気軽な編集と実行を対応付け他者への説明を促すプログラミング学習環境

近 藤 秀 樹†

プログラムの動作を確かめるためにデバッガのステップ実行や変数のインスペクションが有用な場面があるが，初学者が安易にデバッガを利用しても表層的にしか理解を得られないことも少なくない．既存のプログラミング言語やプラットフォームについての意味など「深い理解」を得るために，能動的なソースコード片の操作を支援し，その動作の違いを確かめ易くして，プログラムの動作についての内省を促し，他者に向けた説明を作らせることを支援する学習環境について述べる．

A Programming Learning Environment Afford To Associate Editing and Running Programs And Explain To Others

Hideki Kondo†

1. はじめに

本研究では，プログラムと実行結果とを適切に関連付け，他者との議論を促すことで，プログラムに関する深い理解を獲得させることを目指す．

プログラミングを学ぶ際，学習の過程では，自他の書いたプログラムを深く読めることが重要だが，初学者にとって独力で深く読むことは困難であり，表面的な理解にとどまることが珍しくない．たとえば変数の使い方や型についての理解をしないために，適切な変数名を考えつくことなく，表面的に変数名を当てはめてプログラムを書くことも珍しいことではない．

文章読解や理解の研究では，自分なりの理解を表出したり，収集したりした情報を適切に関連付け，それに基づいて他者と議論することで深い理解を獲得できる可能性[1)]がある．

しかしプログラムと実行結果とを関連付け，自分なりの理解を整理しながら理解を深めたいと考えても，そのような仕組みは十分には実現されて来なかった．

プログラムの実行とソースコードについても，自分の理解に基づいて気軽に整理・対応付けしつつ，他の学習者と議論できるようにすることで，深い理解の獲得を支援できる可能性がある．

本稿では，深い理解の獲得のために，ソースコードの編集と実行結果やアイデアを明示的に関連付けて表現するシステムを実現し，それをもとに他者と議論を行う手法を提案する．プログラムについての深い理解を獲得することは独力では容易ではないが，自分の表現したソースコードに基づいて他者と議論することを通じて理解の支援を試みる．

2. 初学者の問題

プログラミング初学者の問題の一つは，プログラムを深く読むことができないことである．「プログラムを深く読む」とは，ここでは単にソースコードの字面だけにとどまらず，プログラムがコンピュータ上でどのように実行されるのか，変数やメモリなどの計算資源がどのように利用されるのかを予測しながら，どのような出力が得られるのかを了解できるような読み方のこととする．

2.1 初学者のエラーの事例

ここでは初学者がプログラムを深く読むことができなかったと思われる事例を述べる．

2.1.1 変数の命名と活用ができない

初学者がプログラムを深く読むことに失敗している事例の一つは，変数の命名や活用ができないことである．プログラムを自分で書く際，ほぼすべての場面で

† 九州工業大学大学院情報工学研究院
Kyushu Institute of Technology
kondo@mse.kyutech.ac.jp

```
namae = "name"
puts "Hello, " + namae
```

図 1 変数を使うソースコードの例

```
namae = 0
while (namae < 3)
  puts "Hello, name"
  namae++
end
```

図 2 変数を応用するソースコードの例

変数が必要になるが，必要な数だけ変数を作り出し，それらに適切な名前をつけて利用することができない場合がある．以下にその事例を説明する．

Ruby で書かれた図 1 のソースコードは，たった一つの変数 namae が用いられているごく短いものである．ここでは namae という変数に文字列オブジェクトを代入している．

このコードを学習の例題として用いたとき，深く理解できていない学習者は while による繰り返しの場面でも適切な変数を利用できない場合がある．たとえば一部の初学者は図 2 のようなコードを書くことがある．

このようなコードを書く学習者には二つの問題点がある．「命名の問題」と「導入の問題」である．

「命名の問題」とは，学習者が変数に適切な名前を付けることができていない点である．図 2 では，ループのカウンタとして namae という名前の変数を使っている．これは 1 に示したソースコード中で用いられている変数と表面的にまったく同一であり，そのまま再利用されたものと考えられる．しかしながら両者の役割は本来全く異なるものであるので，そのまま再利用してよいとされることは稀である．前者のプログラム内での役割は名前を文字列として保持するという意味を含んでいるのに対して，後者の変数は while の繰り返しの回数を整数で数えるという役割だからである．この両者は明らかに異なるものであるが，同じ名前が使われているのは適切な命名とは言えない．

