VCGL software quality sessions: Lessons from LSP refactoring

Anatoliy Antonov
April 2012

Overview

- Code documentation
  - Variable names
  - Method names
  - Commenting methods
- Methods and their arguments
  - Alternative branches
  - Const methods
  - By-value & by-reference
- Object-oriented programming
  - Class as abstraction
  - Instance vs. local variables
  - Instance vs. class methods
  - Interfaces
- Multi-file projects
  - Header includes
  - License placement

Variable names

- Wrong principle: “if code was hard to write, it should be as hard to understand”. Variable names should link to their **purpose**, not add to reader’s memory load
  - Can you understand purposes of these variables without comments?

```
/** @brief Main vector holding all points on which LSP will be applied */
QVector<TSData*> S;
/** @brief Vector holding all control points (not always in the same order as in S)   */
QVector<TSData*> C;
/** @brief Map each control point to its neighborhood  */
QMap <TSData*, QVector<TSData*> > V;
/** @brief The projection of the control points, obtained from MDP (must have the same order as C) */
QVector<TSPoint> CMDP;
```

  - And comments on variables stay in header file! How easy is it to understand this small piece of code?

```
/* map the position of every point to the proper control number */
QVector<int>  S2C(S.size(), -1);
for ( int i = 0; i < C.size(); i++) {
      int index = S.indexOf(C[i]);
      if ( index != -1 ) {
            S2C[index] = i;
      }
}
```

<-- comment here is also not exact,
adding some more memory load

Method names

- Method call (usually) is request to object to perform some **action**. Thus, it should contain this action (verb) in its name. Excepted from this rule are getter methods that can be named as property (variable-like)

```
 /** perform the LSP algorithm */
void lsp ();

/** Calculate the alpha values on position i,j  */
double alpha (int i, int j, QVector<TSData*>& V_i);

/** Perform Sammon's Mapping on given control points */
void sammon ();
```

comments are right!

it IS much better to actually call these methods
performLSP(),
computeAlpha()
performSammonMapping()

- Mostly you should avoid using generic names like do(), compute(), perform()

```
QVector<TSPoint> calculate (int controlPointsSize);
```

<-- this method implements whole algorithm for projection
name project() seems more adequate

Commenting methods

- Comment should adequately represent what method does

```
/** @brief Perform Sammon's Mapping on given control points
 *
 * @param controlPoints Set of given control points to perform Sammon's Mapping
 * @return Return the mapping of those points into 2D QPoints
 */
void sammon ();          <-- well... actually, this method takes NOTHING and returns NOTHING
                             what is given in @return is side-effect
                             I would use tag @attention to indicate it
```

Alternative branches

- Usually, if you have two clearly alternative branches, you should think about separating them into two different methods (especially if semantics is different)
  - In special cases it can be reasonable to have them in one method
  - However, the following is **not acceptable**

```
QVector<TSPoint> LSP::calculate (int controlPointsSize) {
    if ( projection.size() == 0 && S.size() >= 2 ) {
        pam ( controlPointsSize );
        sammon();
        if (false) {                                  <-- this block is NEVER executed
            lsp();                                        (by the way, compiler should give
        } else {                                          warning “unreachable code”)
            for ( int i = 0; i < S.size(); i++ ) {
                TSPoint tmp(CMDP[i].getX(), CMDP[i].getY());
                projection.push_back(tmp);            <-- usage of push_back() starting from empty vector
            }                                             can lead to multiple reallocations and degrade performance
        }                                                 when number of objects to put into array is known, use resize() first
    }
    return projection;
}
```

acceptable options:
1) bool parameter variable
2) comment this code out!
(and comment why it is commented out!)

Const methods

- Methods that do not modify internal state should have modifier const
  - first, it says “this method does not have side-effects”
  - second, without const you cannot call method on const object

```
class TSPoint
{
public:
        TSPoint();
        TSPoint(double x, double y);
        TSPoint(double x, double y, double z);
        ~TSPoint();

        double distance(TSPoint& point);          <-- cannot I compute distance to other point
                                                      without modifying this one?

        TSPoint operator+ (const TSPoint& c) const;
        TSPoint operator- (const TSPoint& c) const;     <-- correct
        TSPoint operator* (double scalar) const;

        double getX();
        double getY();                            <-- getters should be const by definition
        double getZ();
/* private part */
};
```

