Introduction to 492 and Decoupling

José M. Vidal

Wed Jan 12 14:21:09 EST 2005

We provide the introduction to CSCE 492, and then go on to talk about decoupling. The decoupling slides are based on

- MIT 6.170: Open Courseware, Introduction[1]
- MIT 6.170: Open Courseware, Decoupling[2]
- MIT 6.170: Open Courseware, Decoupling II[3]

1 492 Introduction

- **Prerequisites**: all the appropriate classes (check your list), UML, programming knowledge, time, patience.
- Project-based class.
- I have split-up project into three steps in order to help you stay the course.

1.1 What You Need To Do

- By next week you need to have picked a group. Groups are 3-4 students each.
- Use the class mailing list if you want different partners.
- The week after that you will give a short presentation on your proposed project. See PS 1.
- At that time you will also email your first progress report. See Progress Reports. You will submit a new one every week.
- You will hear from me on whether I approve your project or not. This will usually require a group meeting with me.
- There will be a mid-semester code review/demo. By then you should have a tracer-bullet implementation.
- At the end of the semester there will be a final presentation, demo, and writeup.

1.2 Course Focus

- The job of software engineering requires construction and process.
- There are many processes vying for attention: Waterfall, Personal Software Process (PSP), Extreme Programming, Rational Method, etc.
- We will review some of their common ideas.
- The lectures will concentrate on construction.
- You should concentrate on the engineering on good quality software.
- As far as schedules, I suggest you use Painless Software Schedules[4].

2 What is Software Engineering?

- It is **not** just programming.
- Programming is to SE as building a bird house is to building the Sears Tower

- These are both construction projects. What are the differences?
  - Need more people!
  - Need more resources.
  - Need to be more careful.
  - Therefore, we need careful design.

3 The Challenge

- Civil engineering has had over 2000 years to mature.
- From caves, to huts, to villages, to pyramids, to aquaducts, to cities, to the Sears Tower.
- SE has only been around for less than 50 years. We are still building huts!
- Each program is different.
- It is hard to determine requirements.
- Design and fabrication are very close.

3.1 Complexity

- Research has shown that effort = length$^{1.5}$.
- For 1 page/day this means 100 pages in 4 years.

- The Mythical Man-Month[5] shows that more people is not always better.
  - Need to bring them up to speed.
  - They introduce new bugs.
  - Repeat old discussions.
- Solution: Reduce complexity
  1. Simplify requirements.
  2. Good design.
  3. Use a process that ensures 1 and 2.
- The lectures in this class will be aimed at showing some techniques for reducing complexity.

4 Why Software Engineering?

- Software sales and service represents one of the biggest, if not the biggest contribution to the US economy.
- Software is pervasive: Internet, transportation, energy, medicine, finance, embedded systems (cars).
- Software is expensive. The ratio of hardware to software procurement is almost zero.

4.1 Software is Still Buggy

- IBM survey, 1994
  - 55% of systems cost more than expected.
  - 68% overran schedules.
  - 88% had to be substantially redesigned.
- Advanced Automation System, FAA, 1982-1994
  - Industry average was $100/line, expected to pay $500/line.
  - ended up paying $700-900/line.
  - $6B of work ended up discarded.
- Bureau of Labor Statistics,1997
  - for every 6 systems put into operation, 2 canceled
  - probability of cancellation is about 50% for biggest systems.
  - average project overshoots schedule by 50%
  - 3/4 systems are regarded as operating failures

4.2 Software Can Kill

- In the Therac-25 accidents[6] a radiotherapy machine with software controller had a hardware interlock removed, but the software had no interlock software and failed to maintain essential invariants:either electron beam mode or stronger beam and plate intervening,to generate X-ray. Several deaths resulted due to burning. The programmer had no experience with concurrent programming.

- The International Atomic Energy Agency declared radiological emergency in Panama on 22 May,2001. 28 patients overexposed;8 died,of which 3 as result; 3/4 of surviving 20 expected to develop serious complications which in some cases may ultimately prove fatal. Experts found radiotherapy equipment working properly;cause of emergency lay with data entry If data entered for several shielding blocks in one batch then an incorrect dose was computed. The FDA[7] concluded that interpretation of beam block data by software was a factor.
- Ariane-5 (June 1996). The European Space Agency complete loss of unmanned rocket shortly after takeoff due to exception thrown in Ada code faulty code which was not even needed after takeoff due to change in physical environment. Undocumented assumptions were violated.

