Att skriva testbar kod

Tobias Wrigstad

tobias.wrigstad@it.uu.se

Principer

- Litet förenklat kan vi tänka att ett **enhetstest** är en funktion y = f(x)
- Ett testfall blir då en tupel (x,u) där x är indata och u är det förväntade utdatat
- Denna syn på test är begränsad

  Fungerar bra för funktioner som inte bygger på andra funktioner

  Vad händer om y = g(z) där z = f(x)?

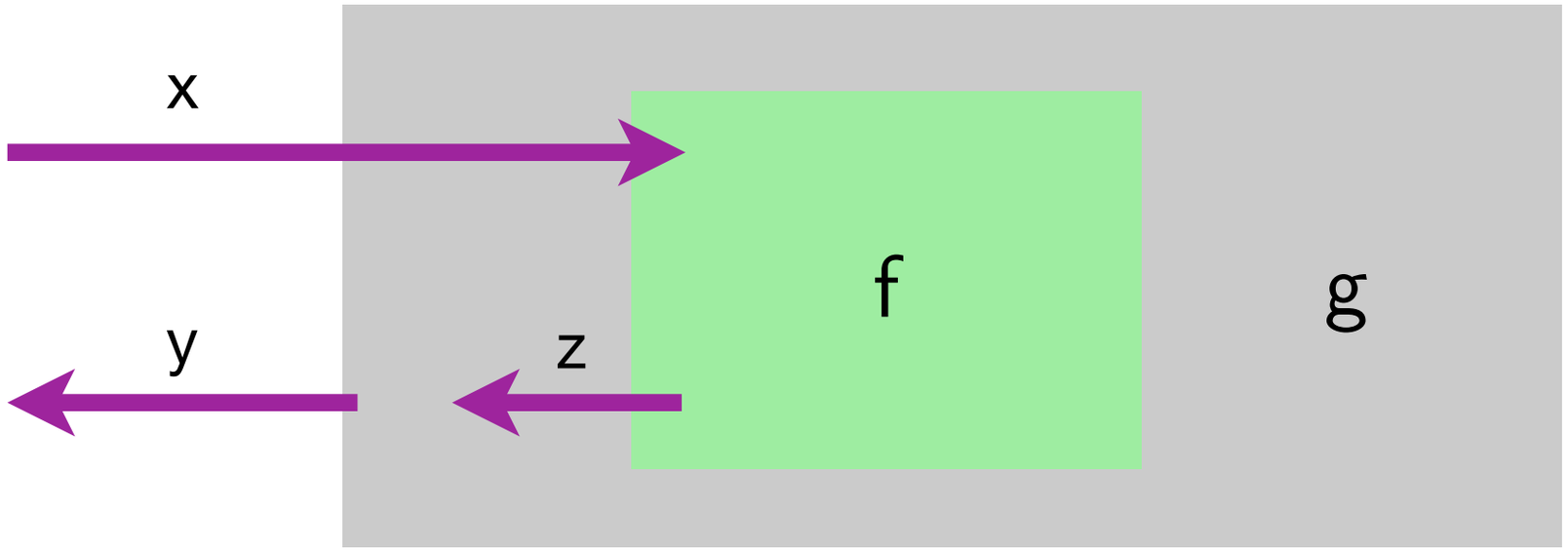
  Varför är detta ett problem?

- Vi kommer att jobba med exemplet att f är en funktion som returnerar en tidsangivelse och att g är en logger som använder sig av f.

Antag y = g(z) där z = f(x)

- Om för något (x,u), har vi y ≠ u – var ligger felet?

