ソフトウェア設計および実験
ソフトウエアテストとデバッグ

森田 和宏

2019 年 6 月 11 日

1　実験内容

1.1　目的

ソフトウェア開発において，その品質を保証するために実施するテストは重要な役割を担う．本実験ではソフトウェアテストの役割を学習し，具体的なテスト手法，不具合修正のためのデバッグ技術の習得を目的とする．

1.2　説明内容

以下の項目について説明する．

- ソフトウェア開発とテストについて
- 具体的なテスト手法について
- デバッグ手法について

1.3　課題

与えられたプログラムに関するテストを実施し，不具合箇所をデバッグする．

2　ソフトウェアの開発とテスト

2.1　テストの重要性

我々は普段から何気なく身の回りにある製品を使用しているが，このときに欠陥があることを想定して利用している人はいないであろう．例えば自動車はアクセルを踏めば進むし，ブレーキを踏めば止まるのが当たり前であり，止まらなければ損害を被るだけでなく，その自動車を製造した企業にとっても大きな損失となることはすぐに予想できる．このため，製造する立場から欠陥はあってはならないものであり，市場に出す前に十分な品質検査，すなわちテストをおこなっている．

ソフトウェアにおいても例外ではなく，無形ではあるが製品となる以上十分な品質を要求されるのは有形の一般製品と変わりがない．

一方，バグの無いソフトウェアは存在しない，といわれるほど最初から完全なソフトウェアを開発すること

は困難であり[*1]，ソフトウェアにおいてもテストを実施することで不具合を最小限に抑える努力が必要なのである．

2.2 ソフトウェア開発の流れ

図 1 は，ソフトウェア開発における一般的な工程を表している．一般に **V 字モデル**と呼ばれ，各開発工程に対応するテスト工程が存在することがわかる．以下に各項目を簡単に説明する．

要求定義
: ソフトウェア利用者が求める要望を洗い出し，整理する作業で，これから作るべきソフトウェアの達成目標が明らかになる．

基本設計
: 要求定義における要求項目を実現するために必要なソフトウェアの機能・構成など基本的な仕様をまとめる作業．要件定義とも呼ばれる．

詳細設計
: 基本設計における仕様を元に，コーディングに必要な各処理の詳細仕様を決定する作業．

コーディング
: 詳細仕様を元にソフトウェアを作成する作業．実際にソースコードと向き合うのはこの工程からとなる．

単体テスト
: ソフトウェアの小単位 (関数など各パーツ) ごとにおこなうテスト．主に詳細仕様どおりに動作するかを確認する．

統合・機能テスト
: 各パーツを組み合わせて実施するテスト．基本設計における機能を満たすか確認する．

システムテスト
: 要求定義の内容が実現されているかを確認するテスト．

2.3 テストの分類

2.2 節で述べた工程を基準にした分類の他にも，ソフトウェアテストは様々な観点から分類ができる．ここでは代表的なものを紹介しておく．

ホワイトボックステスト
: ソフトウェアの内部構造に注目して[*2]，分岐命令，データ構造などが正しく動作するかを確認するテスト．ソースコードレベルでのテストとなり，主に単体テストが分類される．

ブラックボックステスト
: 内部構造は参照せず[*3]，入力と出力のみに注目して，正しく動作するかを確認するテスト．機能レベルでのテストになるので，統合テストやシステムテストが分類される．

[*1] Windows などが毎月更新プログラムを提供していることからも困難さがわかるというものである．
[*2] ブラックボックスに対することから，ホワイトボックスと呼ばれる．
[*3] 内容を見ない (中身が見えない) ことから，黒い箱になぞらえてブラックボックスと呼ばれる．

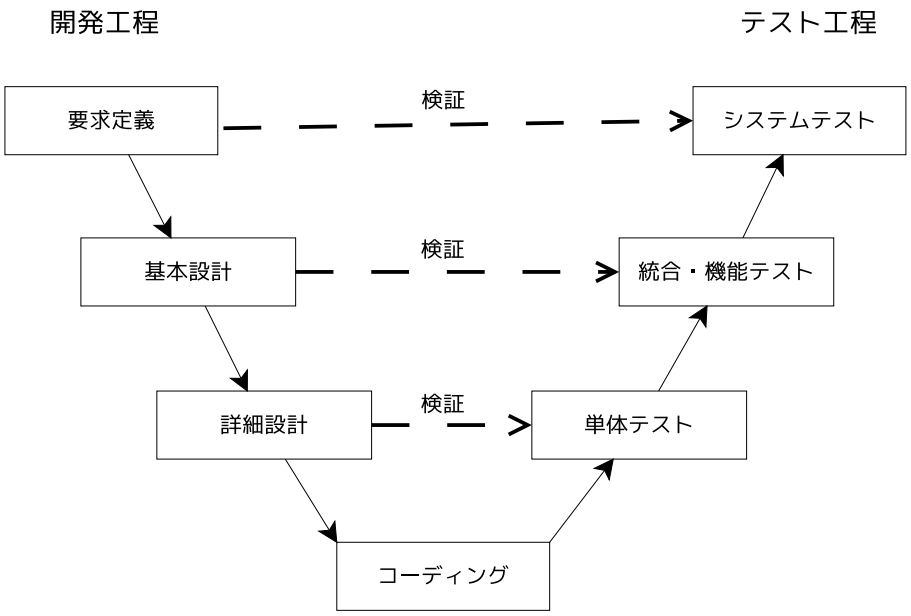


図 1: ソフトウェア開発工程 (V 字モデル)

