卒 業 論 文

ゲーミフィケーションを利用したデバッグ技術学習法の考案と学習支援システムの実装

東北大学工学部

B7TB2507 川原田伸

指導教員　大堀 淳 教授，上野 雄大 准教授，菊池 健太郎 助教

提出年月: 令和 3 年 3 月 8 日

概要

本研究は，ゲームデザイン技法やノウハウを現実世界の活動に応用するゲーミフィケーションを用いて，プログラミング初学者を対象とした，プログラミングにおけるデバッグ学習の促進を目的とする．本論文ではまず，ゲーミフィケーションとデバッグ技術についての基礎を述べる．さらに，目的達成のための第一歩として SML# により試作した対話型デバッグ学習システムについて述べる．また，今後の発展と実用化に関する考察を行う．

目 次

第1章　序論

1.1　背景と目的

あらゆるシステムがプログラムによって動作している現代において，プログラミング教育の重要性は高まりつつある．大学や専門学校等でのプログラミング教育に加え，IT 企業への転職を目的としたプログラミングスクールや小学校でのプログラミング教育必修化もあり，情報工学を専門としない学生や社会人もプログラミングを学ぶ機会が増えている．多くのプログラミング初学者は，コンパイルエラーや実行時のエラーに直面した際，直感的なステートメントの変更や変数の値の出力を行うが，これは本来期待されるデバッグの方法とは異なり，学習効率の低下や直感的なデバッグの習慣化が懸念される．さらに，より実践的で大規模な開発でのデバッグを行う際に適切な対処をすることが難しく，学習におけるプログラミングと実践的なプログラミングの間に障壁が発生してしまう．したがって，プログラミング教育と並行し，デバッグの考え方を教授する機会を設けることは学習者にとって有意義なものであると考えられる．

しかし，プログラミング教育現場でデバッグ自体の講義があることは珍しい．実践的なデバッグの教本としては文献 [1] 等が存在するが，このような実践的な技術は多くの情報工学及び数学知識が必要となり，これを知識量が不足しているプログラミング初学者が自力で理解することは現実的でない．また，アルゴリズムやデータ構造といったプログラミングの学習は現実の問題解決に直結する能力である．それに対し，デバッグの学習はプログラミング学習の補助的な立場を取り，学習者がデバッグの学習を行う意義を理解することが困難であることが考えられる．

本研究では，このようなデバッグ学習に関する問題に対し，ゲーミフィケーション [2] の利用を試みる．ゲーミフィケーションとは，TV ゲーム開発において培われてきた人間を惹きつけるための技術やノウハウを，教育や社会生活に応用することを指す．例えば，RPG を遊ぶ感覚で運動を行うことができるフィットネスゲーム「リングフィットアドベンチャー」[3] は，健康維持活動に対するゲーミフィケーションの例である．ゲーミフィケーションを用いることで，プレイヤーはより意欲的に勉強や社会活動に取り組むことができる．この例のように，ゲーミフィケーションとデバッグ学習を組み合わせることにより，デバッグ学習をより意欲的なものとし，学習障壁を下げることが期待できる．本研究では，ゲーミフィケーションをプログラミングにおけるデバッグ学習に応用し，プログラミング学習者のデバッグ学習を促進することを目的とする．

また，本研究では，目的達成のための第一歩として次の 2 つアプローチを試みる．1 つ目は，書籍等の文章を読んで学習を行う従来の方法では達成が難しい対話的な学習環境を，ソフトウェアにより実現することである．開発言語及び学習対象言語として関数型言語 SML# を使用し，試作品である DebuQ を作成した．ソフトウェア設計にはゲーミフィケーションの考え方を取り入れた．2 つ目は，デバッグ学習を体系的にまとめ，難易度の低い項目から無理なく学習を行うことができるようにデバッグに関する練習問題を作成することである．作成した問題は DebuQ にて実装し，学習者が対話的に取り組める形にする．

1.2 関連研究

デバッグの学習を体系的にまとめ，プログラミングの授業に活用した例としては文献 [4] の授業パッケージと学習支援システムが存在する．デバッグの学習をシリアスゲームとして実現した例としてはRoboBUG [5] が存在する．本研究は，これら 2 つの間の立場を取り，主な作成対象を学習支援システムとするが，学習者の学習意欲向上を目的としてゲーミフィケーション要素を取り入れている．

1.3 本論文の構成

本論文の構成は次の通りである．2 章で，ゲーミフィケーションの概念と各種技術を述べ，プログラミング学習におけるゲーミフィケーションの適用方法を考察する．3 章で，バグが発生するメカニズムと，各種デバッグ技術の基礎について説明する．4 章で，SML# による学習環境システムの設計と実装方法について述べる．5 章で，関連研究について述べ，今後の発展と実用化に向けた課題を議論する．最後に，6 章で本論文の結論を述べる．

第2章　プログラミング学習におけるゲーミフィケーション

ゲーミフィケーションとは，ゲーム開発に用いられるノウハウや技術を，教育や社会生活等，ゲーム以外の活動に応用することを指す [2]．2010 年前後から，教育やビジネス利用を目的としたゲームの実践的な応用論が注目を集め，ゲーミフィケーションに関する本が続々と出版された [6]．本章では，ゲーミフィケーションの概要と各ゲーミフィケーション技術について言及し，さらにプログラミングへの応用例について言及する．

2.1　ゲーミフィケーション技術

TV ゲームにおいて，プレイヤーをゲームに惹きつけるためには種々の工夫を施す必要がある．TV ゲームプレイヤーが行うことは,「液晶画面等のゲーム画面からゲームに関する情報を受け取り，ゲームに接続したコントローラを用いてゲームシステムと対話を行い，この繰り返しから得られる様々な知見をもとにゲーム内で設定された目標を達成すること」である．より端的に言えば,「画面を見てボタンを押すこと」であり，単調作業である．ゲームとは，この単調作業に人間の特性を利用した意味づけを行い，プレイヤーに快楽を与えることと換言できる．この意味づけはゲームに限らず応用できる．ゲームにおけるプレイヤーを惹きつける技術を，ゲーム以外に応用したものがゲーミフィケーションである．

本論文では，ゲーミフィケーションに用いられる，ゲーム開発におけるゲームデザイン面でのノウハウや技術をゲーミフィケーション技術と呼ぶ．ゲーミフィケーションは，ゲーミフィケーション技術をゲーム以外の事象に適用し，対象となる人物の動機付けを行うことを目的とする．

