Large Scale Design

Objectives

- To examine ways of approaching large problems.
- To provide techniques to assist in implementing them.

Large scale problems

- Most assignments completed at university are small, ‘toy’ examples.
- They usually consist of a single module, maybe 5 classes, and some inheritance.
- Most real world systems consist of several modules, a lot more classes, and differing levels of inheritance.

Large scale problems

- Successfully programming a small problem does not imply that large problems will also be solved the same way.
- Large scale problems have their own unique sets of challenges that are usually ignored in smaller problems.

Large scale problems

- *Design* – most small problems can be solved in your head, or maybe with a small jotting on paper.
- *Testability* – most small problems can be quickly tested, as the range of inputs is small.
- *Maintainability* – most small problems are intended to solve a problem once, and then be thrown away. As such, they are not coded with ongoing maintenance in mind.
- Large scale problems do not fit these criteria.

Design

- Most people jump straight into coding. I know I do.
- This stems from the fact that coding is fun, and documentation is boring.
- Even 5 minutes spent on design for a small problem can save hours of coding and debugging later.

Design

- Assuming 5 minutes of design saves 1 hour of coding and debugging, failing to spend 2 hours on a simple design for a large scale problem will blow the schedule out by 3 days (24 hours @ 8 hours per day).
- These numbers have been pulled from thin air, but the principle has not. Stop and take time to plan how you are going to code.

Design

- A typical approach for a large scale problem is to break it up into modules, and then to describe the objects that make up each module.
- For example, a program for sending email has *a message viewer*, *a message composer*, and a *server connection* to download new messages.

Design

- We now have 3 modules to break down into classes. We can write down what they can do, and what they know.
- Message Viewer
  - Knows: list of email messages, which messages are read or unread
  - Can do: view an email, forward an email, reply to an email, delete an email.

Design

- Message Composer
  - Knows: recipient lists (To, Cc, Bcc), message contents, message subject
  - Can do: send message, cancel message.
- Server Connection
  - Knows: server host and server port.
  - Can do: connect, retrieve message, delete message from server, disconnect.

Design

- For about 10 minutes worth of brainstorming, I've now got a fair idea of the data members and objects necessary.
- I will need a MessageViewer class, a MessageComposer class, a ServerConnection class, and an EmailMessage class.
- Again, this is a smaller version of a larger problem. (Feel free to try adding message filtering and folders to the design).

Design

```
const int POP3_PORT=110;

class ServerConnection {
   private:
      std::string host;
      int port;

   public:
      ServerConnection(const std::string&, int=POP3_PORT);
      void connect() const;
      Message retrieveMessage(int msgNo) const;
      void deleteMessage(int msgNo) const;
      void disconnect() const;
};
```

Design

- The class declaration has basically been written from the design work we've done.
- The const-ness of each function can be determined by the function's task.

Design for testability

- Design for testability is concerned with ensuring that what we've created actually works.
- Writing decent tests and test cases is a subject in itself. We'll just look at the basics.
- DFT is summarised by "If we can't test it, how do we know it works?"

Design for testability

- When designing a class it is necessary to know what the class should be doing – we can then write tests to ensure that it is doing what we intend.
- We can also write tests to ensure pre and post conditions are met when entering a function – this should cover most problems in classes.

Design for testability

- There are plenty of open source, free, test frameworks available on the Internet.
- These automate the process of running all the tests that will be generated by a large scale problem.
- These should be used where possible. Try `cppunit` out for one such framework.

Design for testability

- A simple version can be made with Make, some extra test classes, and some accessor functions (or a friend class).
- Note that this is very simple, and will quickly become tedious for large projects, but I have used it successfully on a number of occasions, with real world code.
- This works well if it is hard to test within the code, but easy to test using external programs.

Design for testability

```
class Test_ServerConnection {
   public:
      Test_ServerConnection();
};

// test.cpp
int main() {
   Test_ServerConnection();
   Test_MessageViewer();
   std::cout << “ALL TESTS PASSED\n”;
   return 0;
}
```

Design for testability

```
Test_ServerConnection::Test_ServerConnection() {
   std::cout << "Testing deleteMessage..."
      << std::flush;

   ServerConnection test1("localhost", 110);
   int msgCount = test1.countMessages();
   test1.deleteMessage(0);
   if (msgCount != test1.countMessages()) {
      cout << "FAILED!\n";
      throw 1;   // this will cause the test to exit
   }
   std::cout << "passed.\n";
}
```

Design for testability

- Now for the Makefile

```
TESTOBJS=test.o test_serverconnection.o
CXX=g++

test:  ${TESTOBJS}
    $(CXX) -o test ${TESTOBJS}

.o.cpp:
  $(CXX) -c -o $@ $< ${TESTOBJS}

test.o:      test.cpp
test_serverconnection.o: test_serverconnection.cpp
# ...etc...
```

Design for testability

- So, to test our program..

```
$ make test
g++ -o test test.o test_serverconnection.o
$ ./test
Testing deleteMessage...passed.
ALL TESTS PASSED
$
```

Design for testability

- This is a simple test to check pre and post conditions – here, we are ensuring that when we delete a message, only one is deleted.
- Using the ‘throw’ keyword will cause an exception to be thrown, which will terminate the program. We’ll cover exceptions in a later lecture.