3　ソフトウェアテスト手法

本来ならば，ソフトウェアのテストは起こりうる全ての可能性を網羅的に確認しなければならない．しかし実際は開発コストとのトレードオフでテスト内容が決められる．その際には低コストでもより網羅性の高いテストが実施されるべきである．

本節では，具体的なテスト手法について説明する．いずれの手法も**テスト項目 (テストケース)** を洗い出すことで網羅性を高めるものであり，手法の違いはその観点の違いとなっている．

3.1　制御フローテスト

プログラムの処理の流れ，順序を確認するテストで，ホワイトボックステストに分類される．手順は以下のとおり．

1. ソースコードを元にフローチャートを描く
2. 処理経路を抽出する
3. 抽出した経路を通るようにテストする

例えば図 2 のフローチャートには，分岐が 2 カ所あるので，全ての経路を辿ると 4 通りになる．

3.2　状態遷移テスト

ソフトウェアの動作中に様々に変化する状態に着目したテストで，ブラックボックステストに分類される．このテストでは，図 3 のような状態遷移図を作成する．状態遷移図には，丸で囲まれた**状態**と，状態同士をつなぐ矢印である**遷移**，(矢印に付随した) 遷移を起こす**イベント**が記載されている．

この図に対し，

- 全ての状態を 1 回は通る

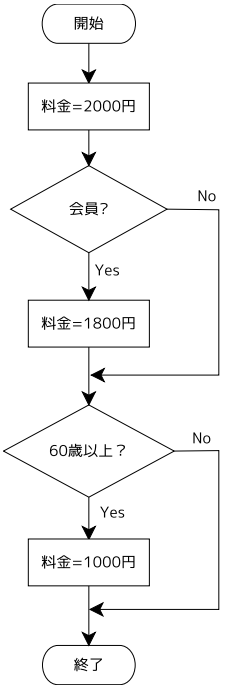


図 2: フローチャート (割引料金計算) の例

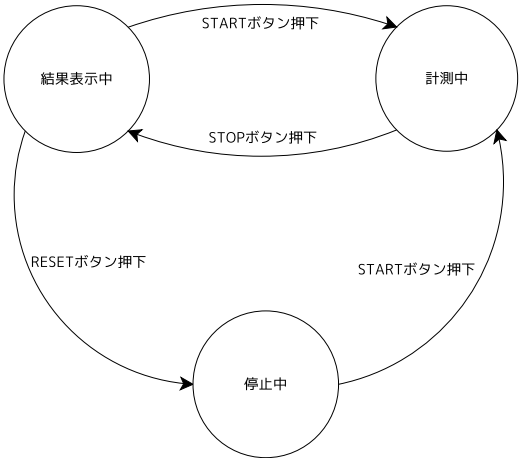


図 3: 状態遷移図 (ストップウォッチ) の例

- 全てのイベントを 1 回は発生させる
- 全ての遷移を 1 回は通る

ようにテストをおこなう.

3.3 同値クラス・境界値テスト

入力される値の条件を確認するテストで，ブラックボックステストに分類される．例えば，パスワード入力画面では入力できる文字に制限があることが多いが，全ての文字パターンを 1 文字ごとに全て確認することは不可能である．そこで以下を確認する.

同一条件となる入力パターン

パスワードに使用できる文字だけの文字列や，範囲外 (短すぎる) 文字数の文字列など.

条件の境界値となる入力パターン

文字数制限が 10 文字以下であれば，10 文字と，11 文字の文字列.

これらをプログラムする際には，条件の境界をチェックするコードを記述するのが普通であるため[*4]，境界付近をテストすれば不具合の発見率が高まることになる.

3.4 デシジョンテーブルテスト

複数の条件によって決定されるソフトウェアの動作を確認するテストで，ブラックボックステストに分類される.

[*4] 上記の例では，11 文字入力されれば再度入力を促したり，エラーメッセージを表示するなどの対処によって，プログラムが正常に動作するようにコーディングされているはずであり，異常終了することはない．諸君らも自身のプログラムをコーディングする際には気を配ろう.