ここでは，文献 [7, 2] による説明と著者による経験に基づいて分類した以下の 8 つについて説明する．このうち,「記録の可視化」と「継続ボーナス」以外の用語は文献 [2] あるいは文献 [7] において使用されている．

- 目標設定
- 成長の可視化
- 称賛演出
- 非強制的参加
- 記録の可視化
- 継続ボーナス
- アンロック
- 即時フィードバック

2.1.1 目標設定

目標設定は，十分に実現可能な目標を利用者に与えることを指す．このゲーミフィケーション技術の目的は，ユーザに行動を起こさせることである．

近年のよくデザインされたゲームにおいては，最初に低い達成難易度の目標が設定されることがしばしばある．ソーシャルゲーム等の，ゲームの熟練度が比較的低い非ゲーマーを対象としたゲームでは，ゲームを開始して最初にチュートリアルが設けられていたり，最初のステージが基本操作のみでクリアできるようなものとなっていることが多い．目標設定はプレイヤーが自分の力で目標を達成することにより，内発的な動機付けとして機能する．

2.1.2 成長の可視化

目標を達成したことを利用者が感じられるような仕組みを指す．目標達成までの距離を定量的に表現することで，プレイヤーの動機づけを行う．

RPG などのゲームにおける，レベルの概念が成長の可視化に相当する．ゲームをプレイするユーザーは常に自分のレベルを確認できるため，目標まであとどれくらいの時間と労力が必要かが可視化されている．しかし，現実世界においては自分のレベルは単調増加するわけではない．そのため，ゲーミフィケーションにおいては，ゲームにおけるレベルの概念をそのまま利用できるとは限らず，工夫して設計する必要がある．

2.1.3 称賛演出

目標達成や，それを目指した頑張りを称賛することを指す．このゲーミフィケーション技術の目的は，利用者に自分の成功を認識させ，成長実感を促して活動の継続や動機づけを行わせることである．

ゲームでは敵を倒したときやレベルアップをしたときに派手な効果音と演出が再生される．ゲームのプレイヤーはこの演出によりゲームが進んでいることを実感し，動機づけを行うことができる．現実の仕事や勉強では，自分が達成したことを実感する機会はゲームに比べて少ない．適切な称賛演出により，プレイヤーの動機づけを促進することができる．

2.1.4 非強制的参加

利用者が自ら能動的に活動に参加できるような環境を指す．利用者が活動したいときに活動することができ，活動したくないときには活動しなくてもよい，または活動をやめてもよい状態とする．

ゲームは好きなときに開始することができ，止めたいときにやめることができる．また，その要素がゲームを楽しむ姿勢につながっている．文献 [2] では，次のように述べられている．

> 強制的にゲームをプレイさせることで、おもしろいゲームをとたんにつまらなくすることもできる。たとえば、学校の授業中、先生がクラスの生徒全員に「あと七〇分以内に、マリオを六面までクリアできなかった者は単位ナシ！」と宣言すれば、おそらく、『マリオ』をたのしく遊ぶことが難しくなるだろう。あまりに強制的に他人から押しつけられた課題は、ゲームをプレイしているという感覚を失わせることになる。[2]

2.1.5　記録の可視化

利用者の成果をいつでも参照可能な形にすることを指す．このゲーミフィケーション技術の目的は，プレイヤーが目標達成や成功，自分の頑張りを定量的に認識し，モチベーションを高めることである．このゲーミフィケーション技術は成長の可視化と密接に関係し，多くの場合は記録を可視化することによって成長も可視化される．

レベル制が導入されている大抵の RPG においては，自身のレベルをいつでも参照することができる．もし，自身のレベルを参照できないようなシステムであったら，ゲームのプレイヤーは自分が目標まであとどれくらい頑張ればよいかがわからず，モチベーションを損ねてしまう．

文献 [2] 冒頭で述べられている，井上明人氏が作成した#denkimeter という節電ゲームは，家庭の電気メーターの数値を記録し，電気メーターの数値の増加を低くすることを目指すゲームである．#denkimeter のプレイヤーは，家の中の様々な家電の電源を落とすなどして，電力消費量を減らし，戦闘力を高めることを目指す．公式ホームページ [8] には，#denkimeter を利用して知人と対戦を行うためのルールなども示されており，節電をゲームにした例と言える．このゲームのように，実質的には利用者が自分の成果を確認できるようになっただけでゲームとなる場合もある．

ゲーミフィケーションの例と言えるかは難しいが，スマホ用アプリケーション「あすけん」[9] もゲーミフィケーション技術「記録の可視化」を利用して利用者のモチベーションを維持させている例であると言える．このアプリケーションは食事を写真にとることで，その食事の栄養などを AI が分析し，記録するアプリである．食事内容や現在の体重，自分で記録した運動量といった情報から，次の食事内容や運動量などの適切なアドバイスをもらうことができる．このアプリは，利用者の消費カロリーと摂取カロリーを可視化することでダイエットの成功率を高めている．このアプリの役割はあくまで記録と提案のみであるにも関わらず，ダイエットにおいて高い評価を受けていることからも，記録の可視化の重要性が伺える．

2.1.6　継続ボーナス

活動の成果ではなく，活動を行ったこと自体を称賛することを指す．このゲーミフィケーション技術の目的は，利用者が活動を継続させることである．

多くのソーシャルゲームでは，アプリケーションを起動することにより，通称ログインボーナスと呼ばれるアイテムやゲーム内通貨を受け取ることができる．このシステムは，ゲーム内の広告を利用者が見る機会を増やすためや，ゲームへの依存度を高めるためなど，採用理由はそのゲームによりまちまちであるが，結果として利用者は高い確率でゲームを継続して起動するようになる．

フィットネスゲームである Wii Fit では，ゲームを起動してからだ測定と呼ばれる簡単な測定とバランステストを行うと，ゲーム内のカレンダーにハンコを押すことができる．トレーニングは行っていないが，Wii Fit を起動する機会が増えることで，自然にトレーニングを行う回数も増える．

2.1.7　アンロック

ゲームをプレイする初期段階ではできることが限られているが，ゲームをプレイし続けることでできることが増えていくようなシステムを指す．具体的には，プレイヤーが使用可能なアイテムや技が段階的に増えること，行動範囲の拡大などが挙げられる．このゲーミフィケーション技術の目的は，初心者の利用者に活動の魅力や楽しさなどを伝えることで，活動を行う動機を与えることである．