4.3 This is a Problem

> "The demand for software has grown far faster than our ability to produce it. Furthermore,the Nation needs software that is far more usable,reliable,and powerful than what is being produced today. We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways."

Information Technology Research:Investing in Our Future.[8] President's Information Technology Advisory Committee (PITAC) Report to the President, February 24, 1999.

5 Decoupling

- A key strategy for limiting complexity is to break a program into parts.
- The way in which parts relate to each other is called **coupling**.
- The more coupling there is in your program the harder (exponentially) it is to maintain.
- As such, we want to study **decoupling**.

6 Decomposition

- A program is composed of parts.
- There a many benefits derived from breaking a program into parts:
- Division of labor
- Reuse: sometimes.
- Modular analysis: small bits are easier to analyze and test.
- Localized change: if the design was good chances are that a new feature will require changes in a few parts.

6.1 Top-Down Design: Bad

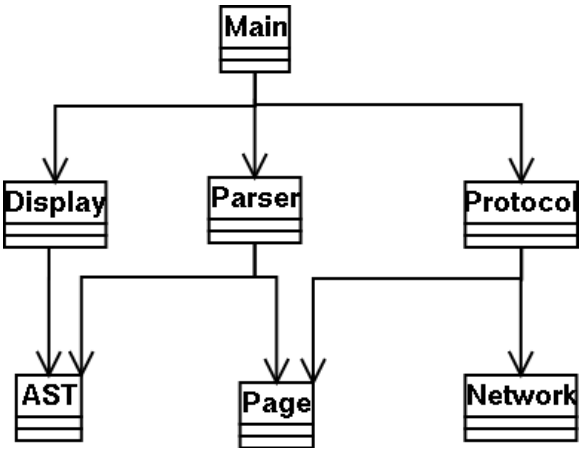
- The idea was to repeat
  1. If the needed part already exists then use it.
  2. Otherwise, split it into subparts, build them, and then combine them to form the needed part.
- Bad idea.
- The very first decision is critical and yet you don't know whether its any good until you get to the leaves.

- In practice, people also tend to hack the leaves to make them provide the functionality that they "should" based on the tree above them. This leads to a lot of coupling.

6.2 Horizontal Design: Good

- Consider multiple parts at roughly the same level of abstraction.
- Refine their description and then see how they all fit together. Do this "in parallel" for all. Do it before implementing any.
- Organize the system around data, not functions.
- *Decouple the parts so that you can work on one part independently of the others.*

7 Dependence Relations



- The simplest relationship is the **uses relationship**. We say that part A uses part B if A uses B in such a way that the meaning of A depends on the meaning of B.
- We can draw a diagram where each box represents a part and an arrow from A to B means that A uses B.
- Using this notation we can draw a block diagram for a web browser.

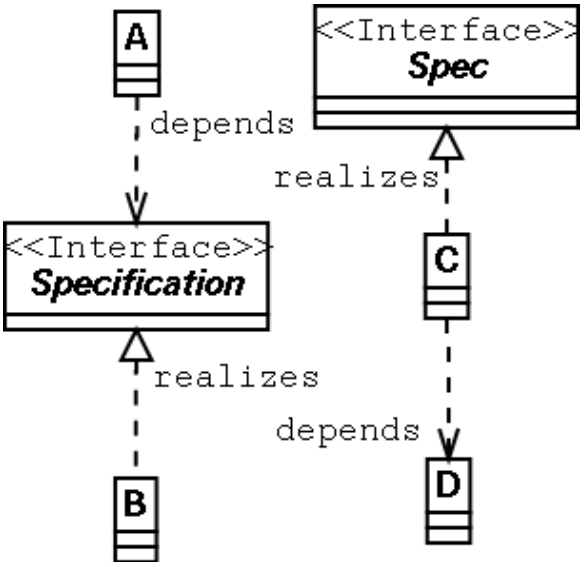
7.1 Uses Diagram Shapes

- *Trees* are often not possible because of reuse: two parts using another part, often as communication medium.
- *Layers* are a very useful way to add structure to the diagram. But, note that any diagram can be layered and layers are only useful if all the elements in a layer share some important common element. (e.g., n-tier architecture).
- *Cycles* are common and do not imply recursion.