表 1: デシジョンテーブル (レンタル料金割引) の例

		ルール (Y:Yes/n:No)					
	ルール No.	1	2	3	4	5	6
条件	旧作	Y	n	n	Y	n	n
	準新作	n	Y	n	n	Y	n
	新作	n	n	Y	n	n	Y
	まとめてレンタル	Y	Y	Y	n	n	n
アクション	60% オフ	Y	Y	n	n	n	n
	40% オフ	n	n	Y	Y	n	n
	20% オフ	n	n	n	n	Y	n
	通常料金	n	n	n	n	n	Y

このテストでは，表 1 のようなデシジョンテーブルと呼ばれる，複数の条件に着目した一覧表を作成する．デシジョンテーブルには，複数の**条件**と，それらに伴う**アクション**を列挙し，各条件の Yes，No によってどのアクションが起こるのかをルールとして記載している．

この全てのルールを網羅するようにテストする．従って，表 1 では 6 通りのテストを実施することになる．

3.5 テストドキュメントの作成

通常ソフトウェアの開発がたった一人でおこなわれることはない[*5]．各開発工程ごとに責任者や作業者が存在して，協調して開発が進むのが一般的である．このとき，各作業者間の情報伝達に用いられるのがドキュメントである．特にテストドキュメントには，テストケースやテスト結果，不具合 (修正) 報告などが含まれるので，ソフトウェアの品質状況を可視化できる利点がある．

ドキュメントの種類は多岐にわたるが，ここでは以下のテストドキュメントについてのみ説明する．

テストケース　テストを実施する際に必要となる情報を記載するドキュメントで，前述のテスト手法を用いて洗い出されたテスト項目を列挙し，まとめたものとなる．主に次の項目が記述される．

- 事前条件
- 具体的な入力値
- テスト手順
- 期待 (想定) する結果
- テスト実施結果欄

テストログ　テストケースを元にテストを実施し，実施結果欄に結果 (○/× など) を記載したドキュメントがテストログとなる．表 2 に例を示す．

不具合報告書　テストによって検出された不具合，つまりテストログに × が記載された項目をピックアップしたドキュメントで，主に次の項目が記載される．

- 不具合の概要
- (不具合を検出した) テストケース
- 再現率
- 重要度

[*5] 最近はスマホアプリの台頭で個人開発されるアプリも増えてはいるが．

表 2: テストケース/テストログの例

No.	手順	期待する結果	判定 (○/×)	不具合 No.	備考
1	マップ情報ファイルが存在しない状態でプログラムを起動する	エラーメッセージを表示して終了する	○		
2	マップ情報ファイルの内容が不足した状態でプログラムを起動する	エラーメッセージを表示して終了する	○		
3	内容が正しく記述されたマップ情報ファイルが存在する状態でプログラムを起動する	ゲーム画面が表示される	○		
4	起動直後のメインウィンドウを確認	・マップ，キャラクターが表示されている・スコアが表示されている	×	1	
5	プレイヤーの動作を確認	路を外れると落ちる	×	2	
:	:	:	:	:	:

表 3: 不具合報告書の例

No.	タイトル	テスト No.	再現率	原因	修正内容
1	起動時にマップが表示されない	4	毎回	マップ背景を転送する際に座標が合っていなかった	座標の計算方法を見直した
2	プレイヤーが路を外れても動く	5	数回に一度	路と重なっているかチェックしていなかった	チェックするコードを追加
:	:	:	:	:	:

- 不具合の原因
- どのように修正したか

“不具合の原因” と “どのように修正したか” は通常不具合修正 (デバッグ) 後に記載される．表 3 に例を示す．

4　デバッグ技法

テストを実施することによって効率的に不具合の検出ができるようになるが，検出された不具合が，具体的にどの部分に問題があるのかを発見できなければ，修正することができない．従って，不具合修正，すなわちデバッグにおいて最も重要なことは，いかに効率よく問題の箇所を発見できるか，ということになる．本節では，効率よくデバッグするための技術について述べる．

4.1 デバッグの基本

デバッグ (debug) とは，プログラム中の不具合 (bug) を取り除くことである．そのために，まずはソースコード上のどの部分で不具合が発生しているのかを発見する必要がある．

不具合の発見に必要な唯一の作業は，ソースコードを順番に「実際に正しい」動作をしているか，確認していくことだけである (**確認の原則**)．しかしこの確認は，大変根気の要る作業となるので，できる限り無駄を省いて効率的におこないたい．そこで，効率的なデバッグ作業の指針を列挙する．

デバッガ (debugger) を使う
: 古典的には，printf() などを使って実行時に変数の値を表示させるコードを埋め込む手法がとられていたが，頻繁にコードの書き換えが発生したり，#ifdef などマクロで制御する方法はコードの見栄えが悪くなるので好ましくない．
デバッガは実行中のプログラムの状態を制御・確認するツールであるので，変数の値の確認も容易になる．また，任意の場所で一時停止でできたり，変数の値の変化を検出したりもできる．

トップダウンに調べる
: 通常プログラムは大きな処理をおこなう関数から徐々に細かな処理をおこなう関数に移っていく．これに従って，まずは大きな関数のどれが正しく動作しないか調べ，次にその関数の内部を同様に調べる，といった手順をとることで，正常動作の関数をスキップできる．

小さく始める
: いきなり複雑な条件を調査するのではなく，まずは単純な条件で調査を始めても意外とバグは見つかるものである．

二分探索を用いる
: 例えば 1000 回くらいの繰り返しの何回目でバグが起こるかわからないときに，ループを最初から 1 回ずつ調べるのは非効率である．500 回目まで動かしてみて，正しければ次は 750 回目まで，ダメであれば 250 回目まで，というように，二分探索的に回数を変えていけば効率的である．