Design for testability

- Note there is a problem with this test – what if no messages exist on the server?
- This highlights a problem – our program is only as correct as our tests show it to be. If our tests are incorrect, or problematic, we could miss some bug.
- It is important to write as complete a test suite as possible, and that these tests are correct.

Design for maintainability

- As a programmer, at some point you will start maintaining code. If you are lucky, it'll be code you wrote, so you understand it.
- Chances are it won't be your code.
- Be nice to whoever gets your code in the future by designing for maintainability.

Design for maintainability

- How easy a piece of code is to maintain depends on a number of factors. These include:
  - Simplicity of code and design
  - Readability of code
  - Code structure and modularity.

Design for maintainability

- Simplicity of a design is essential – this is the principle of **Occam's Razor**.
- *"One should not increase, beyond what is necessary, the number of entities required to explain anything"* – William of Occam
- Occam's Razor can be boiled down to the KISS principle: Keep It Simple, Stupid.
- A design should be no more complex than necessary.

Design for maintainability

- Being able to easily see what code is supposed to do is a big part of maintainability. This is helped by readability.
- Readability is about ensuring correct and consistent indentation, adequate commenting, and good use of whitespace to separate sections of code.

Readability

- My preference is as follows:
  - 3 spaces instead of tabs (NEVER use tabs).
    - In Vim, type `:set expandtab ts=3 sw=3`
  - Braces on the next line, indented to same level as parent line. Code inside braces is indented another level
  - Comments indented to same level as code they are commenting. Comments go above the code.
  - Lines are less than 80 characters in length. Break long lines and indent the second part two more levels.

Design for maintainability

- Separating sections of code into logical modules enables a future maintainer to quickly locate the file that contains a particular declaration or definition.
- Each class should have its own header file and implementation file (`.cpp` file).
- Avoid the temptation to put more than one class into a single file.

Aside: Compilation speed

- Keeping code separate enables Makefiles to quickly compile only those sections that need to compiled.
- Another point to increase compilation speed: Only `#include` a header file when you actually use a member function in a class. If you don't use a member function, use a **forward declaration** instead.

Aside: Compilation Speed

- A forward declaration just declares that the class exists, in place of the header file:

```
class Game;  // forward declaration
class Tournament {
   // . . .
   Game* final;
};
```

- If the game wasn't a pointer or reference, the default constructor would be called – which is a member function call, so needs a `#include`.

Design for maintainability

- Good documentation is also essential – not only for your design work, but also for your code.
- You can go to the extreme of every line of code having a justification for its inclusion.
- A middle ground between that and nothing is probably a good place to be.

Design for maintainability

- It is also worth considering self-documenting code tools, such as Doxygen.
- Doxygen attempts to be a javadoc for C++, and does a very good job.
- A separate ‘make documentation’ style Makefile target could help to automatically generate this.
- http://www.doxygen.org/

Large scale design

- We've seen a number of points that need to be taken into consideration with regards to large scale design, including maintainability and testing.
- These are *human* functions – if a programmer doesn't do them, they don't get done.

Large scale design

- Good large scale design is often a matter of good programming practices and procedures.
- We will take a look at the procedure known as eXtreme Programming or XP – pioneered by Kent Beck and Erich Gamma.
- This is just one approach, and needs to be tailored for individual situations.

eXtreme Programming

- The basic XP philosophy is ‘Embrace Change’. It looks at reducing the cost of making a change late in a product’s lifecyle.
- Traditional change management views changes later in a lifecycle as much more costly than changes earlier in a lifecycle.
- XP tries to change that, and, by making change cheap, allows **agile development**.

eXtreme Programming

- Why XP? The name Extreme Programming comes from the idea that, if certain practices are good, if we take them to the extreme, we'll get huge benefits.
- XP views testing as good, so XP practices are designed around testing.
- XP views change as good, so problems that are discovered are immediately fixed.

