社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－SE－159 (5)
2008／3／17

ソースコード理解支援機能を持つ開発環境

新倉 諭　　鈴木 正人

北陸先端科学技術大学院大学 情報科学研究科

E-mail: {sniikura, suzuki}@jaist.ac.jp

ソフトウェアの大規模複雑化により、開発者がソースコードの構造や振る舞いに関する概要を比較的短時間で得ること (=理解支援) が重要になっている。特にソフトウェアの修正や変更等の要求に対して、数万行のソースコードの中から対象となる部分を素早く抽出できる機能が求められている。理解支援に関する既存研究はいくつか存在するが、これらの多くは情報量の制御が行われていない、あるいは構造や意味を考慮した絞り込みを行うことができないなどの問題を抱えている。　本稿ではC言語を対象にし、多様な要求に対して必要な情報のみを開発者に提供する理解支援機能をもつツールを開発する。理解支援ツールは細粒度のフィルタによって情報の抽出を行う。このフィルタを組み合わせることで、開発者の必要とする情報を短時間で抽出し、開発保守のコストの抑制が期待できる。

Software Development Environment with Source Code Comprehension Support

Satoshi Niikura　　Masato Suzuki

Japan Advanced Institute of Science and Technology
School of Information Science

E-mail: {sniikura, suzuki}@jaist.ac.jp

Code comprehension is one of the important technique for developers to aquire infomation about source codes such as structures and behaviors. Especially, detection of fragments which have to be changed in order to achieve some modification/extenstion on software from many thousands lines of code within a proper period, are required. Most of traditional tools, however, cannot control the amount of information or cannot extract paticular information which relates to some structures and semantics.

We propose a comprehension tools for C language code, which consists of several fine-grained operations called *filters* and their combinations. Our tool can provide information only developer requires. It is expected that developers can decrease cost of software maintenance.

1　はじめに

近年のソフトウェア開発では、ソースコードの再利用によってその量が肥大化し、また構造も複雑化している。また他人の書いたソースコードに触れる機会も増えてきており、手軽に入手できる他人の書いたソースコードの例としてオープンソースソフトウェアがある。これらは目的に合わせて機能を修正して利用することが可能だが、膨大な量のソースコードをの中から開発者にとって修正を

必要とする箇所を発見するには、プログラム全体の構造や振る舞いに関する知識を必要とする。このような知識を与える作業を計算機によって支援する技術は理解支援と呼ばれている。

本稿の目的は、理解支援の中でも特にソフトウェアの修正や変更等の要求に対して対象となる部分を比較的短時間に抽出して開発者に提示するための手法を確立することである。またそのためのツールを開発、実装することで、保守のコストの低減が期待できる。

2 既存の理解支援ツール

理解支援ツールとして以下の2つがある。

1. GNU GLOBAL[2]
   このツールはソースコードに索引付けを行い、開発者は変数や関数の宣言部や実行部を簡単に発見できる。またその結果をHTMLで出力することで、索引付けをハイパーリンクでたどることができる。このツールによって設定参照関係を把握できる。しかしこのツールは、索引付けの結果を利用して関連する変数を参照できるようにするのみで、ソースコードの構造に関する解析を行わない。そのため目的の変数や関数の宣言部を表示することは可能だが、その情報量を制御することはできない。またツール利用者がそのソースコードを初めて読む場合、ソースコードの注目点を決定する方法がないという問題が残る。
2. Semantic Grep[3]
   このツールは、正規表現に基づくSQLによってテキストファイルをフィルタリングするためのツールである。このツールはgrepと同様にあらゆる種類のテキストファイルから文字列を抽出することが可能であり、またHTMLやC言語のソースコードなどの構造化されたテキストを含むテキストファイルに対して構造を指定した文字列の抽出をすることが可能である。このツールによってソースコードの特定の構造を持った箇所の検索と抽出が可能となる。しかしそのソースコードに初めて読む人にとって、抽出する構造を指定することは可能であるが、抽出した構造と変更する機能との関係を把握するには至らない。またツールへ入力するクエリは構造化されているが、クエリの入力は文字列であるため、絞り込みなどの抽出結果を利用した検索にクエリの再構成が必要となってしまう。

