Let's Look At Some Code

- A bit on inversion of control
- A bit on performance analysis
Creation

Classes
Classes

One way of thinking of a class is as an abstract data type plus inheritance and polymorphism. — McConnell, 6.1

(There are other ways of thinking about a class!)
Problems Classes Solve

A **Reason to Create a Class** is a **problem** that creation solves

- **Modelling**
  - Real or abstract objects
- **Complexity Management**
  - Reduce or Isolate Complexity
  - Hide details, limit effects, group control
- **Organisation**
  - Group functionality, manage variants, reuse

*You can always ask: What problem? & Is it (well) solved?*
# SOLID Principles Of Class Design

Synthesised by Bob Martin:

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRP</strong></td>
<td>The Single Responsibility Principle</td>
</tr>
<tr>
<td><strong>OCP</strong></td>
<td>The Open Closed Principle</td>
</tr>
<tr>
<td><strong>LSP</strong></td>
<td>The Liskov Substitution Principle</td>
</tr>
</tbody>
</table>
SOLID Principles Of Class Design

Synthesised by Bob Martin:

<table>
<thead>
<tr>
<th>ISP</th>
<th>The Interface Segregation Principle</th>
<th>Make fine grained interfaces that are client specific.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIP</td>
<td>The Dependency Inversion Principle</td>
<td>Depend on abstractions, not on concretions.</td>
</tr>
</tbody>
</table>
## SOLID Principles Credit

<table>
<thead>
<tr>
<th>Principle</th>
<th>Creator/Coiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRP, ISP, DIP</td>
<td>Bob Martin</td>
</tr>
<tr>
<td>OCP</td>
<td>Bertrand Meyer</td>
</tr>
<tr>
<td>LSP</td>
<td>Barbara Liskov</td>
</tr>
</tbody>
</table>
Class Relations

- Functionality is **divided** across classes (SRP, ISP)
  - How those classes **interact** is critical (ISP, DPSP)
    - They work together
    - In a controlled way (we hope!) (SRP, ISP)
      - Think unit vs. integration testing!
      - Via their **interfaces**
- (Some) Kinds of relations:
  1. **Is-A** (Inheritance)
  2. **Has-A** (Composition)
  3. **Works-With** (Collaboration)
Inheritance

- Class A **specializes** Class B
  - Class A and B **share** something
    - **Physically**: Code, variables, interfaces...
    - **Conceptually**: A is a kind of B
  - **LSP**: an A can **substitute** for a B
    - Callers **don't have to know** the specialising behavior
- Subclasses **extend** Superclasses
  - Add **new methods**
- Subclasses **override** Superclasses
  - **Polymorphism**
    - Critical and dangerous
Composition

A lot more is written about inheritance than about containment, but that's because inheritance is more tricky and error-prone, not because it's better. Containment is the workhorse technique in object-oriented programming.
Composition (2)

- Class B is (partly) made of Class A
  - A not **substitutable** for B
    - Certainly not **conceptually**
  - B **delegates** some aspects to A
    - **Person** has-a **name**
    - Let a **Name** class manage
      - **Structure** of names
        - See falsehoods about names
        - **Manipulation** of names

*The world exhibits* **fractal complexity**
Collaboration

- Classes have **responsibilities**
  - **Individual** classes may not be self-sufficient
  - **Other** classes which help **fulfil** the responsibilities are the **collaborators**
- Collaborators may be **coupled** to a greater or lessor degree
  - Inheritance generally yields **tighter** couplings
  - Composition generally yields **more moderate** couplings
  - Using collaborator **services** generally is even **looser**
- LSP **loosens** couplings
  - **Person** requires **Name**
    - Or **any subclass thereof**
Readability And Understandability

- Recall
  - Readability: ease of comprehending the code
  - Understandability: ease of comprehending the software system
- Abstraction is the connection
Creation

Routines

**Routine activity theory**

A likely offender

**CRIME**

A suitable target

The absence of a capable guardian

Physical convergence in time and space
Routines

One way of thinking about a routine is as an operation for an abstract data type.

Another way of thinking about a routine is as a (typically) named, invocable, block of code with a designated functionality (or purpose).

