夏のプログラミング･シンポジウム「ビューティフル･インターフェイス」2014.8.24

実世界 Live Programming の実現に向けて

加藤淳[†1]

実世界入出力を伴うプログラムは，ディスプレイ上の文字や記号の集合として静的に記述されるが，その実行状態は実空間上で動的に変化する．この時間と空間に関するギャップが，既存の統合開発環境(IDE)での開発を困難にしている．そこで本稿では，時間のギャップを超える開発支援手法として Live Programming を紹介する．さらに，その拡張として，画像表現を利用して空間のギャップも超える手法を提案する．最後に，未来の開発環境について議論する．

Toward Live Programming in the Real World

JUN KATO[†1]

While programs that use real-world input and output (real-world I/O) have static representations as source code rendered on flat displays, their state information dynamically changes along with time in the three-dimensional real world once executed. These gaps of time and space make it difficult to develop such programs in existing integrated development environments (IDEs). This paper introduces live programming as an effort to eliminate the gap in time, followed by proposal of using integrated graphical representations for filling the gap in space. Future research direction in design of programming environments is also discussed.

1. はじめに

かつてプログラミングといえば，紙の上でアルゴリズムを練ることだった．コーディングは，そのアルゴリズムを機械語に直して，パンチカードに穴を空ける作業だった．また，コンピュータは単純な入力データを処理して同様に単純な出力値を返すプログラムを動かす自動計算機だった．その後たった半世紀の間にプログラムの開発はコンピュータのスクリーン上で完結するようになり，プログラミングとコーディングの境界は消失した．コンピュータは様々なセンサやアクチュエータと接続され，プログラムは実世界入出力をリアルタイムに処理するようになった．多くの人にとってコンピュータはもはや自動計算機でなく，インタラクティブに要求に応えてくれる変幻自在の道具となったのである．

このように，プログラムの開発環境とプログラムの用途は，共に進歩してきたと言える．しかし，両者を比べると，開発環境のほうにまだ大きな改善の余地があるように思われる．“Programmers (Software engineers) are people, too.”とは，ソフトウェア工学および Human-Computer Interaction の研究分野で繰り返し語られてきた言葉である．ソフトウェア工学の文脈では，間違いのないプログラムを作ることを求められているプログラマは間違いを犯す人間であり，間違いを防ぐような配慮が開発環境に必要である，といった意味合いとなる[1]．また，より人間的側面を強調した解釈では，ユーザのことを考えて使いやすいプログラムを作ってきたプログラマは，果たして自身の開発環境を本当に使いやすくしてきただろうか，という問いかけとなる[2]．

本稿では，実世界入出力を伴うプログラムの開発が，既存の統合開発環境(IDE)では困難である問題を論じる．この原因として，プログラムが**ディスプレイ上**の**静的表現として記述**されるのに対して，実行状態は**実空間上**で**動的に変化**するという時間と空間に関するギャップが挙げられる．次章では，実世界入出力を伴うプログラムの振る舞いがプログラマの意図に沿っているか判断するためには，この**時空間のギャップ**を解消する必要があることを議論する．次に，時間の溝を超える既存手法として Live Programming を紹介する．さらに，その拡張として，画像表現を利用して次元の

†1 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology

壁も超える手法を提案する．最後に，未来の開発環境について議論する．

なお，本稿は筆者の英語博士論文[3]を下敷きにしている．論文要旨を短くまとめたものは英語[4]と日本語[5]でそれぞれ刊行済みだが，本稿は Live Programming に関する議論を主軸に据えるため論旨展開を大幅に変更したほか，夏のプログラミングシンポジウム登壇発表での議論を反映してある．

2. 実世界入出力を伴うプログラム開発

現在主流の IDE は，ディスプレイ，キーボード，マウスという標準的な入出力デバイスを用いるプログラムを開発するために設計されてきた．このような標準的な入出力を伴うプログラムでは，入出力データの取りうる値はあらかじめ定数で定義することが容易であり，データのやり取りはキーの打鍵などの度に断続的に生じる．IDE 側でデータを分かりやすく表現するためには文字や記号で十分であり，IDE は，文字や記号ベースのユーザインタフェースを提供することでプログラム開発を効果的に支援してきた．