アンロックの典型的な例としては，スーパーマリオ等で使用されているステージシステムがある．ステージシステムは，ゲームのプレイ単位であるステージに挑戦するために,「それ以前のステージを全てクリアしなければならない」等の条件を設定するものである．プレイヤーはステージを次々とアンロックし，ゲームクリアを目指す．

文献 [2] では，絵心教室 DS というゲームを例にして，次のように述べられている．

> まず、かんたんに線を引く練習。そして絵に色を付ける練習。筆のタッチのバリエーションによって質感を出す方法……と一つずつ、高度な内容にステップアップしていく「アンロック」の仕掛けをフルに活用していると言えるだろう。[2]

2.1.8 即時フィードバック

ゲームプレイ後に，プレイヤーが結果を即座に確認できることを指す．このゲーミフィケーション技術は「記録の可視化」の一部とみなすことができる．記録の可視化では，自分の活動内容をいつでも確認できることを指すが，即時フィードバックは活動によって得られる結果を確認できることを指す．

フィードバックの速度が遅かったり，自分の状況を確認できない時間が長いことは，利用者のモチベーションを維持させるという点においては望ましくない．人の欲望は，時間が経つにつれ弱くなっていくのである [2]．

2.2 プログラミングゲーム

プログラミングをゲーム化しようという試みは数多く存在する．

プログラミング学習サイト Progate [10] では，演習をクリアする毎に経験値が与えられ，経験値をためてレベルを上げていくシステムとなっている．各演習では，演習テーマに関する解説スライドで学習を行った後，コーディングを行う課題に取り組む．自分で書いたコードの実行結果はブラウザ上で即座に確認できる．プログラミング言語の他にバージョン管理システムの使用法など，ソフトウェア開発に必要な種々の演習が用意されている．Progate においては，学習の最初から最終演習に取り組む，というようなことはできない．学習者は，簡単な演習をこなしていき，段階的に学習を進めていく．これはゲーミフィケーション技術における目標設定が適用されていると見なすことができる．自分が書いたプログラムとその実行結果が 1 つの画面上で即座に確認できるシステムは，ゲーミフィケーション技術における即時フィードバックである．このように，種々のゲーミフィケーション技術を適用することにより，一層意欲的な学習体験が得られる．

プログラミングをテーマとしたエンターテイメントとしては，AtCoder [11] などの競技プログラミングがある．AtCoder では，定期的にプログラミングコンテストが開催されている．プログラミングコンテストで出題される問題に早く正確に回答することで得点が得られる．コンテストには通常複数の問題が用意されており，各問題の難易度に応じて得られる得点が設定されている．さらに，オンラインゲームでのレーティングにあたるシステムが導入されており，自分の実力が数値によって可視化されている．また，個人でプログラミングの学習を行う際，作成したプログラムで大量のテストケースを用いたテストを行うことは手間であり，初学者には難易度が高い．その点，競技プログラミングはプログラムを提出することにより自動でテストが行われ，合否がすぐに確認できる．従来のプログラミング学習では，課題をクリアするという概念が曖昧であったことに対し，競技プログラミングでは課題のクリアが明確に提示される．これがゲーミフィケーション技術における即時フィードバックと同様の効果をもたらし，学習者の動機づけを促す．

これらの例は，プログラミングに適切なゲーミフィケーション技術を適用し，ゲーム化を実現できることを示している．そこで，以下 2 つのゲーミフィケーション技術をプログラミング学習に適用することを考える．

1 つ目はアンロックである．ゲームに限らず，難易度の低い課題から順に学習を進めていくことはあらゆる学習の正攻法として知られている．しかし，現在のプログラミング学習においては学習のレベルデザインが確立されているとは言い難い．これは，プログラミング能力が数学，アルゴリズム，データ構造などの複数の能力から構成されることによる．また，それぞれの能力は互いに干渉しており，中学や高校数学における学習の一方向性がない．例えば，アルゴリズムを習得するためにはある程度プログラムが書ける必要があり，プログラムを書く場合もある程度アルゴリズムを理解していなければならない．より具体的には，C 言語でソートアルゴリズムを理解するためには，プログラミング知識として配列の理解が必要である．また，配列の存在意義や，配列を使用するメリットの理解を確かなものにするためには，種々のアルゴリズムを配列により実装することが不可欠である．

2 つ目は即時フィードバックである．自分が書いたプログラムが正しいかどうかを確かめるためには，プログラムを実行して何度かテストを行う必要がある．しかし，このテストケースを作り，テストを行うという工程は時間がかかる上，手作業では十分な量のテストを行えない場合が多い．これは他の分野の学習と比較しても明らかにフィードバックが遅い．プログラムで解決したい任意の課題について，プログラムによる課題の実現方法は一般に無数に存在する．学習段階においてはこの多様性を排除し，正解と不正解を明確に区別することで即時フィードバックが実現でき，学習意欲向上が期待できる．

このように，プログラミング学習にはゲーム化できる複数の点が存在するため，プログラミングという分野においてゲーミフィケーションを行うことは十分に可能であると考えられる．

2.3 シリアスゲーム

教育や軍事演習など,「シリアスな用途」に用いられるゲームをシリアスゲームという [2]．シリアスゲームとして有名なタイトルとして,「フード・フォース」がある．これは，アフリカなどの食料が不足する地域に食料を運ぶ仕事を疑似体験できるゲームである．WFP(国連世界食料計画) によってリリースされたゲームであり，このゲームを一通り終えると，WFP が従事している仕事の理解を深めることができる．

一方，ゲーミフィケーションといった場合，必ずしもゲームを開発する必要はない．ゲーム以外の活動にゲームの要素を取り入れれば，それはゲーミフィケーションとして成立する．たとえば，男性用小便器に的当てマークのシールが貼られていることがあるが，これは尿が周囲に飛散することを防ぐ目的を果たしている．文献 [2] では，この例はゲーミフィケーションの第一歩として位置づけられている．

ゲーミフィケーションとシリアスゲームの対比として，文献 [2] では，次のように述べられている．

> 「シリアスゲームは社会のさまざまな問題をゲームのなかに持ち込むことだが、ゲーミフィケーションはゲームを社会のさまざまな場所に持ち込むこと」[2]

この定義には，事例をゲームと捉えるかどうかという点で多少の曖昧性が含まれる．例えば，フィットネスゲームである Wii Fit はゲーミフィケーションだろうか．文献 [2] では，Wii Fit は健康・身体分野での，体重を測定することによるゲーミフィケーションに分類されている．しかし，Wii Fit は，