既存の問題点を解決するためのツールの設計方針を以下に示す。

- 構造に着目した抽出を可能にする
- 情報の絞り込みを可能にする
- 試行錯誤の支援を可能にする

これらを実現するため、フィルタによって構造の情報を抽出する。ユーザはツールに定義されたフィルタを利用することで構造を抽出でき、また抽出した構造に対してさらにフィルタを適用することで、絞り込むことができる。このような操作を繰り返すことで、ユーザが抽出した構造と対応する機能との関係を把握するための支援環境を実現している。

3 ツールの構成

本ツールはC言語で書かれたソースコードを対象とする。ユーザは図1に示すように、ツールに定義された数種類のフィルタを利用して情報の抽出を行う。

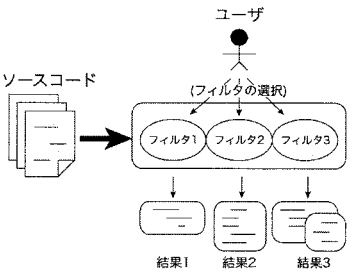

図 1: ツールの構成図

しかしユーザの要求ごとにフィルタを準備しても、その利用は特定の場合に限定される。多様な要求に対応可能なツールを実現するために、フィルタの合成を以下のように定義した。

直列合成 フィルタの出力結果を別のフィルタの入力とする

論理合成 2つのフィルタの出力結果に対してAND/ORによる演算を行う

直列合成によって情報の絞り込みを、論理合成によって複数の抽出結果の選択と統合を可能にする。

前述の合成を可能にするため、7種類の細粒度のフィルタを設計した。フィルタの入力はユーザの指定したソースコードの範囲であり、出力は抽出されたソースコードの範囲である。出力結果の制御をするために、フィルタはパラメタをもつ。例えば、変数名やブロックの入れ子の深さなどがある。フィルタの設計と実装については、4章で詳しく述べる。

ユーザは必要とする情報を抽出するまで、これらのフィルタを単独、あるいは合成して利用する。

4 支援ツールの設計

設計方針を以下に示す。

- 細粒度のフィルタのみ定義
- パラメタによる制御
- 構造, 意味情報に基づいた抽出
- 状況に応じて組合せ可能

4.1 フィルタの入出力

本研究ではフィルタの実現のために抽象構文木(以下AST[1])を用いている。ASTは字句/構文解析の結果から得られる構文木であるが、その各ノードには任意の属性を付加することができる[?]。これを用いてソースコードからの構造の抽出を可能にしている。

ユーザからの入力であるソースコードの範囲は、ツール内部で対応するASTのノードに置き換えられる。実際には入力は範囲であるため、対応するASTのノードのリストとなる。そして組合せを可能にするため、フィルタの出力もまたASTのノードのリストにする必要がある。よってフィルタは、ASTノードのリストから別のASTノードのリストへの関数として実現されている。

フィルタを実装するためのデータ定義を表1に示す。

ここでBlockはユーザからみた範囲を表している。この範囲は、対象のファイルとその開始行および終了行によって構成されている。そしてBlockのリストとしてBlockSetを定義し、ユーザへの入力及び出力となっている。

[1]Abstract Syntax Tree

表 1: フィルタのデータ定義

```
struct AST{
  AST NODE;
  AST *parent;
  AST **child;
  int childNumber;
}

struct BlockSet{
  Block *block;
}

struct Block{
  int fileId;
  int beginLine;
  int endLine;
}
```

4.2 フィルタの種類

以上の設計方針に基づき、以下のようなフィルタを定義する。このうち1.から5.までの5種類は先行研究である永井[1]による提案を拡張、定式化したものであり、残り2種類は[1]において適用実験の結果、必要性が判明したため追加したものである。

1. 制御構造の実行ブロックを抽出するフィルタ(Ctrl)
   このフィルタはプログラムの制御構造の内容を抽出することを目的としている。ここで扱う制御構造は **for, if, while, do, switch** の5種類である(ただし`#ifdef`などのプリプロセッサ命令は考えないものとする)。
   このフィルタに対して以下の式を定義する。
   $\mathcal{F}_{Ctrl}$(AST ast, BlockSet input, int pos, int depth, Descripter descripter) : BlockSet output
   ここでpos, depth, descripterを、このフィルタのパラメタと呼ぶ。このパラメタがフィルタの動作を決定している。各パラメタは次のような値である。
   pos 注目している行の番号
   depth 抽出対象の制御ブロックの入れ子の深