7.2 Uses for Uses Diagrams

- *Reasoning.* How do you know if part A is correct? Check it and check all the ones that is uses, and all the ones they use, and so on... This is known as **impact analysis**.
- *Reuse.* to identify a subsystem, where the subparts do not use anything outside the subsystem.
- *Construction order.* We can determine which parts need to be done first. This helps a lot with division of labor.

7.3 Specifications



- A big problem with the uses relation is its transitivity. I would be better if we could verify a part by only checking the ones it is directly connect to.
- A **specification** tells us exactly what a part should do, but does not actually implement it.
- *Always separate specification from implementation.*
- We say that a part **realizes** if it implements it correctly.
- Specifications are represented by an abstract interface class in an UML diagram.

7.4 Specification Advantages

- *Weakened Assumptions.* We should make specifications as small as possible, that way we can say exactly what is used by the parts.
- *Evaluating Changes.* If a proposed change only affects the implementation part, and not the specification, then we do not need to worry about all the other parts that use the specification.
- *Communication.* If one person is building the part that implements the specification and the other is building a part that uses the specification, they do not need to talk. They must simply agree on the spec.
- *Multiple Implementations.* There can be many parts that implement a specification: different platforms, different requirements, etc.

7.5 Weak Dependencies

- Sometimes a part only requires that another part exist. That is, it does not call on any methods the part (just passes it along).
- We call this a **weak dependency** between the parts.
- For example, in our browser the main part could pass the AST part from the Parse part to the Display part.
- UML does not have a notation specifically for weak dependency, but you can simply annotate a dependency arrow with the label "weak".

8 Decoupling Techniques

- Dependencies are always a liability.
- The increase the rate of growth for program complexity.
- We want to minimize dependencies.
- **Decoupling** means minimizing both the quantity and quality of dependencies.
- The most effective way to reduce coupling is to design the parts so that they are simple and well defined, and bring together aspects of the system that belong together and separate aspects that don't.

8.1 Facade

- The facade pattern involves placing a new part between two existing parts.
- Each use of the old part will now have to go through the new part. The new part present a facade that looks like the old part.
- This makes sense when we expect the part's implementation to change. For example, if you need to use graphics but do not want to tie your whole program to a specific graphics library.

8.2 Hiding Representation

- A specification can (and usually should) avoid divulging how it represents data.
- Users of a specification should access data via functions defined by that spec. This is known as **data abstraction**.
- This way, if the representation changes later the interface will still remain valid.
- This also makes it easier to see the need for the specification.

8.3 Polymorphism

- A container part C will often depend on the part E which provides the elements of the container.
- For example, C might want to compare two elements for equality to make sure it is not adding the same element twice.
- In some cases we can make C **polymorphic** which means that C is written without any mention of special characteristics of E.
- For example, in Java all classes are descendant of `Object`. So, a container that takes elements of type `Object` is as polymorphic as can be.
- Lower degrees or polymorphism are more common, as when a part takes arguments of a given interface type.

8.4 Callbacks

- A part that uses a GUI usually needs to be alerted of events like a button pressing by the user.
- This is best achieved via a **callback** where the GUI part calls a given function on the Main part.
- Dependencies are usually reduced to a weak dependency on an Listener interface.

8.5 Shared Constraints

- Sometimes two parts are required to satisfy a constraint because of outside requirements.
- For example, a Read part and a Write part both need to agree on a file format in order to read and write files. If the file format changes both parts will need to change.
- You should try to localize functionality in one part. That is, use a single point of control.
- More generally, every piece of knowledge in your program should be expressed once and only once.

9 Namespaces

- Most object-oriented (and some non) languages have namespaces.
- By using different namespaces, two or more developers avoid duplicating names (variables, functions, classes, etc.)
- In Java method arguments have the **scope** of the method, member variables have the scope of the class and sometimes beyond (static).
- Further, Java has packages which group together many classes in a namespace.