図 2 のようなプログラムであっても，出力結果は正しく意図したものになるので，それだけを考えれば動作には問題はないように見える．しかしプログラムが

```
def hoge
  namae = 10   # (1)
end

namae = 1      # (2)

def fuga
  namae = 20   # (3)
end

puts namae     # (4)
```

図 3 スコープの理解が正しくないことを示すプログラム

解決しようとしている問題への理解や，プログラムを書く活動そのものへの理解は，プログラミングに期待されるレベルには遠い．

もう一つの問題である「新変数の導入」とは，プログラム中に新しい変数を導入できないことである．2 のソースコードに対して，「“Hello, name” の部分の name を変数で置き換えられるか」を問うと，「変数が足りない」「もうすでに変数を使ってしまったからもうこれ以上変数は使えない」と学習者が主張する場合がある．1 のソースコードを読み取って学ぶ際，変数と namae という名前とが不可分のものとして了解されていると考えられる．

すでに理解が進んでいる学習者は，変数とはいくらでも作ることができることなどを了解しているため，このような返答は起きない．

2.1.2 変数のスコープが理解されない

初学者が深く読むことに失敗している別の事例では，同じ名前の変数がスコープを超えて使われることがある．たとえば 3 のようなプログラムについて考えてみる．このプログラムを実行すると何が表示されるかを初学者に予測させると，その答えは正解である”1” ではなく，“20” になる場合がある．

図 3 では，(2) と (4) は同一の変数であるが，その他はそれぞれ異なる変数である．これは def から end にいたる関数定義によってスコープが分けられているためである．

しかし一部の学習者は，プログラムが上から下に向かって実行されていくという原則を利用しつつも，def - end によるスコープを無視し，表面的に namae への代入を追跡していると考えられる．

```
namae = 0
while (namae < 3)
  puts namae
  namae = namae + 1
end
```

図 4 制御構造が正しく理解されていないことを示す例

```
0
1
2
```

図 5 制御構造を反映した正しい出力

スコープという概念を獲得していない状態でソースコードと実行結果だけを与えられても，こうした誤りに気づくことはそう容易ではない．概念を持たない状態では，単にソースコードだけを独力で読んでいても気づけるとは限らない．

2.1.3 制御構造が理解されない

初学者がプログラムを深く読むことに失敗している事例として，制御構造が正しく理解されていない場合が挙げられる．条件分岐や繰り返しなどの制御構造が無視され，あるいは正しく追跡されず，深く読み取ることができない場合は珍しくない．

たとえば図 4 に示すコードの実行結果について問うと，回答は “0” から”3” の間で多様になることがある．この場合も，プログラムが上から下に向かって実行されていくという原則を利用しつつ，while - end による繰り返しの制御が無視され，表面的に namae への代入を追跡していると考えられる．繰り返しの概念が無視され，puts 命令がひとつしかソースコード上に出現しないことから，値が一つだけ出力されるという了解も読み取ることができる．

しかし正解は決して単一の値ではなく，図 5 のような表示になる．行ごとに値が 1 つずつ表示され，3 行の出力が得られる．

2.2 独力での読みの浅さ

ここまでに挙げた事例では，いずれもプログラムの読みの浅さに問題がある．ソースコード上の表層的なレベルでの議論に止まっており，さらに深く理解することが容易ではない．

教員や TA，あるいはすでに理解している別の学生が「プログラムの動作を深く読む」「ソースコードを一歩一歩追いかける」ようアドバイスすることも珍しくはない．しかしそのアドバイスを受けた学習者が読みを深められることは少ない．