- Routines name and encapsulate behavior
  - At a fine grained level
- Routines are the smallest unit of abstraction
Routines & Classes

- Classes **package** routines
  - Routines provide the **external** interface
  - Routines provide the **internal** implementation
  - **Mixing** these breaks **encapsulation**
- The **set of class routines** (methods)
  - Define the **behaviour** of the class
Problems Routines Solve

*The most important reason for creating a routine is to improve the intellectual manageability of a program*
—McConnell, 7, Key points

- **Modelling**
  - Single actions or services (*verbs*)
- **Complexity Management**
  - Reduce or Isolate Complexity
  - Hide details, limit effects, group behavior, simplify
- **Organisation**
  - Group functionality, manage variants, reuse
What Is Cohesion?

Cohesion is the workhorse design heuristic at the individual-routine level.

For routines, cohesion refers to how closely the operations in a routine are related. —McConnell, 7.2.

- Ultimately, a routine is a block of code
  - I.e., a series of statements
  - I.e., a sequence of LOC
- The form of relation determines the type of cohesion
  - The strength of the relation determines the amount
    - At least, pairwise
The Good: Functional Cohesion

- Relation:
  - contributing to a given operation e.g.,
    - performing a calculation
    - enacting a behavior
    - providing a service

- Threats to functional cohesion
  - Irrelevant or superfluous code
  - Confused operation specification
  - Poor factoring
  - Book-keeping and auxillary behaviors
    - Debugging code, logging, etc.
Non-Ideal Cohesions: Utility

- Sequential, Communicational, and Temporal
  - May be **valid reasons** for a routine!
  - Issues arise when
    - A non-ideal cohesion **is confused for** functional cohesion
    - Seeking non-ideal cohesion **breaks** functional cohesion
- Mitigating the threats
  - Ensure all **pertinent** operations are captured
    - As routines!
  - **Document** the target **cohesion**
Non-Ideal Cohesions

- **Sequential**
  - Relation: order *dependency* and data *sharing* (with incomplete functional connection)
  - Problems: *conceals* operations, *couples* routines, breaks operation/routine *mapping*

- **Communicational**
  - Relation: data *sharing* (but no functional connection)
  - Problems: *Conceals* and *couples* operations

- **Temporal**
  - Relation: "*simultaneous*" execution (no func-conn)
  - Problems: *Conceals & confuses* ops; risks *coupling*
Poor Or Non-Cohesions

- **Procedural**
  - Sequencing without *data sharing*
  - Good variant(?), the *orchestrator*
- **Logical**
  - Functionally unrelated operations with a *master control structure*
  - And it's good variant, the *dispatcher*
- "**Coincidental**"
  - Relation: Existance in the same routine
  - The anti-cohesion!
Cohesion Between...

A class is a collection of data and routines that share a cohesive, well-defined responsibility. A class might also be a collection of routines that provides a cohesive set of services even if no common data is involved. —McConnell, 6
Cohesion Between...

- We’ve mostly talked about internal cohesion
  - I.e., relations between LOC inside a routine
- Routines are bundled in classes
  - To isolate dependencies
    - Esp. shared data
    - Also, implementation needs
    - Also, book-keeping
  - To form a coherent set of services
    - Classes determine the responsibilities

*We can perform similar cohesion analysis over a class*
Code Creation Is Problem Solving

*The problem we’re trying to solve is not lack of code*

- Problem solving is a **practical skill**
  - You get better at it the more that you do it
- A lot of problem solving is **matching**
  - There is an **existing** solution that you **recall**
  - There are **solutions** that **almost** work
    - And can be made too
  - There are **techniques** that are likely fruitful

*Experience goes a long way*
Boehm's Evidence

Following slides derived from Making Software, Chapter 10
Reading Papers

- These papers are **challenging**!
  - Even massaged a bit for the practitioner
  - Lots of technical jargon and techniques
  - Summarizing a vast literature
  - Challenging stats and presentations
- Don't panic!
  - These are read and reread
  - First reading should focus on **key points**
  - Later readings should focus on **the evidence**
The Role Of Architecture

- Key challenge (Boehm, Making Software, Chp 10)
  - How much should you invest in architecture?
    - Analogy to building
      - We pay the architect **10% of the cost of a building**
      - We should spend **10% of the project budget** on architecture
    - Is this **enough**?
    - How would we **know**?