4.2 gdb の操作方法

gdb とは代表的なデバッガプログラムである．gdb を利用するためには，対象のプログラムにデバッグ情報を埋め込む必要があり，gcc の場合は `-g` オプションである．以下に gdb を用いたデバッガの使い方を簡単に説明する．

4.2.1 起動と終了，実行と中断

コマンドライン上で，

```
$ gdb デバッグする実行プログラム
```

とすれば，gdb が起動し，次のような専用プロンプトが表示される．

```
(gdb)
```

このプロンプト上で各種の gdb コマンドを入力すればよい．デバッグ対象のプログラムを実行するときは，

```
(gdb) run
```

とする．よく使われるコマンドには省略形が存在し，run の場合は r である．以後，(r)un と表記する．

実行中のプログラムを中断するときは，gdb のプロンプト上で C-c (Ctrl + C キー) をタイプする．gdb を使わない通常の実行とは違い，プログラムが一時停止 (中断) されているだけなので，再開することもできる．また，(l)ist で，中断箇所周辺のソースコードを表示できる．

gdb 自体を終了させるコマンドは，(q)uit である．しかし gdb は実行プログラムの変更を検出できる．つまり，gdb を起動させたまま，実行プログラムを修正し，コンパイルや make し直すと，gdb は新しくなった実行プログラムを再読込する．gdb を一旦終了させてしまうと，設定したブレークポイントなどの情報も消えてしまうので，デバッグ作業中は gdb を終了しない方がよい．

4.2.2 ブレークポイント

ブレークポイントを設定すれば，プログラムの実行がソースコード上のその場所に到達した時点で，自動的に中断する．コマンドは (b)reak に続けて関数名や行番号を指定する．例えば，

```
(gdb) b main
(gdb) r
```

とすれば，プログラム実行後すぐに，main 関数の最初の処理の場所で中断する．

ブレークポイントはいくつでも設定できる．現在設定しているブレークポイントの確認は (i)nfo (b)reak であり，例えば，

```
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048b0d in main at shoot.c:17
2       breakpoint     keep y   0x08049c0e in InitWindow at window.c:39
(gdb)
```

のように表示される．最左の数字がブレークポイントの識別番号であり，この識別番号を使って，ブレークポイントの削除 ((d)elete 識別番号)，ブレークポイントを設定したまま実行時に中断しなくなる無効化 ((dis)able 識別番号)，その逆の有効化 ((en)able 識別番号) を設定できる．

ブレークポイントは，バグのありそうな場所，変数の値を確認したい場所に仕掛けるようにする．

4.2.3 ステップ実行と実行再開

中断中のプログラムを，ソースコード数行分だけ進めたいとき，いちいちブレークポイントを設定するのは面倒である．このような場合はステップ実行ができる．以下の種類がある．

(n)ext：プログラムを 1 行進める．

(s)tep：プログラムを 1 行進めるが，その際に関数呼び出しがあれば，その関数へ移動する．

ステップ実行は何度も繰り返すことが多いが，そのたびに next と入力するのは手間である．幸い gdb には Enter のみの入力で，直前のコマンドを繰り返すことができるので，この機能を利用するとよい．

中断場所の調査が終り，実行を先に進め (再開し) たいときは，(c)ontinue を利用する．また，今中断している関数だけ終了させ，関数の呼び出し元に戻りたいときは，(fin)ish を利用する．

4.2.4 変数の調査とウォッチポイント

プログラムを中断して，次にするべきことは，変数の値などを調査することである．(p)rint に続いて変数名などを指定すると，その値が表示される．変数以外にも，おおよそC言語が解釈できる**式**が利用できる．便利なのは，構造体のメンバをまとめて表示できることである．例えば，

```
(gdb) p gChara[0]
$2 = {type = CT_Player, pos = {x = 240, y = 1289, w = 48, h = 48}, point = {
    x = 240, y = 1289}, base = {x = 0, y = 0}, dist = 1.5, stts = CS_Enable,
  hp = 4, anipat = 1, anipatnum = 4}
(gdb)
```

のように表示される．

変数の値を調査していると，仮に別の値が入っていた場合にどのように動作するのかを調べたくなる場面に出くわすことであろう．gdb は変数に値をセットすることもできる．以下の例のように，set に続けて代入式を記述するとよい．

```
(gdb) set gChara[0].pos.x = 10
(gdb) p gChara[0]
$3 = {type = CT_Player, pos = {x = 10, y = 1289, w = 48, h = 48}, point = {
    x = 240, y = 1289}, base = {x = 0, y = 0}, dist = 1.5, stts = CS_Enable,
  hp = 4, anipat = 1, anipatnum = 4}
(gdb)
```

特にグローバル変数などで，変数の値が変化したかどうかを確認したいとき，中断のたびに確認するのは骨が折れる作業である．gdb には変数の値が変化したときに中断する機能も備えている．(wa)tch に続けて調査したい変数名を指定すると，その変数の値が変わった時点で中断する．

4.3 core を伴うデバッグ

諸君らはすでに多くのプログラムを作成してきて，gdb に頼らないまでも，デバッグをこなしてきたことであろう．この時に「セグメントエラー (Segmentation fault)」などといったメッセージとともにプログラムが異常終了 (落ちる) 場面に出くわしたことが少なからずあるのではないだろうか．