社会の健康問題をゲームの中に持ち込んだものと捉えればシリアスゲームである．要するに，Wii Fit がゲームであるのか，ゲーム以外の活動であるのかによってシリアスゲームかどうかが決まる．

本論文では，文献 [2] の定義をそのまま用いるのではなく，次のように定義する．「シリアスな用途で用いることが可能なゲームをシリアスゲームという．そして，シリアスゲームを教育や社会生活等，ゲーム以外の活動に用いることはゲーミフィケーションのひとつの形である．」つまり，シリアスゲームはゲームのひとつであり，シリアスゲームを教育や社会生活等，ゲーム以外の目的に用いることはゲーミフィケーションであるとする．

第3章　デバッグ技術

不正なプログラムの動作や状態，プログラムコードをバグと言い，バグを除去することをデバッグという．本章ではバグが発生するメカニズムと，デバッグの際に使用される種々の技術について述べる．以下で使用する用語「欠陥」「感染」「障害」及び 3.1 バグが発生するメカニズム，3.2 デバッグの手順， 3.3 制御フロー，3.4 事実の観察における記述は文献 [1] による説明を参照した．

3.1　バグが発生するメカニズム

以降，バグのうち不正なプログラムコードを欠陥，不正なプログラムの動作を障害，不正なプログラム状態を感染と呼ぶ．欠陥は感染の原因であり，感染がプログラムの外部から見える形で現れたものが障害である．

すべてのコードはプログラマが書くものである．よって，すべての欠陥はプログラマによって作られることになる．ただし，欠陥の存在とプログラマの失敗は同値ではない．不完全な仕様，モジュール間のインターフェースの非互換など，プログラマだけが原因であるとは言えない場合があるためである．時には，誰も間違っていなくても障害は発生しうる．当初は欠陥でなかったコードも，ソフトウェアの使用方法の変化により欠陥になることもある．

欠陥により，プログラムの状態はプログラマが意図しないものになる．これが感染した状態である．プログラム状態の実態はプログラムに関係する全ての機器におけるレジスタの値である．したがって，ネットワークに接続された機器におけるプログラムでは，そのネットワークに接続されている全ての機器の状態がプログラム状態となる．現実にはプログラム状態にはハードウェア等による境界が設けられる．

プログラムの状態が感染すると，その後のプログラムの状態は多くの場合現在のプログラムの状態に依存するため，感染はさらなる感染を引き起こす．

感染により，プログラムの振る舞いが，プログラマが想定した振る舞いとは異なったものとなり，障害として確認される．

3.2　デバッグの手順

デバッグの手順は，次の 7 つの段階に分けられる．それぞれの頭文字から，TRAFFIC と呼ばれる．

1. Track: 問題を記録する．
2. Reproduce: 障害を再現する．
3. Automate: テストケースの自動化と単純化を行う．
4. Find: コード中の感染源の疑いがある箇所を見つける．

図 3.1: 制御フローグラフ

5. Focus: 感染源の疑いが最も高い箇所に着目する．
6. Isolate: 感染の連鎖を分離する．
7. Correct: 欠陥を修正する．

本章では以降，第 4 章の準備として，Find における制御フロー，Focus における事実の観察について説明する．

3.3 制御フロー

以下に引数 n の階乗を計算する C 言語の関数 fact() を示す．

```
1 int fact(int n){
2     int a = n, b = 1;
3     while (a > 0) {
4         b = b * a;
5         a = a - 1;
6     }
7     return b;
8 }
```

関数 fact() は，関数内で定義された変数 a と b に繰り返し書き込みを行うことで階乗の計算を実現している．この制御フローを図示すると，図 3.1 のようなグラフとなる．制御フローグラフは情報がどのように波及していくかを示すものである．ここでは，どのように個々のステートメントが情報フローに影響するか，どのようにステートメントが情報フローから影響を受けるのかについて説明する．

3.3.1 ステートメントが与える影響

すべてのステートメントは計算に関与しているため，少なくとも潜在的には必ず情報フローに影響を与えていることになる．情報フローへの影響は次の 2 つに分けられる．

- 書き込み (Write)
  変数に値を代入することを指す．これにより，プログラムの状態を変更する事ができる．ここで言うプログラムの状態とは，極めて広い範囲を指し，例えばネットワークに接続されたプログラムが実行されればネットワークに接続されているすべてのデバイスの状態が変わる．現実的にはハードウェア境界などで考える範囲を限定する．
- 制御 (Control)
  プログラムカウンタを変更することを指す．図 3.1 では，ステートメント 3 の while 文によって，次に実行されるステートメントが 4 か 6 に決定される．厳密にはプログラムカウンタも変数であるからプログラムの状態の一部であるが，ここでは実用的にプログラムカウンタを時間として考える．

3.3.2 ステートメントが受ける影響

前述したことは，ステートメントが能動的に情報フローに影響を与えるということについてであった．一方で，ステートメントは他のステートメントから影響を受けることがある．以下に例を示す．

- 読み込み (Read)
  プログラムの状態，つまり変数の値を読み込むことを指す．例えば,「v2=v1+1」というステートメントは，変数 v1 から値を読み込んでいるため，変数 v1 の状態に影響される．書き込み (Write) と同様，ここでの「プログラムの状態」は広い範囲を指す．

- 実行 (Execution)
  ステートメント A が影響を受けるためには，ステートメント A が実行されなければならない．一般に，ステートメント A の実行が，別のステートメント B によって潜在的に制御されているならば，ステートメント A はステートメント B に影響を受ける．

3.3.3 ステートメントの依存関係

ステートメント間の依存関係は，次の 2 種類に分類できる．

データ依存 (Data dependence)

次の条件両方にあてはまる場合，ステートメント B は，ステートメント A にデータ依存していると言う．例えば，図 3.1 のステートメント 5 はステートメント 1 にデータ依存している．

- A がある変数 V に書き込み，その変数 V を B が読み込んでいる．
- 上記の変数 V について，制御フローグラフで，A から B への経路があり，それらの経路のうち 1 つ以上「変数 V が他のステートメントから書き込まれていない」経路が存在する．

制御依存 (Control dependence)

ステートメント B の実行が，潜在的にステートメント A に制御されているとき，B は A に制御依存していると言う．例えば，図 3.1 のステートメント 4,5 はステートメント 3 に制御依存している．

3.4 事実の観察

実際のプログラムの動作は，観察を行うことにより特定できる．観察テクニックには古典的なログ出力，対話型デバッガ，事後デバッグ等が存在するが，ここではログ出力について言及する．