この問題は，単にその学習者個人の能力の問題というよりは，自分が十分な理解を得ているかどうかを自分で説明することが本質的な難しさを持っているためである．たとえば自分なりの理解に基づいてプログラムを書きあげたとき，それが間違っているかどうかは容易には分からない．自分なりの理解に基づいて書きあげたプログラムであるから，自らの理解に基づいて検証しても異なる結果にはならない．

そうした認識のバイアスがあるため，自分なりの理解が浅いものであることに気づき，プログラムの表層的でない理解を獲得することは，独力では困難である．せいぜい typo などの些細なエラーが発見できる程度であり，新しい理解を獲得することは難しい．場合によっては，プログラミングについて深く理解するどころか，コンパイルも通らないソースコードでも十分であると誤解され，提出されることすらある．

2.3 理解を深めるために

前節で，独力では浅い理解しか得られないことを述べた．この問題は個人のバイアスによるものなので，自分とは異なる視点から検証できれば，自分の理解を更新できる可能性がある．

その一つの手段は，プログラムを実行してその結果と照らし合わせて確かめることである．正しい理解に基づいて書き上げたと確信しているプログラムであっても，いったんコンパイルしたりテストしたりしてみれば，理解とは異なる動きをすることがすぐに分かる．その結果，自分の理解が浅かったことに気づき，より深く理解するための手がかりを得ることができる．文献の読解に対して概念地図による整理が有用であり，ソースコードやその実行結果，関連するメモやアイデアなどについても同様にして整理できる可能性がある．

もう一つは，他者への説明を行うことである．自分なりの理解を具体的なソースコードに照らし合わせて説明することで，他者の視点からその内容を評価することができる．他者は他者で，自分なりの理解を持ち，それは自分とは異なるので，その違いを議論することになる．その結果，自分の理解では不足しているところや，他者が得ている深い理解を獲得し，自分の理解を深めることができる．

つまり，(1) ソースコードとその実行結果や関連するアイデアなどを明示的に関連付けながら整理し，(2)

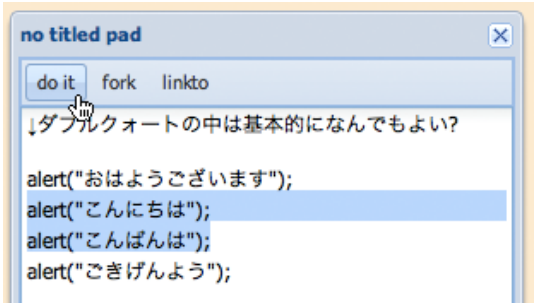


図 7 選択範囲をプログラムの断片として実行

必要に応じて他の学習者と議論ができれば，これまで以上にプログラムを読むことが促され，結果的に深い理解を獲得できる可能性がある．

3. 提 案 手 法

本研究では，プログラムの深い読みと理解のために，(1) 概念地図上でプログラムソースコードとアイデアを整理させるソフトウェアシステムの開発と，(2) それを用いて複数人で議論可能な教室とを提案する．

3.1 深い理解のためのプログラミング環境 necopad

複数のソースコードとそれぞれの実行結果やコメント，アイデアなどを関連付けて整理する環境 necopad を提案する．試作中の necopad のスクリーンショットを図 6 に示す．necopad は標準的なウェブブラウザ上で動作する．PC 以外にもスマートフォンやタブレット型コンピュータでも実行できるため，いつでも気軽にプログラムを編集して試すことができ，他の人間のところにコンピュータごと持って行き，necopad 上に表現したものを使って議論や質問ができる．

necopad は現実のソフトウェア開発を支援するためのものではなく，プログラムを深く読むための支援手法であることを目的としたシステムである．ソースコードやコメントを書き込むための単位として小さなウィンドウを複数開くことができ，ウィンドウに書き込んだ内容を部分的に選択して，JavaScript の断片として即座に評価 (図 7) できる．

また実行結果のメモや複数のソースコードをウィンドウで整理しつつ明示的にそれらを線で結びつけ (図 8)，関連を表現することができる．

これらの機能によって，いくつものプログラムのバージョンやそれぞれごとのメモ，説明を画面上に表現し，かつそれらを一望することができる．

ウェブブラウザの上で動作する JavaScript の開発環境は少なくない．また Java や C++などでのプログラミングを支援する統合開発環境 (たとえば Eclipse[2] や NetBeans[3]) も広く用いられているが，necopad は開発の支援を目的としているわけではない．プログラミングの学習やそれにともなう様々な過程を支援することが目的であり，ウィンドウ間の関連付けや気軽な編集と実行によってその目的を達成しようとしている．