セグメントエラーの原因などは別の講義に委ねることとして，このエラーでプログラムが終了すると，エラー発生時の状態を保存した，core というファイルが生成される．core ファイルによって，gdb で実行し直してエラーの場所まで進める必要なく，エラー箇所を特定できる．方法は，

```
$ gdb 実行プログラム core
```

と，起動時に core を付け足すだけである．これで，エラーの場所で中断しているような状態になっているので，変数の値などを確認すればよい．

また，プログラムのどの場所で中断しているかわかりにくい場合には，関数呼び出し履歴 ((back)trace) や，list を表示するとよい．関数呼び出し履歴の#に続く数字は，フレーム番号である．この番号を指定することで ((f)rame フレーム番号)，呼び出し元のソースを確認できる．

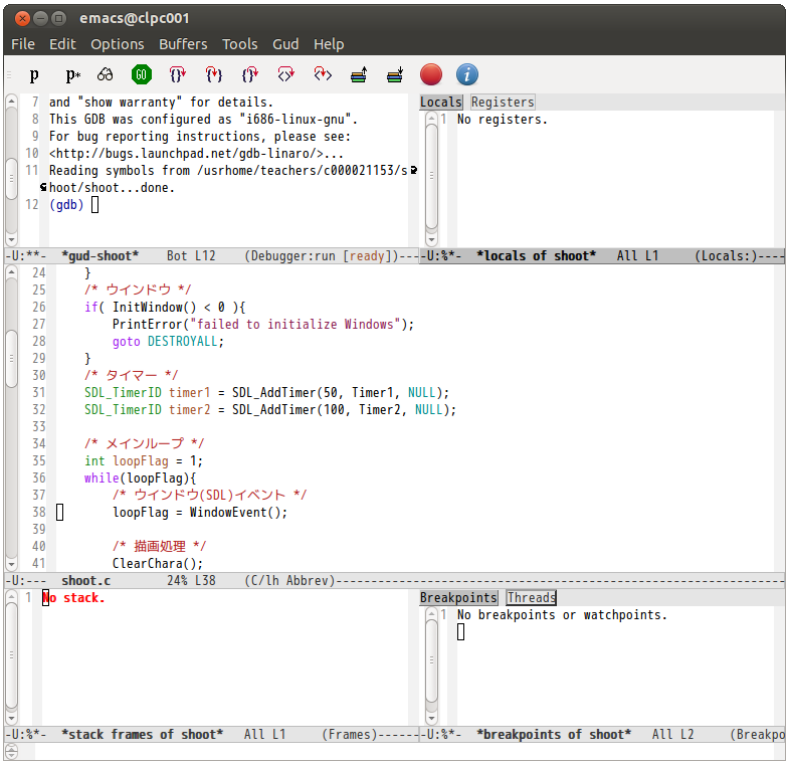


図 4: emacs による gdb 実行例

4.4 エディタとの連携

デバッグは確認作業の連続であるため，一度に確認できる情報量が多いほど効率よく作業が行える．実際市販のデバッガなどは GUI を備え，複数のウインドウに多くの情報を表示できるようになっている．一方 gdb は非常に高機能で優れたデバッガではあるが，コマンドラインツールであるため，情報の確認や動作にいちいちコマンド入力が必要であり，煩わしい面もある．

そこでこれらを解決するような，gdb を GUI 的に扱うことができるフロントエンドツールがいくつも公開されている．ここでは emacs と Visual Studio Code との連携について紹介する．

4.4.1 emacs との連携

まずは次の設定を `~/.emacs` に記述する

```
(setq gdb-many-windows t)
(add-hook 'gdb-mode-hook '(lambda () (gud-tooltip-mode t)))
```

あとは emacs を起動し[*6]，**M-x gdb** とタイプするだけ[*7]である．するとミニバッファに gdb の起動コマンドが表示されるので，実行プログラム名を確認して実行すると，図 4 に示すような表示となる．gdb の各種操作がツールバーボタンに割り当てられていたり，ソースコードが表示されているウィンドウで，左端をマウスクリックするか **C-x Space** とタイプすることでブレークポイントが設定できたり，中断中にマウスを変数に重ねると，その値がポップアップされたりするので活用してみてほしい．

[*6] -nw ではない起動の方が，より GUI 的に扱える．
[*7] 余談だが，make も emacs から起動できる．C-c c とタイプする．エラーがあるときは，C-c e で該当場所にジャンプする．