さ

descripter 制御文の記述子の種類

フィルタのアルゴリズム: 注目するノードに対して、制御文のノードを発見するまで親ノードを辿る。発見した制御記述子が descripter と一致した場合、その制御文の実行部に相当するノードを返す。一致しない場合、さらに探索を続ける。このような探索について、depth の数だけ行う。

2. ブロックを宣言部と実行部に分離するフィルタ (Sep)
このフィルタの目的は、ブロック変数の特定を行うことである。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Sep}$(AST ast, BlockSet input, int pos, int depth, Part part) : BlockSet output
各パラメタは次のような値である。

pos 注目している行の番号
depth ブロックの入れ子の深さ
part 抽出対象の選択: 宣言部/実行部

フィルタのアルゴリズム: 注目するノードに対して、ブロックを示す親ノードを depth の数だけ辿る。そのブロックのノードに対して宣言部とそれ以外のノードのリストに分離する。そして part で示したリストを返す。

3. 変数の出現範囲を抽出するフィルタ (Range)
このフィルタの目的は、ある特定の変数についてその変数に関わる計算を全て抜き出すことである。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Range}$(AST ast, BlockSet input, int pos, String var) : BlockSet output
各パラメタは次のような値である。

pos 注目している行の番号
var 変数名

フィルタのアルゴリズム: 入力ノードの中から変数 var に相当するノードに注目する。注目されたノードから親ノードを辿り、その変数の宣言を行っているブロックを発見する。そしてその変数が宣言されたノードから最後に参照されるノードまでの全てのノードをリスト化して返す。

4. 代入文の要素に依存する式を追跡して抽出するフィルタ (Trace)
このフィルタの目的は、ある代入文の右辺の変数についてデータフロー解析を行うことである。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Trace}$(AST ast, BlockSet input, int pos, int depth) : BlockSet output
各パラメタは次のような値である。

pos 注目している行の番号
depth 依存を追跡する深さ

フィルタのアルゴリズム: ある代入文に相当するノードに注目する。右辺にある変数のノードに対して代入を行っているノードを発見する。これを depth の数だけ繰り返し、発見された全てのノードをリストとして返す。

5. 注目部の近傍を抽出するフィルタ (Neibour)
このフィルタの目的は、注目する範囲の物理的近傍行を抽出することで、ユーザがソースコードを読む時の直観的な操作を再現することである。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Neibour}$(AST ast, BlockSet input, int prev, int follow) : BlockSet output
各パラメタは次のような値である。

pos 注目している行の番号
prev 前方物理行数
follow 後方物理行数

フィルタのアルゴリズム: このフィルタは AST の構造を利用していない。しかし AST のノードに行の属性を与えることで、物理的な範囲に相当するノードを抽出可能にしている。

6. 大域変数の定義部と実行部を抽出するフィルタ (Global)
このフィルタの目的は、複数のファイルで参照される大域変数に対して、その影響範囲を特定することである。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Global}$(AST ast, BlockSet input, String var) : BlockSet output
各パラメタは次のような値である。

var 大域変数名

このフィルタの定理には、前処理が必要である。すなわちソースコードをツールに入力したとき、全ての大域変数宣言、型宣言、関数宣言を行うノードを抽出し、表としている。

フィルタのアルゴリズム: 前処理で得た表から var に一致するノードを表より選択して返す。さらにその大域変数を `extern` で呼び出しているファイルについて、その参照を行っているノードを抽出し、返す。

7. 変数および型の定義部を抽出するフィルタ (Decl)
このフィルタは、型の宣言が別ファイルに記述されているときに参照することを目的とする。
このフィルタに対して以下の式を定義する。
$\mathcal{F}_{Decl}$(AST ast, BlockSet input)
: BlockSet output

フィルタのアルゴリズム: 注目するノードの型宣言を行うノードを、前処理によって抽出された表を用いて抽出して返す。

4.3 フィルタの合成

3 章で述べたフィルタの合成 (直列合成/論理合成) について示す。

ツールは上記 7 種類のフィルタの出力結果について、内部で保存を行っている。保存される結果は AST ノードのリストとして保存されているため、フィルタの合成を行う場合、以前のフィルタの出力結果を参照できる。