一方で，プログラムが処理する入出力データの種類が豊富になってきている(図 1)．マウス，キーボード，ディスプレイという従来型入出力に加え，カメラや深度センサから得られる映像情報や人の姿勢情報，アクチュエータの出力を制御するためのロボットの姿勢情報など，実世界にある情報をサンプリングして扱うプログラムが一般的になりつつある．本稿では，これを実世界入出力と呼称する．

実世界入出力を伴うプログラムでは，複雑な構造のデータが連続的にやり取りされる．しかしながら，このような情報は文字や記号では直感的に提示できない．プログラム開発は通常，プログラミング言語を用いてプログラムの静的表現を記述する**プログラミング**と，その実行状態を観察して意図に沿っているか確かめる**デバッグ**を繰り返すことで進められるが，既存の IDE では，両方のステップでユーザビリティに問題が生じてしまう．

まず，**プログラミング**においては，実世界のことがらを文字や記号といった非直感的な表現で書き下すことが求められる．こうして書き上がったソースコードは，一度書けば必ず動くわけではないため，再度読み返す必要が生じる．また，書き手ではない誰か別の人が読むこともあるだろう．この際に，文字や記号といった静的表現からプログラムの振る舞いを想像することは非常に困難である．

そして，**デバッグ**においては，プログラムの実行状態を非直感的な表現を通して理解するような苦行を強いられる．例えば，一般的なデバッグウィンドウでは文字で変数の値を確認できるが，その時間変化を追うことはできない．また，現在の実行状態がソースコードのどの部分によって引き起こされているのか，理解することも大変難しい．

このように，従来の IDE ではプログラミングとデバッグが別個の行動として設計されているため，ソースコードから振る舞いを想像したり，振る舞いの原因を探ったりすることが難しい．実世界入出力を伴うプログラム開発においては，プログラミングの対象となるソースコードがコンピュータ内の開発環境で記述され，デバッグの対象となる動的な振る舞いがコンピュータ外の実世界にある実行環境で観察されるものであることが，問題をより複雑にしているものと考えられる．したがって，この問題を解決するためには，原因であるコンピュータのディスプレイと実世界の間の**次元の壁**およびプログラミングとデバッグの**時間的な溝**を解消する必要がある．

図 1. 本稿で扱う実世界入出力を伴うプログラムの例.

3. Live Programming

既存の IDE では，プログラミングとデバッグは基本的に別個のステップとして設計されてきた．この時間的に分離されてきた二つの行為を統合的に設計し，プログラムのソースコードと実行時の振る舞いの関係をプログラマに分かりやすく提示する試みは Live Programming と呼ばれている．本節では，先述した**時間的な溝**を解消できる手法として Live Programming を取り上げ, 既存研究を概説する．

Live Programming は，プログラミング言語，ソフトウェア工学, Human-Computer Interaction (HCI)という 3 分野にまたがる学際的な研究トピックである．ソフトウェア工学の観点では，プログラムの設計が固まっておらず完成形が見えない Exploratory Programming と呼ばれる工程で活用が期待できる技術である．HCI 分野では当該工程はプロトタイピングと呼ばれ，様々なツールキット開発と並行して，開発環境のユーザインタフェースを改良する研究が続けられてきた．プログラミング言語は開発環境の一部であり，プログラミング言語研究と HCI 分野の開発環境研究は切っても切れない関係にある．