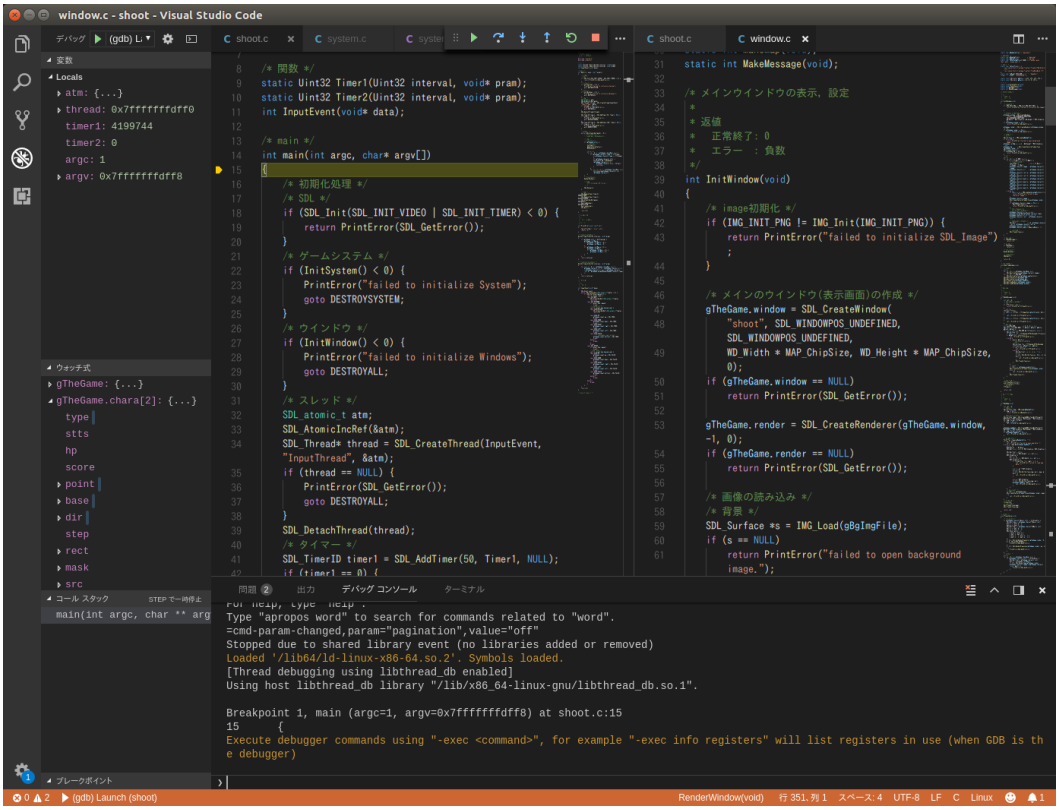


図 5: Visual Studio Code による gdb 実行例

4.4.2 Visual Studio Code との連携

Visual Studio Code は近年主流の Electron 系エディタであり，プログラム開発に特化している．設定や拡張機能によるカスタマイズも豊富で自由度が高い．

Visual Studio Code でデバッグするには，まず拡張機能の C/C++ をインストールする．次に，Linux の場合はデバッグ対象のソースや実行ファイルが存在するディレクトリに，Visual Studio Code の設定ファイルを保存するディレクトリ `.vscode` を作成し，コンパイルの設定ファイル (`tasks.json`) やデバッグのための設定ファイル (`launch.json`) を設定すると利用できるようになる[*8]．emacs と違い，gdb のコマンドを直接操作する必要もなく[*9]，変数の確認なども簡単なので，かなり効率よくデバッグできるのではないだろうか[*10]．

[*8] 課題のファイル内に（今回の課題のための）設定ファイルも含まれているので，確認してみるとよい．

[*9] 全てを網羅してはいないので，必要であれば直接操作できる．デバッグコンソール上で -exec [gdb のコマンド] と入力する．

[*10] `Ctrl + Shift + @` キーでターミナル（シェル）も起動する．また，LaTeX の拡張機能もあるので，このエディタだけで全ての作業をこなすことも可能であろう．

5 課題

5.1 準備

gdb によるデバッグ作業をレポートとして提出してもらうため，gdb 作業をログ出力する次の設定を，`~/.gdbinit` ファイルに記述しておく．

```
set logging on
```

これにより，gdb の出力が `gdb.txt` に追記されていく．

次に，各課題におけるソースプログラム等をまとめた圧縮ファイル game.tar.gz が，徳島大学 manaba にあるので，ダウンロードする．game.tar.gz を展開すると act というディレクトリが作成されるので移動し，make をおこなうと，実行プログラム act が生成される．

この実行プログラムがテスト対象である．概要を以下に示す．また，プログラム全体のフローチャート，各キャラクタの状態遷移図，プレイヤーのデシジョンテーブルをそれぞれ，**図 6，図 7，表 4** に示す．

- ゲームジャンル：横スクロールアクション
- 実行すると，画面中央にプレイヤーキャラが配置される．
- プレイヤーはキーボード十字キーの ←→ で移動，↑でジャンプする．スペースキーで球を投げる．また，ESC キーでプログラムが終了する．
- 画面はプレイヤーの移動する方向にスクロールし，左右の端は繋がっている．
- 下方向に重力があり，プレイヤーや敵キャラなどすべてのキャラクターは，何もないところでは落下する．
- 敵キャラはマップ上のランダムな位置から，プレイヤーとの距離が近い方向にジャンプしながら移動してくる．球に当たると消滅する．
- マップには拠点が配置されており，球を何回か当てると消滅する．
- 敵はマップ上に二体，拠点はマップ上に三カ所，球は画面上に一つしか表示されない．
- 拠点がすべて消滅すると，警告表示の後にスクロールが停止し，ボスが現れる．
- ボスは，ジャンプしながら近づいてくる，プレイヤー方向に投げてくる，を当てる毎に繰り返す．
- ボスを倒すとゲームクリア，敵に当たるか，画面外に落ちるとゲームオーバーとなる．