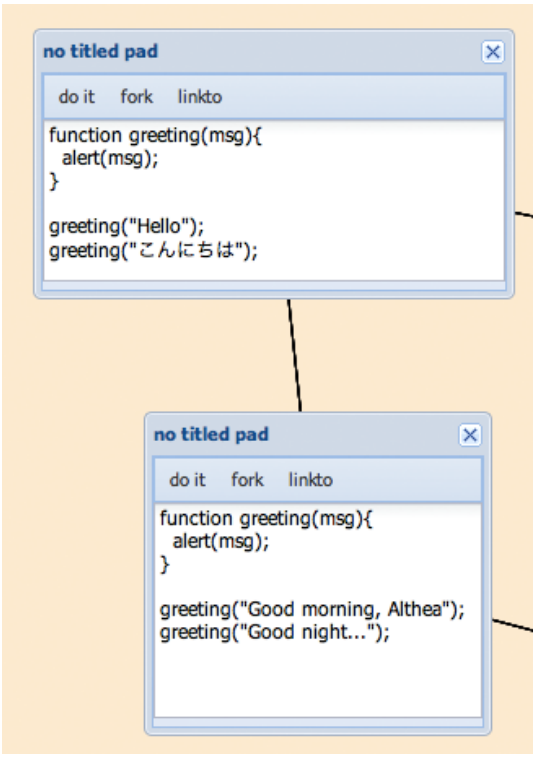


図 8 二つのソースコードを明示的に関連付ける

3.2 議論向けの教室設計

necopad によって，個人が自分なりの理解に基づくプログラムの記述やそれの実行結果などを関連付けて表現し，メモを残して，かつそれらを一望することが可能になった．しかしこれはあくまでも個人の活動であり，深い読みを支援するには十分でない．独力では深い理解が困難だが自分なりの理解に基づいて他者と議論することで，深く理解が可能になる．

しかし従来のコンピュータ教室に necopad のようなシステムを持ち込むだけでは十分な議論は難しい．そうした教室の多くはグループワークやディスカッションに適した構造にはなっていないためである．

- 什器が固定されているため，適切な相手と議論しにくい
  従来のコンピュータ教室の多くは，横長の机が採用されている．このため，一度に議論ができるのはせいぜい両隣のどちらかの人間とだけである．