*Note: statistically general conclusions may not apply in your case!"
Bohem’s Research Questions:

- "By how much should you expect the cost of
  - making changes or fixing defects
  - to increase as a function of
  - project time or product size?"
- "How much should you
  - invest in early architecting and evidence-based project reviews
  - before proceeding into product development?"
Economies

- Commodity manufacturing exhibits **economies of scale**
  - Making 1 chip may be much more expensive than 1000
  - The **unit cost** diminishes as the **number of units** increases
- Software end-unit costs are (can be) **zero**
  - Cheap to make a copy!
    - Installation & configuration may not be
  - So focus on **lines of code** or **bits of functionality**
- Software exhibits **diseconomies of scale**
  - The **unit cost** rises as the **number of units** increases
    - Potentially exponential! *Pgs 166-167 esp. useful*
Cost Ratios

- What's the ratio of cost to fix early vs. late?
  - 1970s
    - 1 in requirements to ≈100 post delivery
  - 1981
    - 1:100 for large code bases
      - But 1:5 for small (2,000-5,000 LOC)
  - 1996 survey
    - (70-125):1
  - 2000s
    - Some evidence of reduction from 1:100 to 1:20
    - Or even flat (for 1 million line code base)
Cost Ratios (For Coursework!)

- What's the ratio of cost to fix early vs. late?
  - Think of your coursework!
  - **Before** deployment (aka submission)
    - Small fixes are cheap
    - Esp. in the currency of the course, i.e., points
  - **After** deployment (aka submission)
    - Even "small" fixes are expensive (or impossible)

- Coursework builds over the semester!
  - So problems can build up
Two Strategies

- **Avoid** late bugs
- Make fixing late bugs **cheaper**
- Failure to do both **kills** the project
  - Failure to do one **may** be mitigated by the other
- **All** our activities should aim for this
  - Thus we want architectures that
    - preclude some bugs
    - confine the effects of all bugs
"20% of the defects account for 80% of the costs"
- "these 20% are...due to inadequate architecture..."

Two sorts of costs
- Direct costs
- Opportunity costs

Two example big failures
- the OS architecture didn't support fail-over when processors failed
  - lacked a key functionality
- assuming all messages are short
  - thus borking on 1 million character messages
Trade Offs

- More up front arch
  - Costs!
  - Runs risk of overruns
    - Since less time for everything else
- Potentially, getting arch right
  - Reduces rework time

*Note, changing requirements can kill getting it right*
Sweet Spots

Effect of Size on Sweet Spots

- Sweet spot
- Percentage of project schedule
  - oriented to initial architectural and risk resolution
- Added schedule directed to reuse
  (COCOMO2000 & R&D factors)
- Total % added effort

Sweet Spot Drivers:
- Rapid Change
- High Assurance
- High work demand
"...the greater the project's size, criticality, and stability, the greater the need for validated architecture feasibility evidence.

"very very small low-criticality projects with high volatility, the architecting efforts make little difference"

Note: There are other cost drivers; check the assumptions!
A Touch Of Management
Scope

- Just a few points
  - Software engineering is vast
  - Just management is vast
- 3 Management Loci
  1. Technical
  2. Organisational
  3. People
    - Other people!
      - You! (But we put this under professionalism)
Technical Management

- Version Control and Backups
  - Even for your own private stuff
  - Always
  - For your dissertation text, SEs, etc.
- Configuration
  - System, tools, environment
  - Know how to get to a clean system
  - Auto
    - Just enough documentation
Methodology

- See Agile Class
  - But!
    - Methodology isn't a panacea
    - Process doesn't ensure results
    - "Simple to understand, difficult to master"
      - Cop out?
- **Have** a methodology
  - However lightweight and idiosyncratic
  - Consistent practices are improvable
    - First improvement on inconsistent practices:
      - Make them consistent!
Organisation