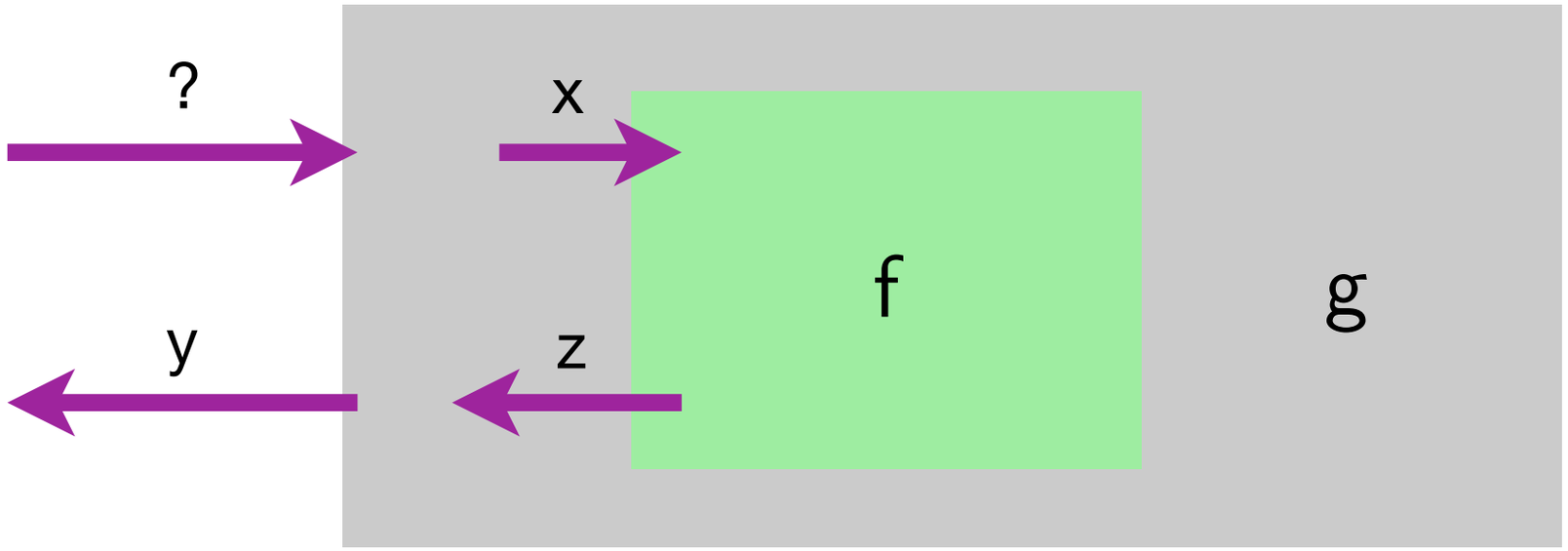
  f? g?



UU

Antag $y = g(z)$ där $z = f(x)$

- Vad är relationen mellan x och indatat till g?



UU

Komposition är integrationstest

- Möjligheten att isolera eller lokalisera en felkälla

  y = f(x) där f bygger på primitiver är ett test av f

  y = g(z) där z = f(x) prövar integrationen av f och g

- Integrationstest är också viktigt men ligger på en högre nivå än enhetstest

  Integrationstest måste förutsätta fungerande komponenter! (Varför?)

- För bättre blame control vill vi kunna pröva ”enbart g”

  Men hur?

  (z,u) skall inte bry sig om f

Att bryta ut beroenden

- Betänk testprogrammet t, vi kan göra antingen

  t(x) = g(u) där u = f(x), eller

  t(x, f) = g(f(x))

Vad är skillnaden?

Att bryta ut beroenden

- t(x) = g(u) där u = f(x)

  g:s beroende av f är hårdkodat inuti implementationen av t

  Vi kan inte byta ut f, eller pröva ”enbart g” (svårt att veta vad ”x är” också!)