フィルタへの入力としてツール内部に保存されたフィルタの結果を選択することで、直列合成を実現する。

2 つのフィルタの抽出結果に対して、共通する AST ノードのみを選択したリストを返すことで、AND 合成を実現する。さらに少なくともどちらか一方に存在する AST ノードを全て選択したリストを返すことで、OR 合成を実現する。

これらの設計のもと、実装したツールの画面を図 2 に示す。

5 ツールの評価

提案するツールの有用性を確認するための評価実験を図 3 のように行う。ソフトウェアの 2 つのバージョンを利用して、以下のように変更に必要な箇所の抽出を行えているかの確認と評価を行っている。ここで、新バージョンのリリースノートの変更点を旧バージョンに対する機能変更要求として利用している。

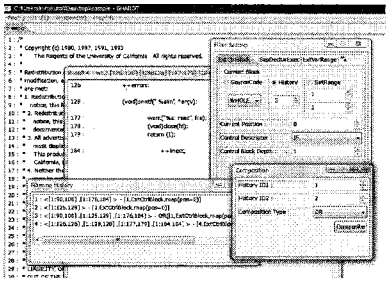

図 2: ツールの実行画面

1. 旧バージョンとリリースノートを用いて、本ツールの適用を行う。
2. 旧バージョンと新バージョンの差分を求める。
3. 1. と 2. の比較を行う。

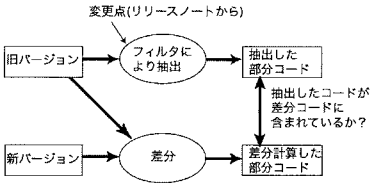

図 3: 実験の流れ

5.1 実験操作

実験に使用した題材を以下に示す。

ソフトウェア名: jless[10]
旧バージョン: 358
新バージョン: 378
リリースノートに記載されている変更点:

- 可変幅 tab 位置の指定
- キーパッド初期化を無効にしない
- キーパッド初期化を無効にする

など十数箇所

今回は「可変幅タブ位置の指定」という変更点を対象に検出実験を行う。その操作と結果について、以下に示す。

1. 大域変数で`"tab"`を含む変数名を探し、最初の手がかりとする。フィルタ (1) を利用し、表 2 が

抽出された。この結果から、大域変数 tabstop とその参照場所を理解することができる。

$\mathcal{F}_{Global}$(AST, WHOLE, "/tab/i")
: ⟨{line.c, 488, 489}, {line.c, 499, 499}, {opttbl.c, 32, 32}, {opttbl.c, 290, 290}⟩ (1)

表 2: フィルタ (1) の抽出結果

line.c

```
488    if (tabstop == 0)
489     tabstop = 1;
499    }while (((column + cshift - lmargin)
                        % tabstop) != 0);
```

opttbl.c

```
32    public int tabstop;
290    NUMBER|REPAINT, 8, &tabstop, NULL,
```

2. 抽出された line.c の 488 行目について、周囲の実行内容を調べるために同一ブロックの抽出を行う。フィルタ (2) を利用し、表 3 が抽出された。この結果が、tabstop を用いてタブをスペースへ変換している箇所であることが読み取れる。

$\mathcal{F}_{Ctrl}$(AST, ⟨{line.c, 0, EOF}⟩, 488, 1, WHOLE)
: ⟨{line.c, 484, 502}⟩ (2)

表 3: フィルタ (2) の抽出結果

line.c

```
484    } else if (c == '\t'){
485     /*
486      * Expand a tab into spaces.
487      */
488     if (tabstop == 0)
 ⋮          ⋮
497     do{
498      STOREC(' ', AT_NORMAL);
499     } while (((column + cshift - lmargin)
                        % tabstop) != 0);
500     break;
501    }
502   }
```

3. フィルタ (1) の抽出結果 opttbl.c の 290 行目についても同様に、周囲の内容を調べるために同一ブロックの抽出を行う。フィルタ (3) を利用し、表 4 が抽出された。$\mathcal{F}_{Ctrl}$(AST, ⟨{opttbl.c, 0, EOF}⟩, 290, 1, WHOLE)
: ⟨{opttbl.c, 289, 294}⟩ (3)