- Take a dynamic view
  - An *improving* organisation is desirable
  - A *good* organisation is also desirable
  - Within certain bounds, *improving* is better
- Goodness vs. Fit
  - An organisation can be good in *many* dimensions
  - But the wrong *fit* for you
  - You need to assess *both*
    - Sometimes a "worse" organisation is a better *fit*
## What Programmers Are Doing

### Table 28.1: One View of How Programmers Spend Their Time

<table>
<thead>
<tr>
<th>Activity</th>
<th>Source Code</th>
<th>Business</th>
<th>Personal</th>
<th>Meetings</th>
<th>Training</th>
<th>Mail/Misc.</th>
<th>Documents</th>
<th>Technical Manuals</th>
<th>Operating Procedures</th>
<th>Misc.</th>
<th>Program Test</th>
<th>Totals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Talk on line</td>
<td>4%</td>
<td>37%</td>
<td>7%</td>
<td>4%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td></td>
<td></td>
<td>32%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Talk with manager</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Telephone</td>
<td>2%</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td>3%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td>14%</td>
<td>2%</td>
<td>3%</td>
<td></td>
<td></td>
<td>18%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write/record</td>
<td>13%</td>
<td></td>
<td></td>
<td>1%</td>
<td></td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Away or out</td>
<td>4%</td>
<td>1%</td>
<td>4%</td>
<td>6%</td>
<td></td>
<td>15%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Walking</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>1%</td>
<td></td>
<td>6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>2%</td>
<td>3%</td>
<td>3%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td></td>
<td></td>
<td>11%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Totals</td>
<td>35%</td>
<td>29%</td>
<td>13%</td>
<td>7%</td>
<td>6%</td>
<td>5%</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: "Research Studies of Programmers and Programming" (Baird 1964, reported in Boehm 1981).
Chapter 26. Novice Professionals: Recent Graduates in a First Software Engineering Job
Strengths And Weaknesses

- """"Among their **strengths** are:
  - Programming
  - Reading and writing specifications
  - Debugging (persistence and hypothesis generation)

- **Weaknesses** include:
  - **Communications**
  - Cognition
  - **Orientation** (engaging with a large code base and preexisting software team)"

*Chapter 26. Novice Professionals: Recent Graduates in a First Software Engineering Job*
Question Asking

"An overarching theme of new developers’ communication problems is knowing how and when to ask questions of others.

"In general, novices do not ask questions soon enough, and often struggle to ask questions at an appropriate level."

Chapter 26. Novice Professionals: Recent Graduates in a First Software Engineering Job
(Dis)Orientation

- "Understanding how team norms differ from those in academic settings confused some subjects."
- "Novices struggled to collect, organize, and document the wide range of information that they needed to absorb."
- "Novices had difficulty orienting themselves in the low-information environments presented by their project team, code base, and resources."
- "Some novices felt woefully isolated from their teams, sometimes not even knowing all the members of their team, and rarely knowing who to talk to about certain issues (or where that person’s office was)."
Techno-Social Skills

- **Technical** skills are important
  - But don't typically **dominate**
  - Technical ability won't make you **flourish**
    - though they help!
  - Low information environments
    - Require mastery of info finding
- **Social** skills key
  - Understanding the system and teams
  - **Good** question asking skills
- The two should **blend**
Professionalism
Responsibilities

- To **self**
  - A key set of **responsibilities** is to your self!
  - You have the **strongest and most fundamental responsibility** there!
- To **others**
  - Inside your circle
  - Inside your organisation
  - With whom you have commercial obligations
  - "The profession"
  - Society at large
Wide Responsibilities

- "Small bugs" lead to security fails
  - That compromise the privacy of millions or billions
- "Small inefficiencies"
  - Huge carbon footprint
- "Harmless" business models
  - Can distort (or enhance!) society
  - You are the product

*Software problems can easily become global problems*
Personal Virtues

- McConnell talks of several
  - (Interesting discussion in Making Software as well)
  - Mostly about what makes a good programmer
    - Or team member
- Key ones to consider
  - Intellectual Honesty and Humility
  - Curiosity
  - Habits
Software Engineering Is Challenging!