9.1 Access Control

- Access to entities can be restricted in some languages.
- In Java a class can be declared as `public` so it can be accessed by any one.
- Member variables and functions can be marked as
  - `public` so anyone can access them.
  - `private` so it can only be accessed from within the class
  - `protected` so it can be accessed from within the package or by a descendent class.

10 Safe Languages

- A **safe language** is one where a part depends on another part only if it names it.
- C/C++ is not safe because you can (by mistake) write-over the contents of any variable.
- Safe languages employ several techniques—strong typing, memory management—to achieve their goal.
- They have been around since 1966: Algol-60, Pascal, Modula, Lisp, Java.
- Some safe languages guarantee type correctnes at compile time by **static typing**.
- Others do it at runtime: **dynamic typing**

11 Decoupling Example

- Your program takes, by necessity, a long time to execute. You want to keep the user appraised of whats going on.
- Solution 1:

```
System.out.println ("Starting download");
```

- But, if you wanted to add a timestamp to all messages then you would have to change them by hand.

11.1 Solution 2

- Why, of course, we should have defined a procedure to do this:

```
public class StandardOutReporter {
  public static void report (String msg){
    System.out.println (msg); }
}
```

- Now, if we wanted a timestamp we could just change it to

```
public class StandardOutReporter {
  public static void report (String msg){
    System.out.println (msg +  at:  +new Date ());
  }
}
```

- Notice the `static`.
- This procedure has become a *single point of control*.

11.2 Solution 3

- But, there is still a dependency on the notion of writing to standardout. What if we wanted to pop up a window?
- Specifically, what if we wanted to maintain a console and a GUI versions of this application?
- For the GUI, the function needs a reference to the widget. How does it get it?
- Define a reporter interface

```
public interface Reporter {
  void report (String msg);
}
```

- Each method that needs it gets a reference to this reporter, as in:

```
void download (Reporter r,....){
  r.report ( "Starting downloading" );
  ...;
}
```

- Implement the standardout reporter:

```
public class StandardOutReporter implements Reporter {
  public void report (String msg){
    System.out.println (msg + " at: " +new Date ());
  }
}
```

- Implement the GUI (Swing) reporter:

```
public class JTextComponentReporter implements Reporter {
  JTextComponent comp;
  public JTextComponentReporter (JTextComponent c){
    comp =c;
  }
  public void report (String msg){
    comp.setText (msg + " at: " +new Date ());
  }
}
```

- The parts that use the `Reporter` interface are de-coupled from its implementation. They don't care if its writing to stdout or a GUI.
- Notice that the GUI implementation creates its own widget reference and keeps a reference to it.

11.3 Solution 4

- But, passing that `Reporter` around could be a pain.
- We can hold a reference to it in a static field:

```
public class StaticReporter {
  static Reporter r;
  static void setReporter (Reporter r){
    this.r =r; }
  static void report (String msg){
    r.report (msg);
  }
}
```

- Now we can just do

```
//at the start of the program
StaticReporter.setReporter (new StandardOutReporter ());

//later on...
void download (...){
  StaticReporter.report ( Starting downloading   );
  ....};
```

- This might eliminate many needles weak references.
- Still, global variables are *dangerous*. They can make the program impossible to understand.

Notes

[1] http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-170Laboratory-in-Software-EngineeringFall2001/BD7247F
DD02-42D2-B8E9-75FF01E4F868/0/lecture01.pdf

[2] http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-170Laboratory-in-Software-EngineeringFall2001/59FB5D5
500F-4146-AA11-4F2E1DF0D9AA/0/lecture02.pdf

[3] http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-170Laboratory-in-Software-EngineeringFall2001/946BDCI
7FFB-4458-A4BC-1EFAC46875C9/0/lecture03.pdf

[4] http://www.joelonsoftware.com/articles/fog0000000245.html

[5] http://www.amazon.com/exec/obidos/ASIN/0201835959/multiagentcom/

[6] http://sunnyday.mit.edu/therac-25.html

[7] http://www.fda.gov/cdrh/ocd/panamaradexp.html

[8] http://www.ccic.gov/ac/report/

This talk is available at `http://jmvidal.cse.sc.edu/talks/decoupling`