表 4: フィルタ (3) の抽出結果

opttbl.c

```
289    {'x', &x_optname,
290      NUMBER|REPAINT, 8, &tabstop, NULL,
291      "Tab stops :",
292      "Tab stops every %d spaces",
293      NULL
294    }
```

4. フィルタ (3) の結果から、さらに広い範囲でのブロックを抽出するため、同じフィルタの入れ子の深さを示すパラメタを変化させて抽出操作を行った。フィルタ (4) を利用し、表 5 が抽出された。この結果から、struct option という構造体の変数 option[ ] の初期化を行う部分であったことが理解できる。

$\mathcal{F}_{Ctrl}$(AST, ⟨{opttbl.c, 0, EOF}⟩, 290, 2, WHOLE)
: ⟨{opttbl.c, 110, 334}⟩ (4)

表 5: フィルタ (4) の抽出結果

opttbl.c

```
110   static struct option option[ ] = {
111    {'a', &a_optname,
 ⋮          ⋮
289    {'x', &x_optname,
290      NUMBER|REPAINT, 8, &tabstop, NULL,
 ⋮          ⋮
334   }
```

5. フィルタ (4) の結果から発見された構造体 struct option について、その構造体の宣言部の抽出を行った。フィルタ (5) を利用し、表 6 が抽出された。$\mathcal{F}_{Decl}$(AST, ⟨{opttbl.c, 110, 110}⟩)
: ⟨{option.h, 52, 61}⟩ (5)

表 7 は表 6 で示す宣言をもとに、表 5 の抽出結果をまとめたものである。この関係から、jless の各オプションはこの箇所で管理されていることが読み取れる。

5.2 結果の比較

今回の実験で抽出した箇所について、新バージョンで比較を行うことにする。

1. 表 3 のタブをスペースへ変換しているブロックに相当する箇所は、新バージョンではその箇所が削除されていた。

表 6: フィルタ (5) の抽出結果

option.h

```
52  struct option
53  {
54    char oletter;
    /* The controlling letter (a-z) */
55    struct optname *onames;
    /* Long (GNU-style) option name */
56    int otype;
    /* Type of the option */
57    int odefault;
    /* Default value */
58    int *ovar;
    /* Pointer to the associated variable */
59    void (*ofunc)();
    /* Pointer to special handling function */
60    char *odesc[3];
    /* Description of each value */
61  };
```

表 7: struct option option[ ] のまとめ

制御文字	オプション名	...	共有変数	特殊関数	...
'a'	&a_optname	···	&how_search	NULL	···
'b'	&b_optname	···	&cbufs	opt_b	···
⋮	⋮	⋮	⋮	⋮	⋮
'x'	&x_optname	···	&tabstop	NULL	···
⋮	⋮	⋮	⋮	⋮	⋮

2. 大域変数 tabstop について調べるため、オプションの管理表を新バージョンでも作成した。共有変数 tabstop が削除され、特殊関数 opt_x が呼ばれるように変更されている。

3. 新バージョンで新たに追加された特殊関数 opt_x は、コードの表記は異なるが、表 3 の拡張である。

これらの結果から、目的の機能についての実行部、および将来の機能拡張時のために旧バージョンに準備されていた特殊関数の表を操作している箇所を抽出することができた。

表 8: 新バージョンの struct option option[ ]

制御文字	オプション名	...	共有変数	特殊関数	...
⋮	⋮	⋮	⋮	⋮	⋮
'x'	&x_optname	···	NULL	opt_x	···
⋮	⋮	⋮	⋮	⋮	⋮

表 9: 新たに追加された特殊関数 opt_x

```
switch (type){
case INIT:
case TOGGLE:
  /* Start at 1 because tabstops[0]
             is always zero.  */
  for (i = 1; i < TABSTOP_MAX; ){
        ⋮
  tabdefault = tabstops[ntabstops-1]
             - tabstops[ntabstops-2];
  break;
case QUERY:
  strcpy(msg, "Tab stops ");
  if (ntabstops > 2){
        ⋮
}
```