3.4.1 観察の目的

前述したように，バグはプログラム状態の感染が原因である．さらに，感染はプログラム中の欠陥によって発生する．これらのうち，ログ出力ではプログラム状態を観察することで感染が発生しているかを判断する．

観察にあたっては，以下の原則を遵守する必要がある．

- 本来のプログラム実行に干渉しない．
  観察によりプログラム状態が変化してしまうと，本来のプログラム実行でのプログラム状態を正しく特定することが困難となる．
- 観察対象，観察タイミングを限定する．
  プログラム実行はプログラム状態の膨大な連鎖であり，全体を観察することは現実的ではない．観察対象とタイミングを限定し，より本質的な観察とする必要がある．
- 科学的手法に則る．
  闇雲に観察を行うのではなく，仮説を立て，仮説を検証するために観察を行う．

3.4.2 観察の方法

事実を観察する最も簡単な方法は，print 関数等によるログの出力である．通常，プログラミング言語には C における printf 関数，SML# における print 関数等の標準出力関数が用意されている．しかし，この方法には以下のような難点もある．

- ログ出力ステートメントと元のコードとの混在による，コードの煩雑化
- 観察対象イベント及び変数の種類が膨大になった際の，出力の煩雑化
- print 関数の引数を展開することによる，プログラム全体におけるパフォーマンスの低下

これらの問題に対応するためには，次のようなことが考えられる．

- 出力フォーマットの統一
- ログ出力をオン・オフ可能にする
- ログ出力の詳細度を変更できるようにする
- ログ出力を再利用できるようにしておく

3.5 デバッグ技術とゲーミフィケーション

本研究では，SML# を対象としてデバッグ学習システムを構築する．ここでは，SML# におけるデバッグ環境と，SML# でのデバッグ体験をどのようにゲーム化するかを考察する．

3.5.1 SML# におけるデバッグ

これまでに述べたデバッグ技術は，SML# でのプログラミングにも適用することができる．ただし，C のような手続き型言語とは異なったアプローチが必要なデバッグ技術もある．例えば，C ではプリプロセッサにより元のプログラムに干渉せずに事実の観察が行えるが，SML# にそのような機構は存在しない．その代わりに，SML# では対話環境によってデバッグを行うことができる．対話環境では，次の事が可能である．

- ログの出力
  SML# における対話環境では，自分でプリンタ関数を作成することなく，基本型と自分で定義した型をプリントすることができる．さらに，プログラム中に現れた変数の推論された型も表示される．これにより，プログラマは自分が書いたプログラムの型と内容が想定したものと一致しているかを確認できる．

- 単体テスト
  作成した関数等が正しく振る舞うかをテストすることができる．問題の修正を行う際に，その問題の再現が可能な場合は，対話モードで簡単に任意の引数による関数の呼び出しを行うことができる．

- プログラムに干渉しない
  ソースコードに直接プリンタ処理を加える場合，必ず副作用が生じてしまう．それに対し，対話環境を使用することでプログラムを外側から観察することが可能であり，元のプログラムに干渉する心配がない．

SML# は関数型言語の特性上，変数の値は参照型を除き変化しない．そのため，誤った値が代入された変数を発見できればそれが感染である．よって，手続き型言語と比べてデバッグ時に時間的なプログラム状態の側面を考える必要がなく，空間的な誤りを発見することが目標となる．

3.5.2 デバッグのゲーム化

デバッグ作業では，多くの場合，次に何をすればよいかが不明瞭になる時がある．これは，効果的なデバッグ技術の使用方法を知らないプログラミング初学者にとっては挫折の原因となりうる．デバッグをゲーム化するためには，ゲーミフィケーション技術の成長の可視化 (2.1.2) で述べたように，学習者が作業の前進を通じて自分自身の成長を実感できることが重要である．本研究では，具体的な障害の例とその解決策を学習者が取り組む演習問題として作成することにより，各種デバッグ技術の存在とその使用方法を教授することを目指す．

第4章　デバッグ学習環境システムの設計と実装

デバッグ作業なしでプログラミングの学習を行うことは困難である．しかし，学校の授業等でデバッグ自体の方法を学ぶ機会はほとんどない．多くの学習者は明確な根拠なしにログを出力し直感的にバグを発見するが，これでは実践的なデバッグ技術が必要になった際に適切なデバッグが行えるとは考えにくい．

そこで本研究では，プログラミング学習者のデバッグ学習を促進するための第一歩として，学習環境 DebuQ(デバック) を SML# にて構築した．

4.1　設計

ここでは，実装したプログラムの外部仕様を説明する．

開発した学習環境は，Linux の shell 上で動作する対話型コンソールアプリケーションである．学習の流れは Progate [10] を参考にした．学習者は学習したいタスクを選び，学習テーマに関する説明を読んだ後に，出題される問題に回答する．なお，本プログラムではタスク開始から終了までの一連の流れをまとめてステージと呼ぶことにする．これをまとめると，学習者は以下の項目を繰り返しこなすことによりデバッグ技術を獲得できるようになっている．

1. 表示されるステージ一覧から，ステージを選択する．
2. 表示される説明を読む．
3. 表示される問題を解く．
4. ステージをクリアしたら，1. に戻る．

グラフィックを使用せずにコンソールアプリケーションとした理由は，開発期間が短いこと，デバッグ学習の内容を考察することを主な目的としたためであるが，shell 上で動作することにより，多くの環境で応用が可能となる利点も存在する．DebuQ は shell 上で動作するため，例えば shell 機能を持つ任意のエディタ内の shell で動作させることが可能である．これにより，学習者はコーディングを行いながら学習を進めることも可能である．

DebuQ で用いているゲーミフィケーション技術は，以下の 2 つである．

- 即時フィードバック
- アンロック

学習者は，問題に回答するたびにフィードバックとして回答の評価が得られる．正答した場合だけでなく，誤答をした場合にもなぜ間違えたかを説明するため，学習の躓きを防ぐことが期待できる．

また，DebuQ 開始時のステージは 1 項目のみ表示されており，ステージをクリアする毎に新しいステージがアンロックされる仕組みとなっている．これにより，学習者は次に何を学習すれば良いかを具体的に知ることができるため，モチベーションの維持が期待できる．