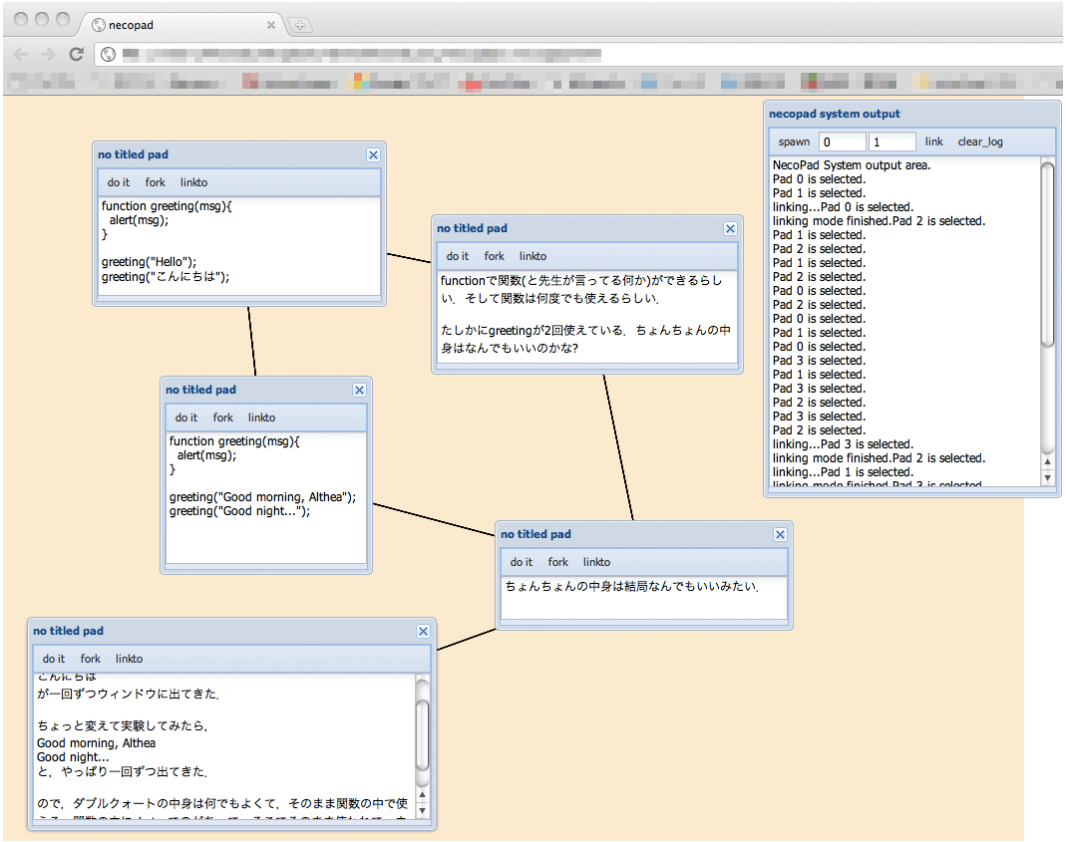


図 6 深い理解のためのプログラミング環境 necopad の全景

前後や反対隣の人間との議論は困難である．椅子も固定されている場合があり，このような場合，議論そのものがほとんど不可能である．

- コンピュータと画面が固定されているため，議論の材料を共有しにくい
  導入されているのはデスクトップコンピュータである場合が多く，横長の机に備え付けられている．モニタに向き合って利用することが前提となっていて，それほど広い範囲で見やすいわけではない．このため，necopad のようなシステムで整理したものがあっても，それを適切な相手に見せられるとは限らない．教室内の遠く離れた場所にいる相手に表現したものを見せることは困難である．

necopad のようなシステムが有意義に働くには，議論したい人のところへ行き，互いに自分の理解や相手の理解を具体的に見せながら質問・調査などが可能でなければならない．

本稿では，上に示した necopad を利用するためのグループワーク向けの教室の実例として，著者らの運用する MILAiS[4] (図 9) を提案する．



図 9 グループワーク向け教室

場所を共有する多くの学習者が互いに影響しあうことがこれまで以上に容易で，従来の授業スタイルにとらわれない，多様な学び方を実現している．

このような教室では活発な相互作用が可能になるように余裕のある空間を確保しており，積極的に設備や人の移動が可能になる．学生も教員も適切な相手のいる場所を見通し，そこまで移動できる．すでに necopad を用いた議論が始まっていることを見つけ，そこ



図 10 necopad の利用を教室によっても支援

に合流することも難しくない．

また，necopad のようなソフトウェアシステムだけでなく，ホワイトボードやイーゼルなどの什器によっても学習者のインタラクションが支えられており，これらも必要に応じて教室内を移動可能で，necopad での学習を支援する可能性がある．

こうした設備を併用することで，たとえば図 10 に示すような協調的な学習場面が想定される．

こうした利用は従来の教室では困難であり，necopad のようなシステムが活用されるためには，学習のための物理的な学習環境についても適切にデザインされることが必要である．