プログラミングの Liveness は，Tanimoto のビジュアルプログラミング言語 VIVA に関する研究論文[6]で初めて詳細に議論された概念である．VIVA は，プログラミングとデバッグの間で従来必要だったコンパイル操作をなくし，ビジュアルなブロックを組み合わせるとプログラムの振る舞いを動的に変更できるようにした. それまで, プログラマはソースコードという「死んだ」表現を操作し，コンパイルして実行することで「生かし」，その様子を観察して意図に沿わなければソースコードの編集に戻ることを繰り返していた. Liveness に関する議論により，生きた状態を保ったままプログラムを編集できることの重要性が初めて認識されたと言える．また，Maloney らは Morphic [7]という Graphical User Interface (GUI) の開発・実行環境において，実行中の GUI コンポーネントの振る舞いをGUI 上で(別個の開発環境を開かずに)直接編集可能な Directness という概念を提唱した．Morphic は，編集内容を動的に反映する Liveness も備えていた．これらの議論を踏まえ，Hancock の博士論文[8]は，開発中のプログラムに関する情報をプログラマへ継続的にフィードバックする開発支援手法を Live Programming と名付けた．

それ以来，McDirmid らがゲーム開発用の Live Programming 環境[9]を作るなど，関連研究が活発に行われている．詳細は筆者の博士論文 2.3.3 小節[3]を参照されたい．近年では，Victor による魅力的なデモ[10]をきっかけに Kickstarter で Light Table IDE と呼ばれる Live Programming 用の開発環境がファンドレイジングに成功し，Xcode の Swift 言語サポートに同様の機能が組み込まれるなど，商用レベルでも注目が集まっている．筆者も GUI の Live Programming 機能を実現するために文字ベースのプログラミング言語を再設計する研究[11]に従事し，開発した機能が TouchDevelop という Web 上の開発環境に組み込まれている．

これまでに紹介した Live Programming に関する既存研究は，プログラミング中にデバッグ情報を提示し，デバッグ中にプログラミングできるようにするなど，プログラミングとデバッグの**時間的な溝**を解消するものが多数を占めている. しかしながら, 実世界入出力を伴うプログラムの開発において，紹介した手法を直接適用できるケースは稀である．なぜなら, これらの手法ではディスプレイ, キーボード，マウスという標準的な入出力デバイスを用いるプログラムを開発することを暗黙の了解にしており，先述した**二次元と三次元の壁**をうまく超えられないためである．例えば，YinYang [12]や Light Table では文字ベースのソースコードの右に変数の値を文字表示する．TouchDevelop では，ブラウザ上の Web アプリの実行画面を選択すると，当該画面を生成したソースコードが同じブラウザの画面上に表示される. 一方で, 実世界入出力を伴うプログラム開発では，取り扱う情報を文字や記号では分かりやすく表現できないことが多い．次節では，画像表現を用いてこの問題を解消する手法を提案する．

4. 画像表現を用いた開発支援手法

本節では，実世界入出力を表す写真や動画といった画像表現を IDE に組み込むことで実世界入出力を伴うプログラムの開発支援に役

立てる手法を提案し，具体例として 3 つの IDE [13][14][15]を紹介する．これにより，スクリーン上にありながら実世界の感覚を想起させる開発環境を実現し，**二次元と三次元の壁**を超えることを目指す．本提案手法はプログラムを生きたまま編集するという狭義の Live Programming (LP)の範疇には収まらないが，プログラムが生きているときの感覚を提供する LP 体験(LPX)を実現するものである．

4.1 写真を用いた入出力データの理解支援

実世界入出力を伴うプログラムの特徴として，センサやアクチュエータの操作に使う定数が実行環境に依存することが挙げられる．そこで，プログラマはまず，実際にセンサやアクチュエータを動かしてみて適切な値を探る必要がある．そして，得られた値を保存し，プログラムから読み込んで利用する．例えば，人の姿勢情報を取得し，特定の姿勢を取っているか判定するプログラムを書くときは，実際にその姿勢を取得，保存して使うことになる．

しかし，データを後から利用できるように保存する際は，名前をつける必要がある．このような姿勢データを保存するとき，どのような名前をつけていいか分からない姿勢が何個も生じることになる．これは，マウスやキーボードのイベントのように名付けが容易な例と対比してみると分かりやすい．標準化されたデバイスではイベントが取りうる値が初めから定まっているため名付けも容易だが，実世界入出力では値の組み合わせが無限通りあって，個別の事例に対して名付けを行うことが現実的でないのである．また，ソースコード中で姿勢情報のデータを利用するときには，事前に付与した名前を使って読み込むか，数値の羅列を記述することになる．いずれにせよ，ソースコードから実際の姿勢を想起することは困難である．このように，既存の IDE では実世界入出力データを直感的に理解できない．

