと，アイデアの創出と評価を目的とするプロトタイピングおよびホワイトボードとで特性が大きく異なる．後者では特に，共同作業の中で発生したアイデアを即座に記述しそのアイデアに対する評価を行うことが重要である．即興的な表現は最初から最後まで同一の詳細度や厳密度で記述されるのではなく，同じ対象に対して段階的に詳細度や厳密度が変化していく. 例えばホワイトボードで UI について議論する時には，ラフなスケッチから始まり，別の議論を経てからスケッチに戻り詳細化していくことが観察されている．

従って，アイデアの創出と評価を目的とした即興的な実演を行う環境には，

- アイデアを表現するのに適したメディア
- 他の要素との関係に応じた適切な詳細度/厳密度

を幅広く提供することが求められると考えられる．

4. SOMETHINGit

本説では，アイデアの創出と評価を目的として試作したライブプログラミング環境 SOMETHINGit について述べる． SOMETHINGit は表現形態としてグラフィカルな表現とテキスト表現の両方を扱うことができ，それぞれの表現形態について曖昧で柔軟な記述と詳細で厳密な記述を選択することができる．

4.1. 3 つの言語

上に挙げた選択を可能にするために，SOMETHINGit では 2 種類のプログラミング言語および 1 種類の形式的仕様記述言語，1 種類のスケッチツールを備えることとした．以下に 3 種類の言語の特性を表 2 にまとめる．

表 2: 3 種類の言語の特性

言語名	Smalltalk	Haskell	VDM-SL
環境	動的環境	コンパイラ インタプリタ	インタプリタ
パラダイム	オブジェクト指向プログラミング	関数型プログラミング	形式的仕様記述
型	動的型付	静的型付	静的型付
目標	Dynamism	Purity	Rigor

Smalltalk は、ユーザによる対話的プログラミングのために開発された言語であり，プロトタイピングから製品プログラムまで様々な適用がされており，特に柔軟性の高さに特徴がある．Smalltalk は動的環境を基盤としており，強力なメタ機構およびリフレクション機構を持っている．

Smalltalk では主にワークスペース，クラスブラウザ，インスペクタ，デバッガの 4 つのツールによって開発を行う．ワークスペースでは，任意のテキストを扱うことができ，また，プログラム片を選択し実行することができる．すなわち，1 つのまとまりのテキスト表現の中に，自然言語としての部分とプログラムとしての部分を混在させることができる．クラスブラウザ，インスペクタ，デバッガはワークスペースの機能に加え，プログラムの実行中にもクラスやメソッドの定義を変更をし，元のプログラムを実行中にプログラムをそのまま新しい定義に従って実行を継続させることができる．また，デバッガは強力なステップ実行の機能を持っている．Smalltalk 開発者は上記 4 つのツールを使って，プログラムを実行しながら修正を施していくことで，即興的なプログラミングを高い自由度で行うことができる．伝統的な edit-compile-run というサイクルから解放され，実行中のプログラムのオブジェクトを補足し，そのプログラムそのものを変更しながら操作し，実行を継続していくことができる．

また，Smalltalk は開発された当時は GUI の先駆者

であり，柔軟な GUI の構築や画像の扱いに優れている．従って，UI に関連したプロトタイピングに向いていると言える．

Haskell[7]は純粋関数型プログラミング言語であり，強力な型推論を備えた静的型付け言語である．非常に抽象度が高い記述が可能であり，かつ型検査および参照の透明性により，品質の高いプログラムを開発することができる．処理系としてコンパイラとインタプリタが実装されており，開発環境から外部プログラムとして使うことができる．

VDM-SL[8]は形式的仕様記述言語であり，プログラムの各コンポーネントについてのデザインディシジョンを表明として記述することができる．また，VDM-SL はコンポーネントを表明によって暗黙的に定義する（陰仕様）ことができると同時に，コンポーネントの明示的記述を与える（陽仕様）ことで，プログラムとして実行することができる．プログラムの機能に関するプロトタイピングに向いているが，GUI 等の UI を直接記述する事には向いていない．

VDM-SL はインタプリタが実装されており，開発環境から外部プログラムとして使うことができる．

これら 3 つの言語はアイデア創出と評価を目的としたライブプログラミング環境 SOMETHINGit において表 3 に示す役割分担をする．

表 3: 3 種類の言語の役割分担

	Smalltalk	Haskell	VDM-SL
UI のラフスケッチ	スケッチツール		
UI の詳細な構成	UI フレームワーク GUI 部品 UI 機能の実装	UI 機能の実装	UI 機能の仕様
テキストのラフスケッチ	ワークスペース	型による概念整理	表明 型による概念整理
テキストの詳細な構築	クラス，メソッド	型，関数	仕様記述

4.2. 5 つのレベルによる結合

SOMETHINGit は下記の 5 つのレベルにより Smalltalk と Haskell および VDM-SL を結合させている．複数レベルでの結合をすることで，1 つのプログラムについて 1 つの言語を選択したり，あるいは，1 つのプログラムを特性に応じて 2 つもしくは 3 つに分割してそれぞれを 1 つの言語で記述したりするのではなく，3 つの言語による記述を混合して用いることを可能にした．