By-value & by-reference: arguments

- Invocation of this method **copies** whole vector **twice**

```
//variable is declared in class LSP as
QVector<TSData*> S;


LSP::LSP(QVector<TSData*> ts) {          <-- first time is copied here (initializing local variable ts)
        this->S = ts;                    <-- second time here (from local variable to instance variable)
}
```

- As we actually do not modify original data points during projection, we can use **constant references** and completely avoid copying

```
//declare variable as
const QVector<TSData*>& S;               <-- this is reference to vector that we can not change


LSP::LSP(const QVector<TSData*>& ts)     <-- parameter is not copied - it is passed by reference
: S(ts) {}                               <-- we need to initialize constant members before constructor body
                                             using constructor initialization list
```

By-value & by-reference: returning

- Using this method **copies** whole returned vector **twice**

```
QVector<int> LSP::randomPermutationVector(int size) {
        QVector<int> tmp(size);
       /* method body */
        return tmp;                                  <-- first copying is done in return statement: local variable is copied to temporary variable
}

//your code looks like this
QVector<int> permVec = lsp.randomPermutationVector(n);   <-- second copying is done in calling code: temporary variable is copied to your variable
```

- Give output vector to method as argument by **non-constant reference**
  - If method cannot fail, then you could use it straight away

```
void LSP::randomPermutationVector(int size, QVector<int>& outPermVec) {
        outPermVec.clear();
        outPermVec.resize(size);
        /* method body */                            <-- directly use outPermVec here
}
//call the method as: QVector<int> permVec; lsp.randomPermutationVector(n, permVec);
```

  - Sometimes you still do **one** copying to ensure consistency of result

```
void LSP::randomPermutationVector(int size, QVector<int>& outPermVec) {
        outPermVec.clear();
        QVector<int> tmp(size);
       /* method body - something can go wrong */    <-- use tmp here
        outPermVec = tmp;                            <-- everything went fine, copy result to output argument
}
```

Class as abstraction

- In object-oriented programming (OOP), class represents **one** abstraction
- This abstraction may be one entity or collection of entities. However, it should be **consistent**, and its name should represent abstraction
  - Having several methods for LSP, several for PAM and couple for Sammon's mapping in one class was too much - I separated them to three different classes

```
class LSP {
public:
    LSP(QVector<TSData*> ts);                                                          <-- public LSP-related methods
    QVector<TSPoint> calculate (int controlPointsSize);
private:
    void lsp ();                                                                       <-- private LSP-related methods
    double alpha (int i, int j, QVector<TSData*>& V_i);

    void sammon ();                                                                    <-- Sammon's mapping-related method

    void pam (int nc);
    double pamConfCost (QVector<TSData*>& medois, QVector<TSData*>& non_medois);       <-- PAM-related methods
    void getNeighbourhoods (QVector<TSData*>& controlPoints, QVector<TSData*>& non_controlPoints);

    QVector<int> randomPermutationVector(int size);                                    <-- one more Sammon's mapping-related
};                                                                                         method
```

Instance vs. local variables

- In OOP, instance variables of class represent **characteristics** (attributes) of abstraction relevant to problem being solved. Intermediate values of abstraction activities should be stored in local variables
- It also helps to understand data flow:

```
//original version: everything is hidden in instance variables
QVector<TSPoint> LSP::calculate (int controlPointsSize, bool bPerformLSP) {


  if ( projection.size() == 0 && S.size() >= 2 ) {
        pam ( controlPointsSize );                    <-- no idea about data flow
        sammon();                                     <-- no idea about data flow

        if (bPerformLSP) {
            lsp();                                    <-- no idea about data flow
        } else {
            for ( int i = 0; i < S.size(); i++ ) {    <-- here is a bug, by the way
               TSPoint tmp(CMDP[i].getX(), CMDP[i].getY());
               projection.push_back(tmp);
             }
        }
    }
    return projection;
}
```

```
//final version: LSP class does not need any variables
void LSP::project (const QVector<ILSPData*>& dataPoints,  int controlPointsSize,
bool bPerformLSP,  QVector<TSPoint>& outProjection){
        outProjection.clear();
        if ( dataPoints.size() >= 2 ) {
                PAM pam;
                QVector<ILSPData*> controlPoints;
                pam.pam ( dataPoints, controlPointsSize, controlPoints );
                QVector<TSPoint> controlPointsProjection;
                Sammon::sammon(controlPoints, controlPointsProjection);

                if(bPerformLSP) {
                        performLSP(dataPoints, controlPoints,
                        controlPointsProjection,
                        pam.getNeighboorhoodMappings(), outProjection);
                } else {
                        outProjection = controlPointsProjection;
                }
        }
}
```