4. 課題と議論

necopad は試作中であり，まだ十分な機能が備わっているわけではない．むしろどのような機能が備わっているべきなのかについての議論が必要である．

- 現実的な機能の実装
  ソースコードとその実行結果を関連付けたり，少しずつ違う複数のバージョン間での比較のために，ソフトウェア開発で有用な機能がそのまま転用できる可能性がある．たとえば実行結果やソースコードの差分をとって比較しやすくしたり，複数のソースコードを自動的にバージョン管理するなどの機能を追加することで，深い理解を支援できる可能性がある．
- JavaScript 以外のプログラミング言語のサポート
  試作中の necopad は JavaScript を利用できるが，プログラミングについて学ぶ上で本当に JavaScript が適しているかどうかは明らかではない．本研究の中核は編集と実行，それらの関連付けを気軽に行うことであり，このことに適した言語がどのようなものかは検討する必要がある．
- 授業デザインの変更
  講義や個人演習だけでは necopad の利用は難しい．他の学生とのインタラクションを授業のデザインにどのように組み込むのは自明ではない．ソフトウェア開発の現場ではペアプログラミングのような手法も提案されているが，学習と開発の文脈は同一ではなく，単純に導入すればよいというわけではない．
- ウィンドウ間の連携
  necopad では小さな単位で深くプログラムを読み書きすることができるため，結果として，小さな単位で十分に理解し動作を確認したプログラムを作ることができる．これらを連携させることで，より大きなプログラムを理解しながら作ることができる可能性がある．

5. ま と め

本稿では，初学者はプログラムについて表層的な理解しか得られていない事例を検討した．独力では理解を深めることが困難な場合があることを議論し，プログラムを深く読むための支援手法として，概念地図に基づいてプログラムと実行結果やアイデアを関連付けて整理するためのシステムと，議論やグループワークに適した教室とを提案した．この手法により，他者との議論を促すことができ，自分なりの理解が不足していることに気づいて，さらに深く理解する必要性を明示することを意図している．

質 疑 応 答

Q 学生への指導がなかなかうまく行かない．学生は優秀だと思うが，やるのが大変なことを避けようとする．最終的にはうまく課題のプログラムを作るが，そのプログラムはどういうプログラムなのかは話せないようだ．学生同士はちゃんと話しあって課題を作っているように見えるが，プログラムをよく理解しているようには見えない．そこのところはどう思うか．

A 自分のプログラムを誰に見てもらったかとか，誰に聞いたのかなど，課題を達成しようとする過程を適切に把握することが有用かもしれない．単に答えを書いて実行して結果が出たから良いとするのではなく，その過程で何をしたのかということを分析する必要があるのではないか．学生同士がちゃんと話し合っているように見えて，単に一方的にしゃべっているだけということも珍しくはなさそうだ．一方で学生は，完全な回答だけを表に

出そうとする傾向があるようにも見られる．つまり，分からないことはまったく書かない．書けるという確信があって初めて書くようだ．しかし実際には，プログラムを書くときには中身は良くはわからないかもしれないが，とりあえず書いてみて，それでも書けないということを通じて，自分が分かっていないことを知り，最終的にはうまいプログラムを書くことができる気がする．その過程を何度も何度も取り扱うべきだと思っている．

Q プログラミングについての授業の TA をやってました．概念地図 (アプリケーションのウィンドウ群?) で，プログラム間の関係性はこのシステムでどのように扱われるのか? (中山様)

A いまのところ，システムの側で形式的に扱うことはしていない．プログラムを書いた本人その人が自分の考えを書き出して整理することを目的としているので，本人が分かることを優先すべきで，それ以上のことはまだやっていない．しかし一方で，fork メニューのように，システムが持っている機能を使って作られたリンクを区別することは可能であるし，それらを適切に視覚化することができれば，本人が自分の理解のプロセスをたどるときにも重要な情報を提供できると考えている．プログラムがどのように改造されてきたのか，その過程でどのようなメモを残していったのかをたどれることは，より深い理解を求めているときに役立つ指標となるかもしれない．

Q ソースを全部書いてコンパイルして通りませんと生徒に言われたりしました．(このアプリケーションでは) プログラムの候補をいくつか作ってそれを最後にマージするようなイメージでいいですか?