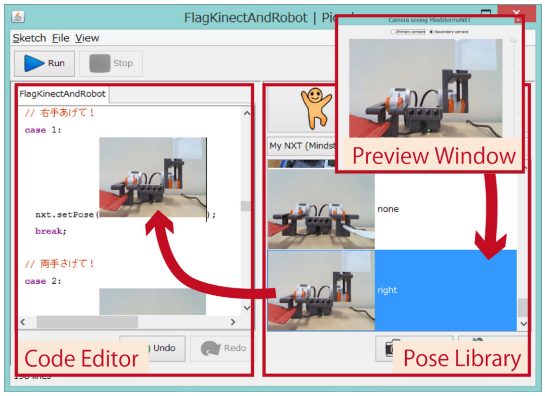

図 2. Picode の実行画面．

そこで，筆者は実世界入出力データを写真として表す Picode [13]という IDE を提案した．図 2 に示したように，Picode はデータを写真と紐づけて管理することにより，文字列ベースのエディタ中で姿勢データを表す写真をインライン表示できる．文字表現と比べると，どのような姿勢を表しているのか分かりやすくなっている．Picode は，人の姿勢情報のみならずロボットの姿勢情報も写真で表すことができ，ソースコードを見ることによって該当する部分がどのような姿勢判定を行おうとしているのか，また，どのような姿勢制御を行おうとしているのか一目瞭然となる．

このように，**ソースコードに写真という画像表現を組み込む**と，静的表現でありながらプログラム実行時の振る舞いを想起させる LPX を提供できる(図 3).

図 3. 写真による LPX の実現．

4.2 動画を用いた振る舞い理解支援

実世界入出力を伴うプログラムでは，センサからの入力やアクチュエータへの出力が連続的に処理される．ブレークポイントや，それに依存するウォッチテーブルのような既存のデバッグ手法は，プログラムへの入力を一時遮断し，この連続性を損ねるため利用できない．また，センサからの入力値には環境ノイズなどが混じるほか，ある瞬間と全く同じ状況は実世界では二度と用意できないため，同一の入力データ列を単純に再現することは事実上不可能である．そのため，マウスやキーボード入力についてのテストケースを用意する場

図 4. DejaVu の実行画面.

合と比較すると，バグの再現が難しい．

そこで，実世界入出力を伴うプログラム開発では，プログラムへの入出力データ列をプログラム実行中に記録しておき，あとから分析したり，入力データを利用してバグの再現性を確保したりする必要がある．しかしながら，既存の IDE では，プログラマは入出力データを記録・分析するプログラムを自前で用意しなければならない．また，録画した入力を使ってプログラムを再実行する仕組みを用意し，バグの再現性を確保する必要がある．

このようなワークフローを支援するため，筆者はプログラムの振る舞いを動画として表す DejaVu [14]という IDE を提案した．DejaVu はプログラムの振る舞いを動画として記録・管理し，それを 2 つのユーザインタフェース Canvas と Timeline で提示する(図 4)．プログラマは，Canvas に手書きでアノテーションしたり，変数をドラッグ&ドロップして内容を可視化したりできる．また，同内容が時系列で Timeline に表示される．プログラムを実行するたびに新しい動画が録画され，動画はいつでも好きな再生速度で再生できる．また，プログラムを書き換えたあと，録画された入力データを用いてプログラムを再実行できる．これにより，例えば姿勢情報を用いたプログラム開発において，姿勢をトラッキングするカメラの前と開発環境であるコンピュータの前を何度も往復する手間を解消できる．

このように，**プログラムの動的な振る舞いを動画という画像表現で表す**と，プログラムに実際の実世界入力を与えなくともプログラム起動時の振る舞いを理解しやすい LPX が提供可能である(図 5).