Instance vs. class methods

- Again, object is defined by set of attributes, plus functions to operate on it
- When your abstraction lacks attributes (e.g. it is algorithm performed on external data), then it makes no sense to create objects of this class - write its functions as **static** (attached to class) methods
  - Without comments, LSP class definition now looks like

```
class LSP {
public:
    static void project (const QVector<ILSPData*>& dataPoints,
                         int controlPointsSize,
                         bool bPerformLSP,
                         QVector<TSPoint>& outProjection);
private:
    static void performLSP (
                         const QVector<ILSPData*>& dataPoints,
                         const QVector<ILSPData*>& controlPoints,
                         const QVector<TSPoint>& controlPointsProjection,
                         const QMap <ILSPData*, QVector<ILSPData*> >& neighborhoodMappings,
                         QVector<TSPoint>& outProjection);
    static double alpha (int i, int j, const QVector<ILSPData*>& dataPoints, const QVector<ILSPData*>& V_i);
};
```

It would also make sense to add private constructor declaration (to forbid creating instances of LSP)

Interfaces

- Sometimes your code needs **only a few** operations from its arguments. If code can be reused in different scenarios, declare interface and depend on its methods in your realization. Users of your code will need only to realize your interface (via direct subclassing/inheritance or through Adapter pattern), code itself will be used unchanged
- In this example, code was using big and complex TSData class (you can look at it in original code), while it requires only **one** method from input - calculate distance between two points
  - That was extracted to interface

```
struct ILSPData {                                                  <-- struct has all members public by default, which is exactly what interface requires
    virtual double distance(const ILSPData& other) const = 0;      <-- “virtual” with “=0” show that this method is to be implemented in realizations
    virtual ~ILSPData() {}                                         <-- virtual destructor is required for proper memory deallocation of subclasses
};                                                                     without inheritance empty destructor is redundant, compiler makes it for you implicitly
```

Header includes

- h file should include headers for types used in class definition which are:
  - templates (compiler limitation)
  - used as instance variables by value,
  - used as method arguments by value,
  - used as template parameters
- For types used only by references or by pointers only **declaration** is provided

```
//file lsp.h
#include <QVector>      <-- template
#include "tspoint.h"    <-- template parameter
struct ILSPData;        <-- used only by pointer - declaration will suffice, even if in this case pointer is template parameter
                            (do not forget to #include “ilspdata.h” in implementation file!)
class LSP {
public:
    static void project (const QVector<ILSPData*>& dataPoints,
                         int controlPointsSize,
                         bool bPerformLSP,
                         QVector<TSPoint>& outProjection);
private:
     static void performLSP (
                         const QVector<ILSPData*>& dataPoints,
                         const QVector<ILSPData*>& controlPoints,
                         const QVector<TSPoint>& controlPointsProjection,
                         const QMap <ILSPData*, QVector<ILSPData*> >& neighborhoodMappings,
                         QVector<TSPoint>& outProjection);
     static double computeAlpha (int i, int j, const QVector<ILSPData*>& dataPoints, const QVector<ILSPData*>& V_i);
};
```

License placement

- LSP implementation includes some “license” on code, and this license was placed in the beginning of the each file

- I extracted it to separate file, which, to my opinion, has several advantages:
  - If text of “license” is to be modified, it should be only modified once
    (this one actually also applies to source code: do not create opportunities to forget to update somewhere)
  - This file can give overview (list of affected files) and additional information (e.g. modifications after original author, remarks, usage of the code), i.e. can be used as place for external documentation
  - Huge banner is replaced with short reference which does not steal much attention from the code itself

Final remarks

- Remember: other people may want to use your code some day
- Make code easy to understand without context (“readable as a story”)
- Leave code consistent (that’s about unreachable code)
- Be aware of runtime behavior (here: references to avoid copying big objects)
- Use const for variables which are not modified after initialization
- Use power of object-oriented programming (class methods, interfaces)