6 評価

これらの実験結果を抽出するためには、多くの試行錯誤を行っている。この様子を図 4 に示す。

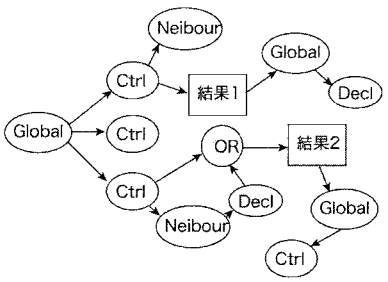

図 4: フィルタ適用の試行錯誤

この試行錯誤は、変更点に対して必要のない抽出を含んでいる。これはユーザが抽出対象の構造を把握していないことが原因である。本ツールはそのようなユーザがパラメタを変化させて試行錯誤を繰り返すことを前提とした操作の支援を行う設計

になっている。既存ツールとしてあげたSemantic grepでは、ユーザが抽出したい構造をクエリとして文字列で細かく構成可能のため、抽出対象の構造が明確である場合に適切といえる。本ツールでは、ユーザの漠然とした要求に対応するソースコードの箇所を試行錯誤によって発見するため、抽出要求と構造との対応が不明瞭である場合に適している。

しかし試行錯誤による抽出操作はユーザにとって負担になる場合もある。個々のフィルタは細粒度であるため、それらの出力を把握しつつ合成を行うことは、しばしば労力を要する。ただしこの問題は、典型的な抽出操作に対して事前に合成済みのフィルタを準備しておくことで、解決できる。その実現は今後の課題である。

また本ツールはC言語を対象としているが、C言語の特徴の1つであるポインタに関して、フィルタの設計をしていない。また他にも、現在のフィルタの合成で抽出できない構造も存在すると考えられる。しかし本ツールは、そのような場合にフィルタを新たに追加できる設計を行っている。よってより多くの実験を行い、不足するフィルタの検討が必要と考えられる。

7 おわりに

理解支援機能のひとつとして、変更や修正に影響する箇所を素早く抽出する機能をもつツールを細粒度フィルタの合成により設計、実装した。各フィルタの出力をパラメタにより細かく制御し、その結果に直列合成や論理合成などを繰り返すことで、様々な要求に柔軟に対応できる。またフィルタの直列や論理合成により利用者は必要な情報のみを得ることができる。

オープンソースのソフトウェアによる実験を行い、ツールの有効性を確認した。しかし現在は必要なフィルタの選択や合成を行うために、ユーザは多くの試行錯誤を必要とする。例えば特定の状況で使用されるフィルタとパラメタの組を事前に提供するなどの機能を拡張することで、利便性を向上させることが今後の課題である。

参考文献

[1] 永井路人, ソースコード理解支援のための表示自由度の高い視覚化ツールの研究, 北陸先端科学技術大学院大学 情報科学研究科 修士論文, 2006.03.

[2] Tama Communications Corporation, GNU GLOBAL source code tag system, http://www.gnu.org/software/global/

[3] Jani Jaakkola, Pekka.Kilpelainen, sgrep, http://www.cs.helsinki.fi/u/jjaakkol/sgrep.html

[4] Andrew M. Bishop, C Cross Referencing and Documenting tool, http://www.gedanken.demon.co.uk/cxref/

[5] R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey, Semantic Grep: Regular Expressions and Relational Abstraction, University of Wateloo Ontario, Canada, N2L 3G1, Proceedings of the Ninth Working Conference on Reverse Engineering(WCRE'02)

[6] 吉田敦, 山本晋一郎, 阿草清滋, 意味を考慮した差分抽出ツール, 情報処理学会論文誌 Vol.38 No.6 P.1163-1171, 1997.06.

[7] 荻原剛志, 會沢実, 鳥居宏次, 構文木の相互比較による複数バージョン比較分析方法の提案, ソフトウェア工学の基礎II P.21-30, 日本ソフトウェア科学会 FOSE'95

[8] 五月女健治, JavaCC コンパイラ・コンパイラ for Java, テクノプレス, 2003.

[9] Diomidis Spinellis 著, (株)トップスタジオ 訳, 鵜飼文敏/平林俊一/まつもとゆきひろ 監訳, Code Reading オープンソースから学ぶソフトウェア開発技法

[10] Kazushi (Jam) Marukawa, Jam less (jless), http://www25.big.jp/ jam/

[11] Bjarne Stroustrup 著, 長尾高弘 訳, プログラミング言語 C++ 第3版, Addison-Weslry Publishers Japan Ltd., 1998.11.