eXtreme Programming

- There are four fundamental values in XP
  - Communication
  - Simplicity
  - Feedback
  - Courage

eXtreme Programming

- Communication occurs between the customer and the developers. Customers set priorities (which job needs to be done), and the developers assess how long each job needs to take.
- Constant communication between customer and developer keeps the customer informed (they can adjust priorities), and allows developers to ask questions of the customer.

eXtreme Programming

- Simplicity is concerned with building what you need, and nothing more. The customer specifies jobs to be completed, and that is ***all*** that is done.
- Feedback is about applying the knowledge you are gaining as you code your application into making decisions about features and schedules.

eXtreme Programming

- Courage is about being able to make a decision that might be difficult, such as whether to rewrite a module, or ship a release without a feature.
- None of these values can really exist in isolation – feedback requires communication, communication ensures simplicity, courage ensures quality.

eXtreme Programming

- XP has 12 programming practices
  - Code and Design Simply
  - Refactor Mercilessly
  - Develop Coding Standards
  - Develop a Common Vocabulary
  - Adopt Test-driven Development
  - Practice Pair Programming

eXtreme Programming

- XP practices (cont)
  - Adopt Collective Code Ownership
  - Integrate Continually
  - Add a Customer to a Team
  - Play the Planning Game
  - Release Regularly
  - Work at a Sustainable Pace

eXtreme Programming

- Code and Design Simply is aimed at producing software that only solves the problem required. This in turn is likely to make it easy to change (as it doesn't do much).
- Refactor Mercilessly is meant to aid coding imply – whenever an opportunity exists, fix code . This process is **refactoring**.
- See http://www.refactoring.com/ for lots of articles on refactoring.

eXtreme Programming

- Developing coding standards ensures that each member of the team writes code the same way, making refactoring easier. Code looks the same.
- Developing a common vocabulary allows developers and customers to communicate with each other and understand what each other means.

eXtreme Programming

- Adopting Test-driven Development ensures that the code you write works as expected. In XP, you write a test first, then write code that passes that test.
- Pair Programming is the process of two developers working with a single keyboard and screen. One person codes, the other corrects and guides the coder. These positions change frequently as a piece of code is developed.

eXtreme Programming

- Adopting Collective Code ownership makes everyone part of the team – everyone is responsible. If an error is discovered, it is refactored, without blame.
- Integrating continually reduces the impact of new changes. Change occurs in small increments, rather than a large jump in features.

eXtreme Programming

- Adding a customer to a team ensures rapid feedback – if a developer needs clarification, they simply ask the customer. Likewise, the customer can see progress quickly and easily.
- Playing the planning game is vital to ensuring that the most important work is completed first.

eXtreme Programming

- Releasing regularly allows the customer to experience the product sooner, and allows them to see what their money is buying.
- Finally, XP advocates a 40 hour work week, to keep developers fresh, happy, and wanting to come into work.

eXtreme Programming

- An XP release cycle consists of consulting with the customer on the features they would like added. These are called **stories**.
- A story describes a way of using the program.
- The planning game ensures the critical stories are scheduled first.

eXtreme Programming

- Stories are implemented by programmers, sitting in pairs in **the bullpen**. The bullpen is an open area that allows communication between developers.
- After a story is completed, it is integrated, and *all* tests are run again to verify the program is still functioning correctly.
- Once all stories are completed for a cycle, the program is released. A new cycle is begun.

eXtreme Programming

- The value in XP is only discovered by doing it yourself. Get a decent test harness, grab a few friends, and implement a project using XP techniques.
- (Don’t do the assignment together – that’s plagiarism!)
- You will start to see the benefits and weaknesses of XP.

eXtreme Programming

- XP advocates having few design documents – this goes against common thinking.
- Some people view this as a weakness – it is – however, XP is also concerned with implementing *what you need right now*, and tries to ignore future requirements. Change is cheap, right?

eXtreme Programming

- XP cannot easily be applied in all situations. You may wish to only apply the testing philosophy and pair programming, for example. Thankfully, something is better than nothing in this case.
- Change isn't always as cheap as it seems – sometimes, massive refactoring is required. This is where courage is needed.

Finally…

- Keep up to date with literature on programming practices – chances are, XP will no longer be the ‘flavour of the month’ in a few years time.
- By keeping up to date, you become a better programmer, and able to implement large scale programs elegantly and efficiently.

- https://www.youtube.com/watch?v=hbFOwqYIOcU
- https://www.youtube.com/watch?v=kFM2Vcu-BRo
- https://www.youtube.com/watch?v=C0WiUiznoUQ
- https://www.youtube.com/watch?v=LkhLZ7_KZ5w