Level 1: ツール

ワークスペースはホワイトボードで行われるような，浮かび上がったアイデアを書き留めるためのスケッチ的なテキスト表現に用いられる Smalltalk 環境での開発ツールである．テキスト表現中の一部を指定して，その表現を Smalltalk プログラム片として実行する機能を持っている．以下にワークスペース上の Smalltalk プログラム片を実行した結果を示す．以後，本稿では記述片の実行について，「➡」記号の前に実行するプログラム片，側にその結果出力を示す．

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each] ➡ 55
```

SOMETHINGit はワークスペースに“Haskell It”機能を追加する．これによってユーザは，ワークスペース上にフリーテキスト，Smalltalk プログラム片，Haskell プログラム片を混在させる形でアイデアを記述することができる．以下にワークスペース上の Haskell プログラム片を実行した結果を示す．

```
take 2 [4, 4, 1] ➡ [4, 4]
```

ブラウザは，ワークスペースで扱うよりもまとまった記述を整理し記録･閲覧するために用いられる．また，インスペクタはあるオブジェクトに着目し，そのオブジェクトに対するインターフェイスとしてのワークスペース機能を提供する．SOMETHINGit は VDM-SL での記述を支援するツールとして，VDM Browser を提供する．VDM Browser は，インスペクタとブラウザの機能を併せ持つツールで，ブラウザとしてひとまとまりの VDM-SL 仕様を記述する場を提供すると同時に，その仕様に関する状態の閲覧/更新や記述片の実行を行うインスペクタとしての機能も提供する．以下に，VDM-SL の記述片とその実行結果を示す．

```
[1,2,3] ^ [4,5] ➡ [1,2,3,4,5]
```

ユーザは，これらのツールを Smalltalk 環境上で利用することで，1 つの開発環境の中で複数の言語記述を併用することができる．

Level 2: eval 関数

SOMETHINGit は Smalltalk プログラム中で Haskell

およびVDM-SLの式を実行する機能を提供している．eval 関数自体はプログラム片の評価を行うための基本的な機能であるが，Level1 でのツールによる結合において複数言語により記述された記述片を簡単に１つにまとめて実行することを可能にする．

以下に，Smalltalk プログラムとして実行した結果を示す．

```
SIHaskell evaluate: ‘take 2 [4, 4, 1]’
➡ anOrderedCollection (4 4 )
SIVDM evaluate: ‘[1,2,3] ^ [4,5]’
➡ anOrderedCollection (1 2 3 4 5 )
```

実行結果はHaskellおよびVDM-SLの式ではなく，Smalltalk オブジェクトとして返ってくる．これは後述の Object-Value Mapper により変換される．

Level 3: クロージャ

SOMETHINGitは，Haskell および VDM-SL での機能記述単位である関数および操作をSmalltalkのクロージャオブジェクトと互換なオブジェクトとして取り出す機能を提供している．クロージャ単位での結合をすることで，２つの言語による実装の混合を細粒度で行うことができる．

Smalltalk の UI フレームワークでは，機能をクロージャ単位で組み合わせて適用することができる．異なる言語の機能単位をクロージャレベルでSmalltalk プログラムに組み込むことを可能にすることで，他の言語による記述であることを強く意識することなく，あたかもSmalltalk で実装された機能単位であるかのようにUIを構築することができる．

また，VDM-SL 仕様が提供する機能単位をSmalltalk で UI を構築する時に，モジュールのような大きな単位での実装を行い入れ替えるのではなく，VDM-SL での最小の外部機能単位である関数や操作ごとにSmalltalk で実装し，VDM-SL による定義と入れ替えていくことができる．Smalltalk で記述されたプロトタイプを Haskell での実装に置き換えていく際にも同様である．Haskell プログラム中で定義されている関数をSmalltalkのクロージャとして利用する例を以下に示す．

```
| hProgram take2Closure |
hProgram:= SIHaskell source: ‘take2 = take 2’.
take2Closure := ‘take2’ asHaskellExpressionIn: hProgram.
take2Closure value:#(4 4 1) asOrderedCollection
➡ anOrderedCollection (4 4 )
```

上記の例において，2 行目の Haskell プログラム’take2 = take 2’は関数 take2 を定義している．その Haskell プログラムを Smalltalk オブジェクト hProgram としている．3 行目では，hProgram が定義する Haskell 関数 take2 を Smalltalk のクロージャ take2Closure としている．4 行目で take2Closure クロージャに引数#(4 4 1) asOrderedCollection を渡して実行している．後述の Object-Value Mapper によって引数は Haskell でのリスト[4, 4, 1]に変換され，take2 [4,4,1]が実行され，Haskell での実行結果は[4, 4]となる．これが再び Object-Value Mapper により Smalltalk オブジェクトに変換され，結果として anOrderedCollection (4 4)となる．

同様に，VDM-SL 仕様で定義されている関数および操作をSmalltalkのクロージャとして利用することができる．