このプログラム act はバグが満載でまともに動作しないので，以降の課題に取り組むことでプログラムを完成させてほしい．なお，参考のために game.tar.gz には (完全ではないが) デバッグ済みのプログラム act-release も同梱している．

5.2 課題 1

ゲーム概要やフローチャート，状態遷移図などを参照し，このゲームプログラムをテストするためのテストケースを作成せよ．また，テストケースを元にテストを実施した結果を，テストログに記述し，レポートせよ．テストログは表 2 を参考に作成すること．

なお，あるテストケースの不具合によって，別のテストケースのテストができない場合には，下記の課題 2 を平行して実施し，不具合を修正しながらテストをおこなうこと．

5.3 課題 2

テストログを参照し，不具合と判定された項目について，gdb を用いてデバッグせよ．デバッグによって修正した内容を不具合報告書にまとめ，レポートせよ．不具合報告書は表 3 を参考に作成すること．また，gdb の作業ログ gdb.txt も提出すること．

5.4 課題 3

このゲームプログラムは単純であるため，面白さに欠けると思われる．そこで，より楽しくなるような改良を加えよ．改良は，単に画像やマップを変更するだけではなく，ソースコードの追記が必要となるものであること．また，改良に伴って発生するテストケースについてももれなくテストし，課題 1，2 のテストログ，不具合報告書に追記すること．

この課題 3 は仮に取り組まなくてもレポートは受理するが，採点には含まれる．すなわち，課題 1，2 だけでは最大でも 60〜70% の評価点しか得られない．

5.5 レポート提出物

以下のデータを，任意のディレクトリに作成・保存し，圧縮アーカイブにして，期限までに manaba にて提出すること．**未提出は認めない**．たとえ期限までに間に合わなかったとしても，再提出を受ける旨と途中経過の内容を提出すること．

1. **レポート**
   テクニカルライティング，レポートの書き方に従って必要事項をまとめ，LaTeX により作成し，pdf 形式に変換しておくこと．以下の項目を課題の結果をして記載すること．
   - 課題 1 で作成したテストログ
   - 課題 2 でデバッグした不具合報告書
   - 課題 3 をおこなった場合はその改良内容
2. デバッグ後（課題 3 をおこなった場合は改良後）のプログラムソース，画像データなど，make・実行・動作確認に必要なファイル一式（ソースにはコメントをつけること）
3. デバッグログ gdb.txt

5.6 再提出

提出されたレポートの，提出物，内容に不備がある場合，評価点が低い場合には，一度だけ再提出を課す．manaba に**不備内容とともに追加課題を提示する**ので，これらに取り組んだ上で別に定めた再提出期限までに提出すること．なお，採点のポイントは以下の通り．

- テクニカルライティングに従っているか．
- テストケースの網羅性
- デバッグ具合
- gdb の扱い具合
- 課題 3 での改良具合
- オリジナリティ（コピーの可能性があると判定された場合には再提出を課す）

参考文献

[1] 石原一宏, 田中英和： この 1 冊でよくわかる ソフトウェアテストの教科書 品質を決定づけるテスト工程の基本と実践, ソフトバンククリエイティブ (2012).

[2] Norman Matloff, Peter Salzman 著, 相川愛三 訳： 実践 デバッグ技法, O'Reilly Japan (2009).