- t(x, f) = g(f(x))

  Beroendet av f har brutits ut och skickas in som en del av testet

  Enkelt att pröva ”enbart g” med testfall där relationen mellan u = f(x) är känd

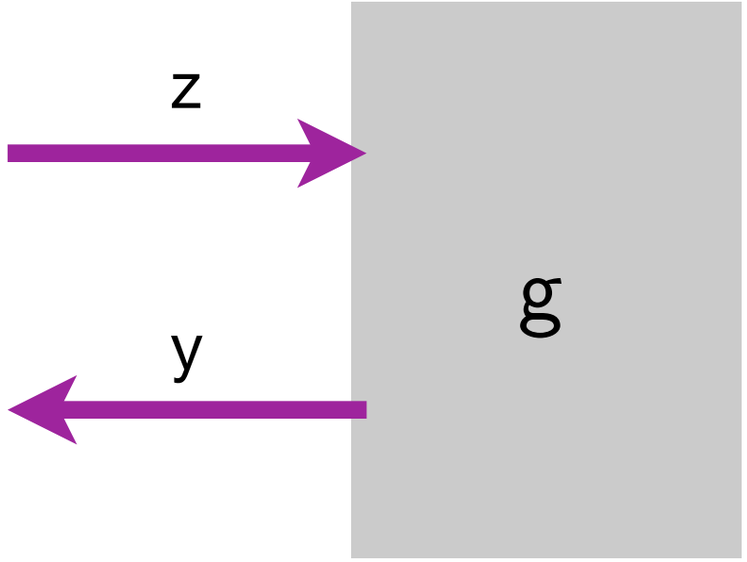
  Högre-ordningens programmering

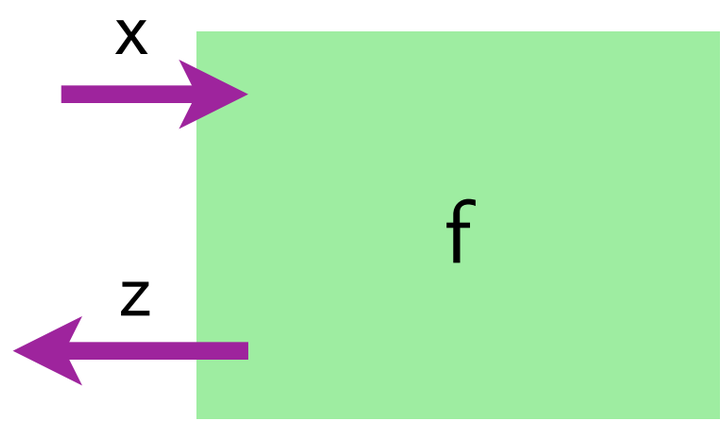
- Att faktorera ut beteende bryter (potentiellt långa) beroendekedjor som också försvårar återanvändning

- Utbrutna beroenden ger mindre byggstenar = enklare återanvänding

Test av g med $(x, f) = u$ där $u = g(f(x))$

- Relationen mellan x och z är nu klart, och vi behöver inte ens köra f utan kan använda kända z från f's tester i testerna av g.





UU

Bra kod är testbar kod ($\neg$testbar $\Longrightarrow$ $\neg$bra)

- När vi skriver kod måste vi alltid ta dess *testbarhet* under beaktande
- Funktioner skall kunna testas enskilt

  Minimera beroenden

  Minimera möjliga felkällor

- Inkapsling och informationsgömning kan ”förstöra” testbarhet

  Kan alla funktioner testas/åtkommas?

  Hur enkelt är det att skicka med indata till ett test?

  Kan vi enkelt kontrollera indata och utdata?

  (Code coverage vs. inkapsling)

PROBLEM!

UU

Exempel: Logger

- Anta att vi har designat och implementerat ett bibliotek för att logga C-strängar till disk i logger.c

  Loggern måste först initialiseras med `initLogger(filename)`

  Meddelanden skrivs till loggern med `logMessage(msg)`

  Loggern rivs ned med `destroyLogger()`

  Meddelanden buffras och skrivs till buffern (internt)