```
| vdmSpec cat45Closure |
vdmSpec := SIVDM specification: 'functions cat45:seq of nat -> seq of nat cat45(xs) == xs ^ [4, 5]'.
cat45Closure := 'cat45' asVDMFunctionIn: vdmSpec.
cat45Closure value:#(1 2 3) asOrderedCollection
➡ an OrderedCollection(1 2 3 4 5)
```

Level 4: Object-Value Mapper

SOMETHINGit は Haskell および VDM-SL の値とSmalltalk オブジェクトの相互変換を提供している．相互変換はユーザ拡張が可能である．表 4 に標準状態での変換表を示す．

表 4: Object-Value Mapper 変換表

言語	Smalltalk	Haskell	VDM-SL
リスト	OrderedCollection	list	seq
集合	Set	Data.Set	set
マップ	Dictionary	Data.Map	map
レコード型	SIAlgebraicData	*	record, tuple, token
整数	Integer	Int	int
小数	Float	Fraction	real
文字	Character	Char	char
文字列	String	String	seq of char
Smalltalk Object	Object	SIObject	-

Level 5: コールバック

SOMETHINGit は Haskell 関数として，Smalltalk オブジェクトへのメッセージ送信を行う関数 messageSend を提供する．Haskell 関数からのコールバック機能を提供することで，Smalltalk によるプロトタイプから Haskell による実装に置き換えていく際に，Haskell での再実装から利用する機能の一部に Smalltalk 実装を残したまま，Haskell での再実装を進めていくことができる．

例えば，以下の Smalltalk プログラム片では，Haskell の list 型の値に対応する Smalltalk オブジェクトに対して at:メッセージを送信している．

```
| ghci at |
ghci := SIHaskell prelude.
ghci program: 'let at array index = messageSend array "at:" [index] :: IO Int'.
at := 'at' asHaskellExpressionTyped: '[Int]->Int->IO Int' in: ghci.
at value: #(4 4 1) value: 3
➡ 1
```

Smalltalk オブジェクトへのメッセージ送信は IO を介して行われるため，関数 messageSend の返り値の型は IO モナドのインスタンスである必要がある．

5. 関連研究

Squeak etoys [9] および Scratch[10]は子供達がビジュアルにプログラミングするための環境である．タートルグラフィックスや音響などの動作の記述を通して，子供達がプログラミングの専門知識なしにその場で試行錯誤をして作品を作り上げていく．また，スケッチツールによって画像による表現をすることが可能である．SOMETHINGit は Squeak etoy 向けに開発された Smalltalk 環境とそのライブラリの上に構築されている．Squeak etoys や Scratch が試行錯誤による個人の学習を目的にしているのに対し，SOMETHINGit は即興的な実演を通した知識の伝達および共有を目的にしている．

Lively Walk-Through は，SOMETHINGit 上に開発された UI プロトタイプ環境である．機能仕様策定者と UI デザイナがそれぞれ作成した機能仕様と UI スケッチ画を組み合わせて１つの UI プロトタイプを構成し，開発対象となるシステムに対するそれぞれの理解を説明し，議論し，合意事項を構築することで，「作るモノの世界」と「使うコトの世界」を摺り合わせることを目的としている．UI プロトタイプを構成し評価する過程において SOMETHINGit による即興的な記述への支援がされる．Lively Walk-Through が機能仕様策定者と UI デザイナによる UI プロトタイピングという特定の工程を想定して設計されているのに対して，SOMETHINGit はより広くソフトウェア開発における即興的な記述を捉えることを目標としている．

参考文献

[1] Beck,K., Extreme programming explained: embrace change, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999

[2] Fronza,I., Sillitti,A., and Succi,G., An interpretation of the results of the analysis of pair programming during novices integration in a team. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09), pp. 225-235, 2009

[3] Bardram,J.E., Christensen,H.B., Hansen,K.M., Architectural prototyping: an approach for grounding architectural design and learning, in Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture(WICSA 2004), pp. 15-24, 2004

[4] Nakakoji,K. and Yamamoto,Y., Conjectures on How Designers Interact with Representations in the Early Stages of Software Design, in Software Designers in Action: A Human-Centric Look at Design Work, M. Petre, A. van der Hoek (Eds.), Chapman & Hall 2013 (in print)

[5] McDirmid,S., Living it up with a live programming language, In Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07), pp.623-638, 2007

[6] Collins,N. , McLean,A. , Rohrhuber,J. and Ward,A., Live coding in laptop performance, Organised Sound, v.8 n.3, p.321-330, 2003

[7] Hudak,P., Hughes,J., Jones,P.J. and Wadler,P., A history of Haskell: being lazy with class. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III), p. 12-1-12-55, 2007

[8] Fitzgerald,J. and Larsen,P.G., Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press, 1998

[9] Kay, A. Squeak etoys, children, and learning, http://www.squeakland.org/resources/articles.

[10] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y Kafai, Scratch: programming for all. Commun. ACM 52, 11 (November 2009), pp. 60-67, 2009