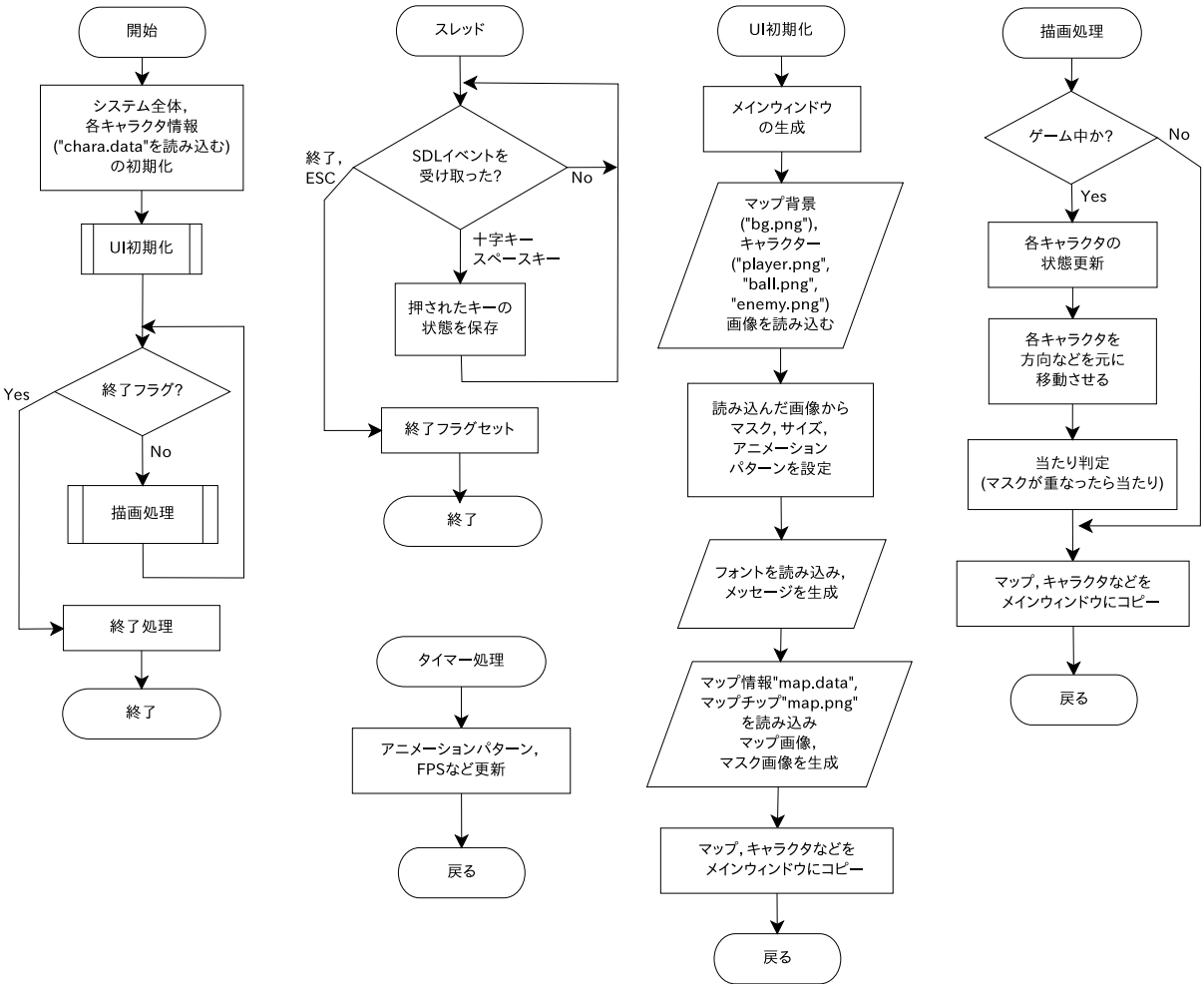
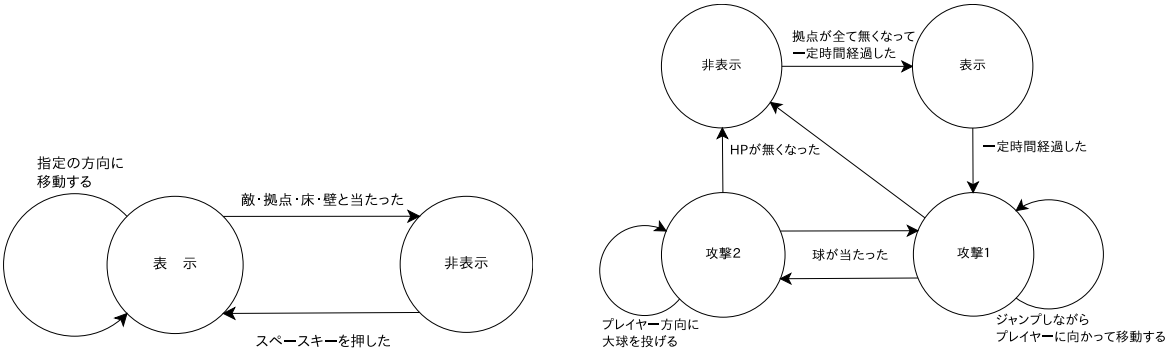


図 6: プログラム全体のフローチャート



(a) 球　　(b) ボス

図 7: 各キャラの状態遷移図

表 4: プレイヤーのデシジョンテーブル

		ルール (Y:Yes/n:No)																
	ルール No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
条件	↑キーが押されている	n	n	Y	Y	n	n	n	n	n	n	Y	Y	n	-	-	-	-
	←キーが押されている	n	n	n	n	Y	n	Y	n	Y	n	Y	n	Y	-	-	-	-
	→キーが押されている	n	n	n	n	n	Y	n	Y	n	Y	n	Y	Y	-	-	-	-
	スペースキーが押されている	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	n	Y	n
	下方が床である	n	Y	n	Y	n	n	n	n	Y	Y	Y	Y	Y	-	-	-	-
	左方が壁である	Y/n	Y/n	Y/n	Y/n	n	n	Y	n	n	n	n	n	Y/n	-	-	-	-
	右方が壁である	Y/n	Y/n	Y/n	Y/n	n	n	n	Y	n	n	n	n	Y/n	-	-	-	-
	画面に球が表示されている	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	Y	n	n
アクション	上方へジャンプする	n	n	n	Y	n	n	n	n	n	n	Y	Y	n	n	n	n	n
	下方へ落下する	Y	n	Y	n	Y	Y	Y	Y	n	n	n	n	n	n	n	n	n
	左方へ移動する	n	n	n	n	Y	n	n	n	Y	n	Y	n	n	n	n	n	n
	右方へ移動する	n	n	n	n	n	Y	n	n	n	Y	n	Y	n	n	n	n	n
	球を投げる	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	Y	n