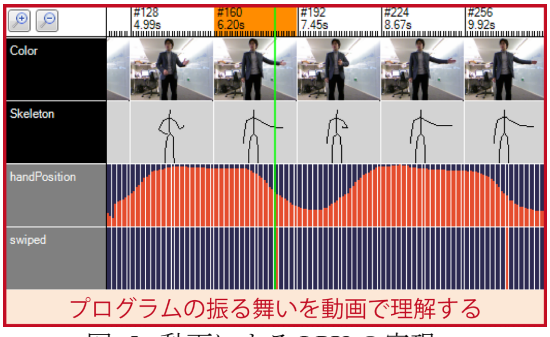

図 5. 動画による LPX の実現.

4.3 画像の編集操作による実装支援

これまでに紹介した画像表現はいずれも実世界で起こる状況を想起させる**理解**支援手法であり，プログラムの**実装**作業を直接支援するものではない．実世界入出力を伴うプログラム開発では，すでに実装済みの部分を実行した結果を利用してソースコードの次の行を書くようなインクリメンタルな実装プロセスを取ることが多い．例えば，画像処理パイプラインの実装では，実装済みのコードの出力結果を用いて機械学習のトレーニングを行い，その結果に対して更にフィルタをかけることがある．

通常の IDE でこのような作業を行うと，次のような問題が生じる．まず，コード補完の選択肢が多すぎる．例えば，画像処理用のライブラリである OpenCV では 160 個を超える画像処理用のメソッドが用意されており，どれが適したアルゴリズムか選ぶために通常のコード補完はほとんど役に立たない．また，プログラムを再実行する際は最初から実行し直しになるため，プログラムの規模が大きくなればなるほど実装を進めていくオーバーヘッドが大きくなっていく．

そこで，筆者は画像表現に対して GUI による操作を繰り返し行うことでプログラムを実装できる VisionSketch [15]という IDE を提案した．VisionSketch では，プログラムの実装済みの部分の実行結果が画像表現で表される(図 6)．主たるインタフェースは DejaVu の Canvas インタフェースと似通っているが，大きく異なるのは，VisionSketch では画像表現を GUI で操作可能な点である．具体的には，画像表現上に矩形や円形などの図形を描画すると領域選

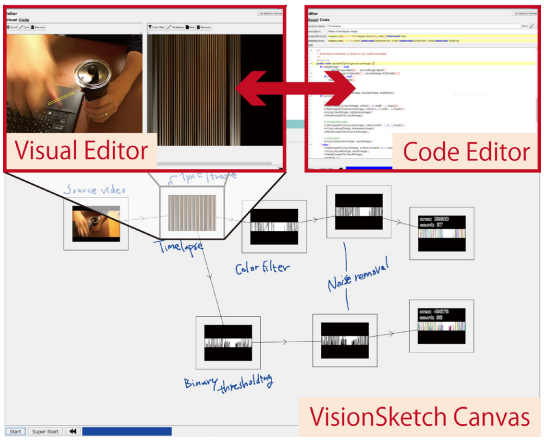

図 6. VisionSketch の実行画面.

択ができ，その領域に適用できるメソッドの一覧が表示される．さらにメソッドを選ぶと，即座に結果が表示される．描画済み図形もインタラクティブに編集でき，矩形を丸に書き換えると補完されるメソッドも変わる．すなわち，一般的な文字ベースの IDE におけるコード補完が変数の型に依存するものであるのに対し，VisionSketch では与えるパラメタの型から適用できるメソッドを絞り込んで提示する．このようなインタラクションは，プログラムを常に実行済みのところまで実行しておく Live Programming の既存手法[16]を利用して実現している．なお，既存のメソッドの処理内容に不満がある場合は，シームレスに文字ベースのコードエディタに切り替えて実装内容を編集できる他，新たなメソッドを実装できる．

このように，文字や記号ベースのプログラミング言語を操作するだけでなく，**画像表現とのインタラクションを通して実装作業を行えるようにする**ことで，実世界入出力に対応した LPX を提供できる．

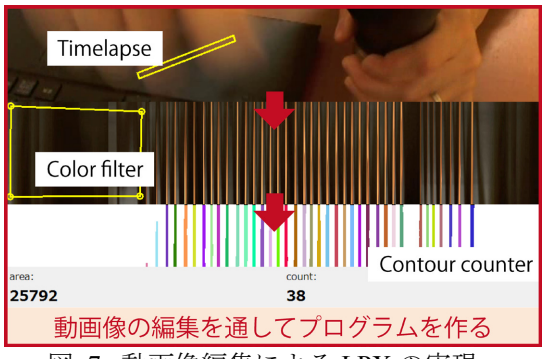

図 7. 動画像編集による LPX の実現.