- Vi skall nu titta på kod för loggern och prata om dess testbarhet

```c
/* logger.c  */
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <time.h>
#include "logger.h"

/* Constants */
#define BUFSIZE 1048576
#define TIMESTAMPMAX  26

/* Module variables */
static FILE *logfile = NULL;
static char logbuffer[BUFSIZE];
static unsigned int logsiz = 0;
static time_t logtime;

/* Start code */
void initLogger(const char *fn) {
  assert(fn);
  assert(logfile == NULL);

  logfile = fopen(fn, "w");
}
```

```c
void logMessage(const char *msg) {
  assert(msg);
  assert(logfile);

  int msgsiz = strlen(msg) + 1;
  if (logsiz + msgsiz + TIMESTAMPMAX > BUFSIZE) {
    flush();
  }
  time(&logtime);
  char *timestamp = ctime(&logtime);
  strcat(logbuffer, timestamp);
  char *nl = strpbrk(logbuffer, "\n");
  if (*nl) *nl = ' ' else strcat(logbuffer, " ");
  strcat(logbuffer, msg);
  strcat(logbuffer, "\n");
  logsiz += strlen(msg) + strlen(timestamp) + 2;
}

void destroyLogger() {
  assert(logfile);

  flush();
  fclose(logfile);
  logfile = NULL;
}
```

```c
static inline void flush() {
  fwrite(logbuffer,
         logsiz,
         1, logfile);
  logsiz = 0;
}
```

Vad är problemen med att testa loggern?

- Låt oss börja med att titta på de tester som vi skall skriva

  Skrivs logmeddelandena ut korrekt?

  Är logmeddelandenas tider korrekta?

UU

En problemlista

- Loggern skriver alltid till filer på disk
- Buffertens storlek är fix och kräver därför större testdata för att tömmas
- Tidsangivelser skapas internt i loggern vilket omöjiggör jämförelser mellan två test eftersom deras tider kommer att skilja sig
- Vidare är loggerns design inte så bra

  Koden är svår att testa (som vi har sett), återanvända (varför?!) och modifiera

En bättre logger

- Möjligt att externt styra...

  ...var utdata skall skrivas

  ...buffertens storlek

  ...hur tidsangivelser skapas

```c
/* better-logger.c  */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <time.h>

/* Constants and helper functions */
#define TIMESTAMPMAX  26
#define initLogger(p) initLoggerWithPath(p, BUFSIZE)

/* Module variables */
static FILE *logstream;
static char *logbuffer;
static unsigned int logsiz;
static time_t logtime;
static unsigned int BUFSIZE = 1048576;
static unsigned int USES_BUFFER = 1;
```

```
void initLoggerWithPath(const char *_fn, unsigned int _bufsiz) {
  assert(_fn);
  assert(logstream == NULL);

  initLoggerWithStream(fopen(_fn, "w"), _bufsiz);
}

void initLoggerWithStream(FILE *_logstream, unsigned int _bufsiz) {
  assert(_logstream);
  assert(logstream == NULL);

  BUFSIZE = _bufsiz;
  logstream = _logstream;
  if (USES_BUFFER = BUFSIZ > 0) {
    logbuffer = (char *)malloc(BUFSIZE);
  }
}

static inline void flush() {
  fwrite(logbuffer, logsiz, 1, logstream);
}

static inline void flushAndReset() { /* should be public??? */
  flush();
  logsiz = 0;
}
```

```
const char *buffer() {
  return logbuffer;
}

void logMessageWithTime(const char *_msg, const time_t *_logtime) {
  assert(_msg);
  assert(_logtime);

  const int msgSize = strlen(_msg) + TIMESTAMPMAX;
  if (USES_BUFFER) {
    if (logsiz + msgSize > BUFSIZ) flushAndReset();
  } else {
    if (BUFSIZ < msgSize) logbuffer = realloc(logbuffer, (BUFSIZ = msgSize * 2));
  }

  char *timestamp = ctime(_logtime);
  strcat(logbuffer, timestamp);
  char *nl = strpbrk(logbuffer, "\n");
  if (*nl) *nl = ' ' else strcat(logbuffer, " ");
  strcat(logbuffer, msg);
  strcat(logbuffer, "\n");
  logsiz += strlen(msg) + strlen(timestamp) + 2;

  if (!USES_BUFFER) flushAndReset();
}
```

```
void logMessage(const char *_msg) {
  assert(_msg);

  time(&logtime);
  logMessageWithTime(_msg, &logtime);
}

void destroyLogger() {
  assert(logstream);

  flushAndReset();
  fclose(logstream);
  free(logbuffer);
  logstream = NULL;
}
```