DebuQ で学習できる内容を以下に示す．

- エラー出力の読み取り
- 科学的手法
- ステートメントの依存関係
- ログの出力

エラー出力の読み取りは，文法ミス等によって発生するコンパイルエラーからプログラムの欠陥の場所を特定する能力である．これは比較的簡単に解決できるものの一つであるが，プログラミング初学者がこのエラーを見た場合，母国語以外によるエラーメッセージの表示や例外名などの出力により混乱してしまう可能性がある．DebuQ では，エラーメッセージのうち注目する箇所を行番号に限定し，欠陥の修正を試みる課題を採用した．

科学的手法とは，障害に対して既に確認された事実と矛盾しない仮説を立て，その仮説の真偽を確かめることにより欠陥の場所を絞り込む手法のことである．DebuQ では，SML# の対話環境によるログの出力を通じて仮説を検証する問題を用意した．

ステートメントの依存関係は，3.3.3 で述べた依存関係のことである．このうち，DebuQ では主にデータ依存の関係を追跡する課題を用意した．

ログの出力では，3.4 で述べた方法により，障害に関する事実を観察する．

4.2 実装

作成したプログラムは，次の 2 つのモジュールから成る．メインルーチンのプログラムは付録 A.1 に，問題演習ルーチンのプログラムは付録 A.2 に示した．

- メインルーチン
  ステージの選択，挑戦できるステージの表示，達成状況の表示等，プログラム全体の制御を行う．
- 問題演習ルーチン
  指定された問題ファイルの読み込み，説明の表示と問題の出題，入力の受付，正誤判定等の各ステージにおける処理を行う．

メインルーチン main.sml では，main 関数に学習の進行状況を示す変数 env を渡し，全てのステージをクリアするまでループを行う．変数 env の型は以下のレコード型である．

```
env : {task : string list, appeare : int list, unlock : int option}
```

task 属性は，各ステージのタスク名をリストで保持する．appeare 属性は，プレイヤーが挑戦可能なステージの番号をリストで保持する．unlock 属性はアンロックするステージの番号を表し，直前に挑戦したステージをクリアしたなら，その次のステージ番号を n として `SOME n`，直前にステージをクリアしていなかったら `NONE` とする．

stage.sml では，問題ファイルの字句解析を行う Lexer ストラクチャ，構文解析を行う Parser ストラクチャ，文章の表示や問題の出題等を行う Play ストラクチャの 3 つが定義されている．stage.sml における字句解析を行うストラクチャLexer のシグネチャLEXER を以下に示す.

```
signature LEXER =
sig
  exception formatError

  datatype command =
    Select
  | SomeSelect
  | Answer of int * bool
  | Input
  | SomeInput

  datatype token =
    Statement of string
  | Split
  | Begin of command
  | End

  val lexer : string -> token

  val fileLexer : string -> token list
end
```

command 型は，コマンドを示す型である．token 型は，トークンを示す型である．問題ファイル中の 1 行が 1 つのトークンに対応し，トークンは文章 Statement，文章の区切り Split，コマンドの開始 Begin，コマンドの終了 End からなる．Select は選択問題，SomeSelect は複数選択問題，Answer は選択問題の選択肢，Input は文字列を答える問題，SomeInput は複数の文字列を答える問題を表している．lexer 関数は，一行の string 型の値を受け取り，それを token 型のデータに変換する．fileLexer 関数は，ファイル名を string 型の値として受け取り，対象ファイルの各行を token 型に変換した値のリストを返す.

以下に，サンプル問題ファイル sample.txt を示す.

```
問題です.
(1) 正解
(2) 不正解
(3) 不正解

<begin select>
<begin answer 1 true>
(1) は正解です.
```

```
<end>
<begin answer 2 false>
(2) は不正解です.
<end>
<begin answer 3 false>
(3) は不正解です.
<end>
<end>
```

sample.txt を字句解析した結果を以下に示す.

```
# Lexer.fileLexer "sample.txt";
val it =
  [
    Statement "問題です. \n",
    Statement "(1) 正解\n",
    Statement "(2) 不正解\n",
    Statement "(3) 不正解\n",
    Statement "\n",
    Begin Select,
    Begin (Answer (1, true)),
    Statement "(1) は正解です. \n",
    End,
    Begin (Answer (2, false)),
    Statement "(2) は不正解です. \n",
    End,
    Begin (Answer (3, false)),
    Statement "(3) は不正解です. \n",
    End,
    End
  ]
  : Lexer.token list
```

sample.txt における”<”と”>”で囲まれた行はコマンドを示しており，その他の行は学習者にそのまま表示する文章として字句解析が行われている．各 Answer コマンドで指定された文章は，学習者がその選択肢を選んだ際に表示される文章を表している．

構文解析処理を行うストラクチャParser のシグネチャPARSER を以下に示す.

```
signature PARSER =
sig
  datatype qtype =
    Statement of string
  | Split
```

```
    | Select of {index : int, pass : bool, msg : string} list
    | SomeSelect of {index : int, pass : bool, msg : string} list
    | Input of string
    | SomeInput of string list

    val parse : Lexer.token list -> qtype list
  end
```

qtype 型は，問題または文章を表す型である．Statement は学習者にそのまま表示する文章，Split は文章の区切り，Select は選択問題の各選択肢，SomeSelect は複数選択問題の各選択肢，Input は文字列を答える問題の解答，SomeInput は複数の文字列を答える問題の解答を表す．

以下に，sample.txt の字句解析結果を構文解析器 Parser で構文解析を行った結果を示す．

```
# Parser.parse (Lexer.fileLexer "sample.txt");
val it =
  [
    Statement "問題です．\n",
    Statement "(1) 正解\n",
    Statement "(2) 不正解\n",
    Statement "(3) 不正解\n",
    Statement "\n",
    Select
    [
      {index = 1, msg = "(1) は正解です．\n", pass = true},
      {index = 2, msg = "(2) は不正解です．\n", pass = false},
      {index = 3, msg = "(3) は不正解です．\n", pass = false}
    ]
  ]
  : Parser.qtype list
```

Lexer.Statement はそのまま Parser.Statement として変換され，sample.txt 中の，<select begin> から，それに対応する<end>までの各選択肢が Parser.Select として保持されている．

4.3 ステージ作成

4.3.1 ステージの作成方法

DebuQ は，事前に作成された準備ファイル群を読み込むことで対話的な学習を実現している．準備ファイル群は，ステージファイル，ステージリストからなる．ファイル名が qlist のファイルをステージリスト，ファイル名が qn(ただし n はステージ番号) のファイルをステージファイルとする．付録 B にステージ 1 のステージファイル q1 を示してある．