5. 統合開発環境の未来

前節では，Live Programming の定義を「プログラムを生かしたまま編集できるようにする技術」から「プログラムが生きているときの感覚を与えるインタラクション手法」に拡張した．筆者は，今後の IDE は“Programmers (Software engineers) are people, too.”を念頭に置いて，プログラマの**プログラミング体験 (Programmer's eXperience, PX)**をより快適にするための発展を遂げるべきだと考える．本節では，IDE がどのような PX を目指すべきかという観点から IDE の未来について論じる．

5.1 クロスモーダル・マルチモーダルの活用

本稿で提案した画像表現による開発支援の自然な延長として，**視覚情報を用いて視覚以外の感覚を想起させるクロスモーダルな IDE** や，**視覚以外に訴えるマルチモーダルな IDE** が考えられる．

五感に訴える実世界の情報を取り扱う IDE では，その内容を表す画像表現を提示することでユーザビリティが向上することが考えられる．例えば，鰻のかば焼きの香りデータを提示するためには，「鰻のかば焼き香り」と文字で表すよりも鰻のかば焼きの写真を貼り付けたほうが，実際の感覚をより強く共起できるだろう．また，IDE が感覚フィードバックに対応したデバイスに接続されている場合は，データを表す画像表現を選択した際に感覚を再現できる．例えば，スピーカーやマイクを利用する音声情報処理，音楽情報処理のプログラム開発においては，ソースコードエディタ中で音のデータを表すために，文字列リテラルの代わりに波形を表示し，クリックすると音が再生されるようにすると直感的だと考えられる．触覚フィードバックを提供できるディスプレイ上に IDE が表示されている場合は，触覚データを表す画像表現に触れたときに，その触り心地を再生することができるだろう．

5.2 すべての人のための開発環境

エンドユーザプログラミングは，プログラミングの前提知識を持たないユーザでもコンピュータの自動化能力の恩恵に預かれるようにする試みであった．そのために，ビジュアルプログラミングや例示プログラミングといっ

たアプローチが開拓されてきた．しかしながら，ビジュアルプログラミングは文字ベースの言語と比較して組みやすいプログラムの種類に得て不得手があり，また，例示プログラミングはシステム側の推論を挟むため精密なロジックを組みにくい弱点がある．このように，昔ながらの高級言語によるプログラミングそのものを代替する方法は未だ生まれていない．

むしろ，オフィスワーカーが Excel のマクロなどを利用して業務を効率化したり，アーティストが Algorithmic Art などの分野で自己表現にプログラミングを利用したりするなど，文字ベースのプログラミングはより多くのユーザに利用されるようになってきている．近年では，プログラミングにより収入を得る者以外をエンドユーザと呼び，そのようなカジュアル層にターゲットを絞ったエンドユーザ・ソフトウェア工学[17]という言葉も生まれている．また，Computational Thinking [18]は，データのモデル化や処理の定式化といったプログラミングの諸概念は，コンピュータで解ける問題に限らず，より一般の問題解決にも役立つという考え方である．

これらの現状を踏まえると，アルゴリズムとデータ構造という概念はすべての人が学習すべきなのかもしれない．そこで，Victor が述べた Learnable Programming [19]のような視点が重要となる．Victor は主に数値などの簡単なデータ構造を操作するアルゴリズムの処理経過を分かりやすく提示する必要性を説いたが，一般の人が実際に解きたい問題は，より複雑なデータ構造を扱う場合が多いだろう．既存の IDE は簡単なデータ構造をプリミティブに持つ言語しかサポートしないことが多かったが，今後は**様々なデータ構造を分かりやすく提示し，アルゴリズム開発に活用できる IDE** が必要になると考えられる．本稿で提案した 3 つの IDE は，その先駆的な例と言える．