Observationer

- Det är nu möjligt att testa loggaren utan att blanda in filhanteringen
- Koden blir något mer komplex, men inte mycket (är enbart beroende av biblioteksfunktioners korrekthet)
- Koden är nu enkel att testa, återanvända och utöka
- I språk som är på högre nivå än C är det enkelt att gå ”ännu längre”

Bryt ut beteendet helt

(För enkelhets skull skrivet i Python)

```
from time import ctime
```

```
def unbuffered(time, msg):
    print time, msg
```

```
def buffered(time, msg):
    global logbuffer
    if not "logbuffer" in globals():
        logbuffer = []
    if len(logbuffer) > 1024:
        for msg in logbuffer:
            print msg
        logbuffer = [time + " " + msg]
    else:
        logbuffer.append(time + " " + msg)
```

```
def initLogger(behaviour = buffered):
    global logbehaviour, logbuffer
    logbehaviour = behaviour

def logMessage(msg, time = ctime()):
    logbehaviour(time, msg)

initLogger()
logMessage("Foo")
logMessage("Bar")
```

...eller ännu bättre...

- Vi kan skapa ett beteende som skriver ut på en sträng!

```
def createStoreToArrayBehaviour(buffer):
    def storeToBuffer(time, msg):
        buffer.append([time, msg])
    return storeToBuffer

myBuffer = []

initLogger(createStoreToArrayBehaviour(myBuffer))
logMessage("Foo")
logMessage("Bar")
for msg in myBuffer:
    for element in msg:
        print element
```

```c
typedef void(*log_BH_ptr)(char *msg, void *resources);


static log_BH_ptr  log_BH    = NULL;
static time_BH_ptr time_BH   = NULL;
static void       *resources = NULL;


void initLogger(log_BH_ptr _log_BH, time_BH_ptr _time_BH void *_resources)
  assert(_BH);
  log_BH    = _log_BH;
  time_BH   = _time_BH;
  resources = _resources;
}

void logMessage(const char *_msg) {
  assert(_msg);
  char *timestamp = time_BH(resources);
  char *logmsg = malloc(TIMESTAMPMAX + strlen(_msg));
  strcat(logmsg, timestamp);
  strcat(logmsg, _msg);
  log_BH(logmsg, resources);
  free(logmsg);
}
```

```c
void logToString(char *msg, char *logtape) {
  /* Should of course check sizes etc. */
  strcat(logtape, msg);
}

void logToFile(char *msg, FILE *logfile) {
  fprintf(logfile, msg);
}

void logToFileWithBuffer(char *msg, FILE *logfile) {
  static int BUF_SIZE = 1024;
  static int BUF_USED = 0;
  static char *buffer = calloc(BUF_SIZE);
  int msg_length = strlen(msg);
  if (BUF_USED + msg_length > BUF_SIZE) {
    /* flush and reset */
  } else {
    strcat(buffer, msg);
    BUF_SIZE += msg_length;
  }
}
```

```c
char *generateTimeStamp(void *ignore) {
  static time_t logtime;
  time(&logtime);
  return ctime(logtime);
}

/* Here is how to set it up */
char *myString = malloc(1024);
initLogger(logToString, generateTimeStamp, myString);
```

Att testa ett binärt sökträd

- Vad är problemet här?

```
/* bst.h */
typedef struct _tree *Tree;

struct _tree {
  int value;
  Tree left, right;
};

Tree mkTree(int v);
void insert(Tree t, int v);
```

```
Tree t = mkTree(5);
insert(t, 1);
insert(t, 3);
insert(t, 7);
assert(t->element == 5)
assert(t->left->element == 1)
assert(t->left->left == NULL);
...
```

Hjälpkod för att testa ett binärt sökträd

```c
/* Returns the number of nodes in a tree */
int size(Tree t) {
  return (t) ? 1 + size(t->left) + size(t->right) : 0;
}

/* Returns the longest path to a leaf in a tree */
int depth(Tree t) {
  return (t) ? 1 + max(depth(t->left), depth(t->right)) : 0;
}
```

Låter oss pröva viktiga egenskaper för binära sökträd utan att exponera implementationen.