ステージリストでは，各ステージのステージ名を定義する．n 行目の文字列をステージ番号 n のステージ名とする．DebuQ におけるステージリストは以下の通りである．

ステージファイルは，各ステージについて作成する必要がある．ステージファイルの各行には，Statement か Command のどちらかを書く．

- Command：”<”と”>”で囲んだ文字列．以下の種類がある．
  - `<begin select>`
    `<end>`までの各 answer についての問題である．
    true の選択肢を答えたら終了する．
  - `<begin someselect>`
    `<end>`までの各 answer についての問題である．
    選択肢から 1 つずつ選び，正解を全て答えたら終了する．
  - `<begin input>`
    文字列を入力する問題である．
    `<end>`までの文字列と同じものを答えたら終了する．
  - `<begin someinput>`
    空白区切りの複数の文字列を入力する問題である．
    正解と同じ入力を行ったら終了する．
    順番は合っていなくてもよい．
  - `<begin answer (int) (bool)>`
    問題の選択肢を表す．`<end>`までの文字列を問題回答時のメッセージとする．
    `(int)` と `(bool)` にはそれぞれ数字，真偽値が入る．
    例えば`<begin answer 1 false>`, `<begin answer 2 true>`などとなる．
  - `<end>`
    各 begin 命令の終了を表す．
- Statement：Command 以外の文字列

以下に，ファイル構成 debuq，qlist，q1 からなる場合の動作を示す．ただし，debuq は DebuQ の実行ファイル，qlist は文字列”サンプル問題”一行のみからなるファイル，q1 は 4.2 における sample.txt と内容が同じファイルとする．

```
$ ./debuq
1．サンプル問題
タスクを選択してください (q:終了) => 1
問題です.
(1) 正解
(2) 不正解
(3) 不正解

2
(2) は不正解です.
3
(3) は不正解です.
1
```

(1) は正解です.
Congratulations!

4.3.2 本研究で作成されたステージ

作成したステージと，そのステージをクリアすることで獲得が期待されるデバッグスキルを表 4.1 に示す.

ステージ 1 では，単純なバグの例である型エラーを題材にし，デバッグの基礎を学ぶ．学習者はコンパイルエラーの出力からプログラム中のエラー箇所を推定し，選択問題によって修正を行う．このステージでは，学習者に学習内容を正確に理解させることより，問題に取り組む姿勢を持たせることを優先させている．また，間違った選択肢を選んだとしても，ゲーミフィケーション技術の即時フィードバックに則り，学習者が選んだ選択肢が間違いである理由とどのように選択肢を選べばよいかを表示する.

ステージ 2 では，デバッグに限らず多くの問題解決の際に有効な考え方である科学的手法に則ったデバッグの方法を学ぶ．この問題は文献 [1] の 6 章「科学的デバッグ」の記述を参考に作成した．このステージでは，障害が発生した際に仮説を立て，それが正しいかそうでないかを検証する能力を身につける.

ステージ 3 では，ステージ 2 で学習した科学的手法に加えて，変数の依存関係からバグを探し出す方法について学ぶ．このステージでは，3.3 で述べた制御フローの考え方を，SML# ではどのように利用するかを提示する.

ステージ 4 では，ログの出力を用いてより実践的な仮説の検証を行う方法について学ぶ．ログの出力については 3.4 で述べた通りである．このステージでは，繰り返し入力を受け付け計算処理を行う再帰関数を定義し，ログを出力することでプログラム中の欠陥の特定を試みる.

表 4.1: ステージ一覧

ステージ番号	ステージ名	デバッグスキル
1	型エラー	エラー文章の読み取り
2	仮説を立てる	仮説の立て方と検証
3	変数の依存関係	各変数の依存関係の確認
4	変数の値の確認	ログの出力

表 4.1 におけるステージは難易度順となっている．ステージ 1 が最も簡単で，ステージ 4 が最も難しい．デバッグの難易度は一概に決めることは出来ないが，3.1 で述べた通り，バグは欠陥，感染，障害の順に波及していく．よって，確認された事実がソースコードの欠陥から遠いほどデバッグが難しいと考えられる．ステージ 1 では対話環境におけるエラーメッセージから直接欠陥箇所を推定できるが，ステージ 2 以降では実行結果誤りを扱っているため，障害から感染を発見し，感染から欠陥を探す必要がある．さらに，ステージ 2 では短いプログラムを先頭から読み進めるが，ステージ 3 ではデータ依存，ステージ 4 では制御依存を考える必要がある.

第5章　関連研究と議論

5.1　関連研究

5.1.1　学習支援システム

デバッグ学習を支援する試みとしては，文献 [4] における授業パッケージと学習支援システムが存在する．これは，場当たり的なデバッグを行う学習者の学習姿勢を問題であるととらえ，体系的なデバッグ学習を支援するシステムを提供している．プログラミング言語は Java を対象とし，実装誤りの中でも実行結果誤りを引き起こすバグを対象としている．これは，コンパイルエラー等のエラーメッセージが得られず，バグを特定するのが困難であるためである．また，バグの発見と修正の 2 つの作業のうち，バグの発見を学習対象としている．学習する体系的デバッグ手順は，複数の関数からバグを発見するものであり，プログラムの構造図やデータフローを作成することにより，段階的にバグを絞り込んでいくものとなっている．

5.1.2　シリアスゲーム

デバッグ学習をテーマにしたシリアスゲームとしては，RoboBUG [5] が存在する．RoboBUG は，コンピュータサイエンス 1 年生のデバッグ学習の導入として開発されたシリアスゲームである．C++ のデバッグを想定しているが，必要に応じて他の言語のデバッグ学習にも対応できるフレームワークが用意されている．RoboBUG では，以下の 4 つのスキルを習得できる．

- Code tracing
  コードを読み，適切に動作するかを確認する．
- Print statements
  プログラム内部に状態を出力するコードを挿入する．
- Divide-and-conquer
  ソースコードを分割し，バグを独立させる．
- Breakpoints
  任意のタイミングでプログラムを中断する．

また，RoboBUG にはゲーミフィケーション技術におけるアンロックが使用されている．プレイヤーは条件を満たすと新しいツールを使用することができるようになっている．