このように複雑なデータ構造へのサポートを先鋭化した形態の一つが，様々なデータ構造を直感的かつ効率的に編集できる，インタラクティブなコンテンツ制作ツールと言えるかもしれない．既存の例としては，Adobe Flash や Unity が挙げられる．いずれもコンパイラを備えているため，出力フォーマットを HTML と JavaScript か独自形式(*.swf, *.unity3d)から選べる．とくに Flash に関してはブラウザの対応状況などで紆余曲折あり，一時存在が危ぶまれたこともあってかなり意識的に「出力フォーマットにとらわれないような多目的制作ツールになる道を進んでいく」と言われている[20]．IDE はプログラムという一種のコンテンツを制作するためのツールであるから，コンテンツ制作ツールと共通の知見で使い勝手を改善できると考えられる．

5.3 自己進化可能な制作支援ツール

IDE は，プログラミングに必要な機能を全て提供することで，そのワークフローを支援する「全部入り」のツールである．本稿で提案した 3 つの IDE は，開発対象のアプリケーションに応じてそれぞれオープンソースのライブラリや IDE をベースに開発したものだが，この手間は一般のプログラマが一々かけられるものではないだろう．

一方で，Vim や Atom などのテキストエディタにスクリプトを組み込んで自分が真に使いたい機能を取捨選択し，時に自分で実装するプログラマが数多くいる．これは，一般的な IDE よりもエディタのほうが手軽にユーザインタフェースを拡張しやすいからこそ可能なことである．また，Unity には，プロジェクト単位でツールのユーザインタフェースを拡張できる特別なクラス CustomEditor が用意されている．ただし，ユーザインタフェースをいじりやすければよいという訳でもない．かつての開発環境には，Morphic [7]のように見えているものすべての実装をすぐに編集可能なものもあったが，これは一般には普及しなかった．

以上のように，IDE は適切な自由度のいじりやすさを持っているべきだと考えられる．これは，一般のコンテンツ制作ツールにも言えることである．現時点でも様々な制作ツールにスクリプト言語が用意され，手作業を超えた制作効率を実現できるように配慮されているが，本質的な機能向上には別の IDE でプラグイン開発が必要なことが多く，ユーザインタフェースを拡張しづらい．

デジタルツールを駆使する次世代クリエイターに必要なのは，**ユーザが機能拡張を行い，知的生産の効率を向上できる自己進化可能な道具**ではないだろうか．そのために，IDE を含

むコンテンツ制作ツール設計者は，本質的な機能は壊せないよう，しかし柔軟に拡張できるようにツールの拡張性を設計する必要がある．筆者は，このような未来ビジョンの一例として，Kinetic Typography という動画表現のための制作ツール TextAlive [21]を試作している．

6. おわりに

本稿では，まず実世界入出力を伴うプログラム開発の難しさをソースコードとプログラムの実行状態の間にある二つの溝(**時間的な溝**と**二次元と三次元の壁**)として定義した．このうち**時間の溝**を解消する手法としてプログラムを生きたまま編集する技術である **Live Programming (LP)**を紹介した．さらに，**次元の壁**を超えるために画像表現を用いた開発支援手法を提案した．これはプログラムが生きているときの**感覚**を与える **Live Programming Experience (LPX)**を実現するものであり，従来の LP の範疇には収まらないため，その定義の拡張を提案した．また，今後**プログラミング体験(PX)**を向上するための指針を議論した．

実際に PX を向上するためには，プログラミング言語，ソフトウェア工学，HCI の垣根を超えた学際的な研究を推進する必要がある．ソフトウェア工学に関する国際会議 ICSE 2013 では Live Programming に関するワークショップが開催されたほか，翌年のプログラミングに関する国際会議 SPLASH 2014 では Future Programming Workshop が開催され，PX を重視した IDE 研究が盛り上がりを見せ始めている．SPLASH 2015 では，まさに PX にテーマを絞ったワークショップの開催が予定されている．