A そのとおり．課題をストレートに達成すればいいというのではなく，自分なりの理解を使って多様なバリエーションを作ってみて，どれも自分の理解に反しないようなことを確かめることでより深く理解をすることができると考えている．もちろんそれはそんなに簡単な過程ではなくて，自分の理解とは食い違う振る舞いをするプログラムを作ってしまうことがある．そんなときに，そのプログラムがどのように理解と違うのかを書いておいて，プログラムそのものと関連付けることができるのが強みである．そしてそれをそのまま他の人に見せながら説明したり議論を始めたりできる．そうして深く理解ができたら，最後にマージするような感覚になるかもしれないが，十分に理解ができているので，新しく書くことも容易だろう．

Q アプリケーションのウインドウとリンクについて，役立つような良いマップとはどのようなものか，例えばどういうものなのか．こういうのが良かったなどいい例を見せるほうがいいのではないか．(森下様)

A マップそのものに優劣があるとはまだ考えていない．そういう客観的な指標があるのかどうかまだ分からないし，一般原則があるかどうかも分からない．また仮に，こういうマップが良いマップだということをあまりに最初に見せてしまうと，今度はマップを表面的に真似するだけになってしまう．マップそのものはまずは自分の理解を深くするための手段であって，各人でどのようなマップがよいのかは異なっている可能性も高い．理解の助けになる程度の小さな断片を見せたほうがいいかもしれない．

Q ウィンドウ群について，論理的な構造を書けばいいのでは．

A プログラムの論理的な構造を視覚化することが目的というわけではない．その時その時の学習者の理解の状態を見ることが目的である．書く人がどういう状態で，どういう理解で書いたのかが重要で，そのウィンドウを見て，どこがおかしいかを議論することを狙っている．リンクのつながりや数を見て指摘できることはあるかもしれない．

Q 教材づくりに使えるかも?間違い例を作れれば，やっちゃいけないことを示せるかも．実行結果も一緒に見せるとわかりやすいか．

A たしかに過去の多くの人のマップを類型化できれば，あるいは自分の現在の状態と比較できるかもしれない．ただ，類型や実行結果を提供することで，学習者自身の探索や能動性が損なわれる危険性は常に意識しないといけないだろう．

Q お互い学生が見せ合うが，誰が見せたのか記録がない．わからない．記録を取らないといい効果がでないのでは?名前を書こう．

A 確かに現在のシステムではそういう記録は残らない．教室のデザインと一体のものとして他者と関わって議論を進めて欲しいし，そのような記録も込みで運用することが望ましいとは思うが，それとは別にシステム上での挙動も記録して追跡できることが必要だろう．名前での区別もそうだし，どこからどこまでが自分の仕事で，どこからどこまでが他者のコメントなのかも区別しないと，自分の理解を確認することが難しくなるだろう．名前とコメントとを自然に残したくなるような仕組

みが必要かもしれない．

Q プログラミングの授業をやってます．学生は仲間同士であまり教え合わない気がします．

A 環境が人に及ぼす影響はもっと深刻に捉えた方がよいだろうと考えている．そもそも仲間同士で教え合うような教室になっていないかもしれない．たとえば机も椅子もコンピュータも固定されていて，画面をのぞき見ることができないようプライバシーフィルタを貼り，授業外での学生のつながりを分断するような席順を強制して，ちょっとでも似通ったソースコードを提出したらカンニングの疑いがかかるような評価基準を徹底すれば，学生同士のインタラクションは自然に断絶されることになるだろう．独力での活動だけが評価され，他の手段が抑制される危険性が高い．しかしこれは従来の教室にいることで起きているだけであり，適切な環境があれば勝手に議論して教え合う可能性が否定されているというわけではないのではないか．現に，学生同士で skype のチャットなどで積極的に情報を交換している学生も一定数いる．従来型のやり方になじみのある学生はそれで問題がないが，そうした能動的な学生を抑制しているかもしれないということに注意したほうがよさそうに思っている．

参 考 文 献

1) 三宅なほみ, 2006,「学習科学：協調的な実践科学と理論構築との互恵関係を目指して」,『人工知能学会誌』, 21(1), 77-84.

2) `http://www.eclipse.org/` 2011 年 10 月 31 日確認

3) `http://netbeans.org/` 2011 年 10 月 31 日確認

4) 近藤秀樹、尾関智恵、楢原弘之,「インタラクティブな活動を促進する学習環境 MILAiS の構築と初期の運用の経過」, 第 27 回日本教育工学会全国大会論文誌，451-452.