Växer det vid insättning? Duplikat?

```c
Tree t = mkTree(1);
assert(depth(t) == size(t) == 1)
for (int i=2; i<5; ++i) {
  insert(t, i);
  assert(depth(t) == size(t) == i)
}
```

Hjälpkod för att testa ett binärt sökträd

```
char *getPathForElement(Tree t, int element) {
  char *result = *path = (char*) malloc(depth(t));

  while (t) {
    if (element == t->element) {
      *path = '\0';
      return result;
    } else if (element < t->element) {
      *path++ = 'L';
      t = t->left;
    } else {
      *path++ = 'R';
      t = t->right;
    }
  }
  return ELEMENT_NOT_FOUND;
}
```

Låter oss se hur element flyttas eller stoppas in i trädet.

Borttagning!

```
Tree t = mkTree(3); insert(t, 5); insert(t, 1); insert(t, 2);
assert(strcmp(getPathForElement(t, 2), "LR") == 0)
assert(strcmp(getPathForElement(t, 5), "R") == 0)
```

UU

Hjälpkod för att testa ett binärt sökträd

```
void *getElementForPath(Tree t, char *path) {
  while (t && *path) {
    switch (*path++) {
    case 'L':
      t = t->left;
      continue;
    case 'R':
      t = t->right;
      continue;
    default:
      return MALFORMED_PATH;
    }
  }
  return t ? &(t->element) : PATH_TOO_LONG;
}
```

Låter oss se hur element flyttas eller stoppas in i trädet.

Borttagning!

```
Tree t = mkTree(3); insert(t, 5); insert(t, 1); insert(t, 2);
assert(getElementForPath(t, "LR") == 2)
assert(getElementForPath(t, "R") == 5)
```

UU

Sammanfattning

- Att beakta testbarhet vid utvecklingen styr koden bort från vissa mönster

  T.ex. funktioner som initierar en datastruktur i ett enda svep

  Vi vil kunna testa små bitar åt gången (isolerade från resten av systemet)

  Nackdel: nu kan vi se objektet i ett felaktigt tillstånd

- Undvik globalt tillstånd

  Data sparas mellan test

  Koden kan ibland bli något mer komplex utan globalt tillstånd

Sammanfattning

- Arbeta efter principen att varje kodenhet endast skall ansvara för ett åtagande

  Konsekvens: fler kodenheter (funktioner, klasser, moduler, etc.)

- Minimera beroenden

  Annars blir testerna väldigt komplexa

Tips och tumregler

- Det är extremt viktigt att välja bra namn (på allt!)
- Undvik ”fancy koding” men var smart
- Få rader ökar läsbarheten, för få rader minskar den
- Många små funktioner som går att kombinera är en design som underlättar återanvädning och underhållsbarhet – och därför också testning!
- Testning går att tänka på som återanvändning
- Använd `assert`:s, speciellt för sådant som aldrig skall hände (omöjliga situationer)
- Undvik `NULL`
- Initiera alltid variabler även om du vet att de kommer att tilldelas före de används

Tips och tumregler

Se koden i denna FL för flera dåliga exempel!

- Ta bort redundanta eller oanvänd kod – mindre = mer läsbart
- Undvik tilldelningar i booleska uttryck och i argumentposition
- Gör loopinvarianten tydigt – gärna på ett enda ställe
- Kod med för många hopp (`break`, `return`, etc.) är svår att följa
- Pröva alltid det mest sannolika fallet först (det är det som nästa är intresserad av!)
- Pröva alltid gränsvärder och index mot storlekar
- Titta alltid på returvärdena från funktioner som du utgår från lyckas (t.ex. `malloc`)
- Dokumentera allt användande av `malloc` inuti en funktion som leder till data som returnernas (annars blir minneshanteringen knepig)
- Lämna alltid tillbaka resurser på ett förtjänstfullt sätt