- As we've seen!
  - "In software development, even basic knowledge changes rapidly. The person who graduated from college 10 years after you did probably learned twice as much about effective programming techniques." — McConnell, 33.8.
- Part of professionalism
  - Is keeping up
    - Blogs
    - Check out a textbook every few years
    - Dedicated time to learning!
Software Engineering Is Challenging!

I graduated in Computer Science in the early 2000s.

When I took a Databases class, NoSQL didn’t exist.
When I took a Computer Graphics class, OpenGL didn’t support shaders.
When I took a Computer Security class, no one knew about botnets yet.
When I took an Artificial Intelligence class, deep learning didn’t exist.
When I took a Programming Languages class, reactive programming wasn’t a “thing”.
When I took a Distributed Systems class, there was no Big Data or cloud computing.
When I took an Operating Systems class, hypervisors didn’t exist (in PCs at least).
When I took a Networking class, there was no wifi in my laptop or internet in my phone.

Learn the fundamentals. The rest will change anyway.
Software Engineering Is Challenging!

- As we've seen!
  - "Older programmers tend to be viewed with suspicion not just because they might be out of touch with specific technology but because they might never have been exposed to basic programming concepts that became well known after they left school" — McConnell, 33.8.
- Ageism, sexism, ablism (and several other -isms) are big problems
  - particularly in Comp Sci and Soft Eng
- CS and SE also show a generous spirit
  - Look at many open source software projects
Ageism

- **Headline:** How the tech industry’s youth cult is driving older workers to plastic surgery
- **Headline:** The Brutal Ageism of Tech
- **Headline:** Is Silicon Valley Ageist Or Just Smart?
  - **Quote:** "Facebook’s Mark Zuckerberg told the audience: “I want to stress the importance of being young and technical. Young people are just smarter.”
- **Headline:** Silicon Valley’s dirty secret - age bias <-- Key one to read!
Anatomy of an Enduring Gender Gap
Woman In CS: Another Graph

Anatomy of an Enduring Gender Gap

1 Includes “Computer Science” and “Data processing or computer programming.”
Outright Harassment

- The Internet Problem We Don't Talk Enough About
- Online Harassment 2017
  - All sorts of harassment (and there's a lot)
  - Details some disproportionate effects
    - "44% of men and 37% of women"
    - Sexualised abuse:
      - 21% of women/9% men ages 18 to 29
      - 53% women ages 18 to 29 received unsolicited explicit images
    - "35% of women [16% of men]...describe their most recent incident as either extremely or very upsetting"
Professionalism

- We conceive ourselves as a **profession**
  - See the BCS or ACM or IEEE
  - We have professional **standards**
  - We take **responsibility**
    - Python Community Code of Conduct
    - GNU Kind Communications Guidelines
- Creativity, spontaneity, fun
  - Are not opposed to professionalism!
Aim For The Best
Make Things Better
Wrap Up
Thanks!

Fourth time for me!

Second time for Christos!!

We hope you've learned stuff

We have!
Coursework

- You’ve done a lot
  - CW4 still to come!
  - Let’s talk about the report
Exam

- **2 hr** limit
- **Electronic**
- A **fast** person should take ≈1 hr
  - Fast != does best
- Final version still must go through moderation
  - So I can’t say **firmly**
  - But expect
    - 23+ something MCQs
    - 3-5 Short Essay
MSc Projects

- Project book will be coming out soon
  - Pick projects that **challenge** you!
  - Special considerations if you're interested in PhD studies
- Three projects from me (at least) potentially of interest:
  - Coursework Tools
  - Generating multiple choice questions based on digitalised clinical pathways
  - Legal Tech Projects
SUPERFUN!

- Wed is HALLOWEEN!!!
  - I will be doing something!
  - Lots going on!
    - Cool movies this weekend at the Victoria Baths!!
- A week from Monday!
  - Remember, Remember the Fifth of November
  - Guy Fawkes night!
  - Awesome fireworks and giant bonfires!!
    - https://tinyurl.com/bonfires2018
  - Don't miss it
  - I recommend Heaton Park (which is awesome anyway)
  - Platt Field Park is closer
THRILLS.....
....And CHILLS!!!!