5.1.3 本研究との相違点

どちらも，直感的に行われていた従来のデバッグ学習を改善する取り組みであると言える．

文献 [4] では主に学習支援システムを作成しているが，ゲーミフィケーション技術を積極的に取り入れているわけではない．無論，無理に学習をゲーム化する必要はない．文献 [4] は，授業パッケージとしての学習支援システムを作成するのであれば，ゲーミフィケーション技術を使用しなくとも十分な効果が確認された例である．ゲーミフィケーションは学習者のモチベーションを保つ効果があるが，学習内容がより専門的になった場合には，ゲーム要素が学習の妨げになる可能性がある．例えば，ゲーミフィケーション技術の目標設定は，具体的な学習内容を提示することで実現が可能である．しかしこれは学習の自由度が下がることも意味しており，より柔軟で効率的な学習には従来の非ゲーム的な方法が適当である．

文献 [5] では主にシリアスゲーム，つまりゲームを作成している．ゲーム化することのメリットは，学習者が意欲的にデバッグの学習を行うことができるようになる点である．しかし，前述したようにゲーム要素を取り入れるほど，より高度な内容の学習は困難となる．実際に，シリアスゲームである RoboBUG はデバッグ学習の導入として開発されており，その後に改めてデバッグについての学習を行う必要がある．

本研究で開発した DebuQ は，これら 2 つの例の間の立場を取る．つまり，あくまで開発対象は学習環境システムである点が文献 [5] における RoboBUG とは異なり，ゲーミフィケーション技術を積極的に取り入れている点で文献 [4] の研究とは異なる．

5.2 議論

ここでは，DebuQ の今後の発展と実用化及び，評価方法に関する考察を行う．

DebuQ を導入する利点として，対話的な説明や問題の作成をテキストベースで行うことができる点がある．しかし，現在は独自の記述方法に従って作成する必要がある．よって，説明や問題の作成が XML や JSON といった一般的な形式に対応すれば，より導入障壁が軽減すると考えられる．

他には，学習者が使用するインターフェイスの改良が挙げられる．DebuQ は，ゲーミフィケーション技術としてアンロックや即時フィードバックを取り入れているが，その他の技術は採用できていないのが現状である．よって，他のゲーミフィケーション技術を使用することにより，より意欲的な学習が期待できる．DebuQ はコンソールアプリケーションとして開発を行ったが，グラフィック技術に対応することでより多様なゲーミフィケーション技術を取り入れることができると考えられる．例えば，ステージクリア時のグラフィックによる演出を追加することが，ゲーミフィケーション技術の称賛演出を取り入れる方法として最も単純で効果的である．また，グラフィックによる操作の方が直感的であり，CLI よりも GUI を使用する機会が多い学習者にとっては親しみやすいという利点もある．

内容に関しては，現状の DebuQ はデバッグ学習の導入という側面が強いという課題が存在する．実践的なデバッグ能力を獲得するためには，実際に自分の手によりログの出力やステートメントの依存関係を追跡することが効果的であると考えられる．したがって，現状の DebuQ の問題の後に，より実践的な課題に取り組む機会を設けることが望ましいと考えられる．この要素の実現に関する技術的な問題としては，学習者が作成したプログラムを解釈，実行する機構が必要となることである．そのためには，例えば既存のコンパイラとの連携を実現する必要があると考えられる．

本研究では研究期間の都合上，評価実験を行うことができなかった．評価実験を行う場合は，文献 [4] のように，プログラミングの基本的な文法を覚えた段階の学習者に DebuQ によるデバッグの学習

を行わせ，デバッグ学習の実施前と実施後での試験の成績を確認することで学習効果の評価が行えると考えられる．試験の内容は，バグが存在するプログラムの修正を行わせるものが考えられる．DebuQ 内で出題される問題と同様のものが考えられるが，問題に取り組む前の説明文章及び不正解時のコメントを示さない形式で行う．DebuQ による学習前よりも学習後の方が全体の成績が向上した場合は，DebuQ がデバッグ学習に効果的であると考えられる．

第6章　結論

デバッグ学習の促進への第一歩として，ゲーミフィケーション要素を取り入れたデバッグ学習環境 DebuQ を開発した．DebuQ はゲーミフィケーション技術「即時フィードバック」を実現するため対話的な学習を実現し，ゲーミフィケーション技術「アンロック」を用いた段階的な学習順序設計となっている．学習者は，DebuQ を用いることで，デバッグ技術として「エラー出力の読み取り」「科学的手法」「ステートメントの依存関係」の 3 つを学習できる．

しかし，採用できたゲーミフィケーション技術は主にこの 2 つのみであり，実用化に向けてはユーザーインターフェースの改良，学習者が自分の手でログ出力等の作業体験ができる環境への対応が必要である．

謝辞

本論文の作成にあたり，指導してくださった大堀淳教授，上野雄大准教授に感謝申し上げます．また，研究方針の決定や本論文の推敲の際に数多くの的確な助言をいただきました菊池健太郎助教に深く感謝申し上げます．並びに，研究生活を支えてくださった大堀・上野研究室の皆様に感謝いたします．

付 録A　DebuQの実装

A.1　main.sml

A.2　stage.sml

付 録B　ステージ1のステージファイルq1

関連図書

[1] Andreas Zeller 著, 中田秀基監訳, 今田昌宏, 大岩尚宏, 竹田香苗, 宮原久美子, 宗形紗織訳. デバッグの理論と実践 -なぜプログラムはうまく動かないのか. オライリージャパン, 2012.

[2] 井上明人. ゲーミフィケーションー＜ゲーム＞がビジネスを変える. NHK 出版, 2012.

[3] リングフィットアドベンチャー. https://www.nintendo.co.jp/ring/.

[4] 山本頼弥, 野口靖浩, 小暮悟, 山下浩一, 小西達裕, 伊東幸宏. 場当たり的なデバッグを行ってしまう学習者に体系的デバッグ手順を指導する授業パッケージと学習支援システムの構築. 教育システム情報学会誌 Vol.35,No.1 2018 pp.21-37, 2018.

[5] Jeremy S. Bradbury Michael A. Miljanovic. Robobug: A serious game for learning debugging techniques. In *the 2017 ACM Conference*, 2017.

[6] 井上明人, 木村知宏, 福田一史, マーティン・ロート, 松永伸司. ゲーム研究の手引き. 文化庁, 2017.

[7] 岸本好弘, 三上浩司. ゲーミフィケーションを活用した大学教育の可能性について. 日本デジタルゲーム学会 2012 年次大会発表原稿, pp. 91–96, 2012.

[8] #denkimeter. http://www.denkimeter.com/, 2020.

[9] あすけん. https://www.asken.jp/, 2020.

[10] Progate. https://prog-8.com/.

[11] Atcoder. https://atcoder.jp/.