今後，日本でも IDE の研究開発がさらに盛んになり，人々の PX が向上することを願っている．また，筆者自身もそのための研究を続けていきたい．とくに，質疑でもご指摘いただいた，コンパイラコンパイラやパーサジェネレータのような，IDE 開発用 IDE については中長期的ゴールとして実現してみたい．

謝辞　博士論文執筆では多くの方々のお世話になった．また，PX に係る多様な専門分野の先生方に主査･副査を務めていただき，議論を重ねられたことは大変幸運だった．夏のプログラミングシンポジウムの登壇発表では，質疑応答を通して議論を深めることができた．この場を借りて関係各位に深謝したい．

参考文献

[1] Arnold, K.: Programmers Are People, Too., ACM Queue, Vol.3, Issue.5, pp.54-59 (2005).
[2] Myers, B.: Software Engineers are People Too, Invited talk, ICSE 2012, Zurich, Switzerland (2012).
[3] Kato, J.: Integrated Graphical Representations for Development of Programs with Real-world Input and Output (2014).
[4] Kato, J.: Integrated Visual Representations for Programming with Real-world Input and Output, In Adjunct Proc. of UIST 2013, pp.57-60 (2013).
[5] Kato, J.: 研究会推薦博士論文速報, 情報処理, Vol.55, No.9, pp.1017 (2014).
[6] Tanimoto, S. L.: VIVA: A Visual Language for Image Processing, Journal of Visual Languages and Computing, Vol.3, Issue.2, pp.127-139 (1990).
[7] Maloney, J. H., Smith, R. B.: Directness and Liveness in the Morphic User Interface Construction Environment, In Proc. of UIST 1995, pp.21-28 (1995).
[8] Hancock, C. M.: Real-time Programming and the Big Ideas of Computational Literacy (2003).
[9] McDirmid, S.: Living It Up with a Live Programming Language, In Proc. of OOPSLA 2007, pp.623-638 (2007).
[10] Victor, B.: Inventing on Principles, Invited talk, CUSEC 2012, Montreal, Canada (2012).
[11] Burckhardt, S., Fahndrich, M., Halleux, P., McDirmid, S., Moskal, M., Tillmann, N., Kato, J.: It's Alive! Continuous Feedback in UI Programming, In Proc. of PLDI 2013, pp.95-104 (2013).
[12] McDirmid, S.: Usable Live Programming, In Proc. of OOPSLA Onward! 2013, pp.53-62 (2013).
[13] Kato, J., Sakamoto, D., Igarashi, T.: Picode: Inline Photos Representing Posture Data in Source Code, In Proc. of CHI 2013, pp.3097-3100 (2013).
[14] Kato, J., McDirmid, S., Cao, X.: DejaVu: Integrated Support for Developing Interactive Camera-based Programs, In Proc. of UIST 2012, pp.189-196 (2012).
[15] Kato, J., Igarashi, T.: VisionSketch: Integrated Support for Example-centric Programming of Image Processing Applications, In Proc. of GI 2014, pp.115-122 (2014).
[16] Edwards, J.: Example Centric Programming, ACM SIGPLAN Notices, Vol.39, Issue.12, pp.84-91 (2004).
[17] Ko, A. J., Abraham, R., Beckwith, J., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The State of the Art in End-user Software Engineering, ACM Computing Surveys, Vol.43, Issue.3, Article.21 (2011).
[18] Jeannette, M. W.: Computational Thinking, Communications of the ACM, Vol.49, Issue.3, pp.33-35 (2006).
[19] Victor, B.: Learnable Programming, http://worrydream.com/LearnableProgramming/.
[20] Hall, A.: Flash (Pro)の未来, http://aphall.com/2014/10/future-of-flash-pro-ja/.
[21] 加藤 淳, 中野 倫靖, 後藤 真孝: TextAlive: インタラクティブでプログラマブルな Kinetic Typography 制作環境, WISS 2014 予稿集 (to appear).