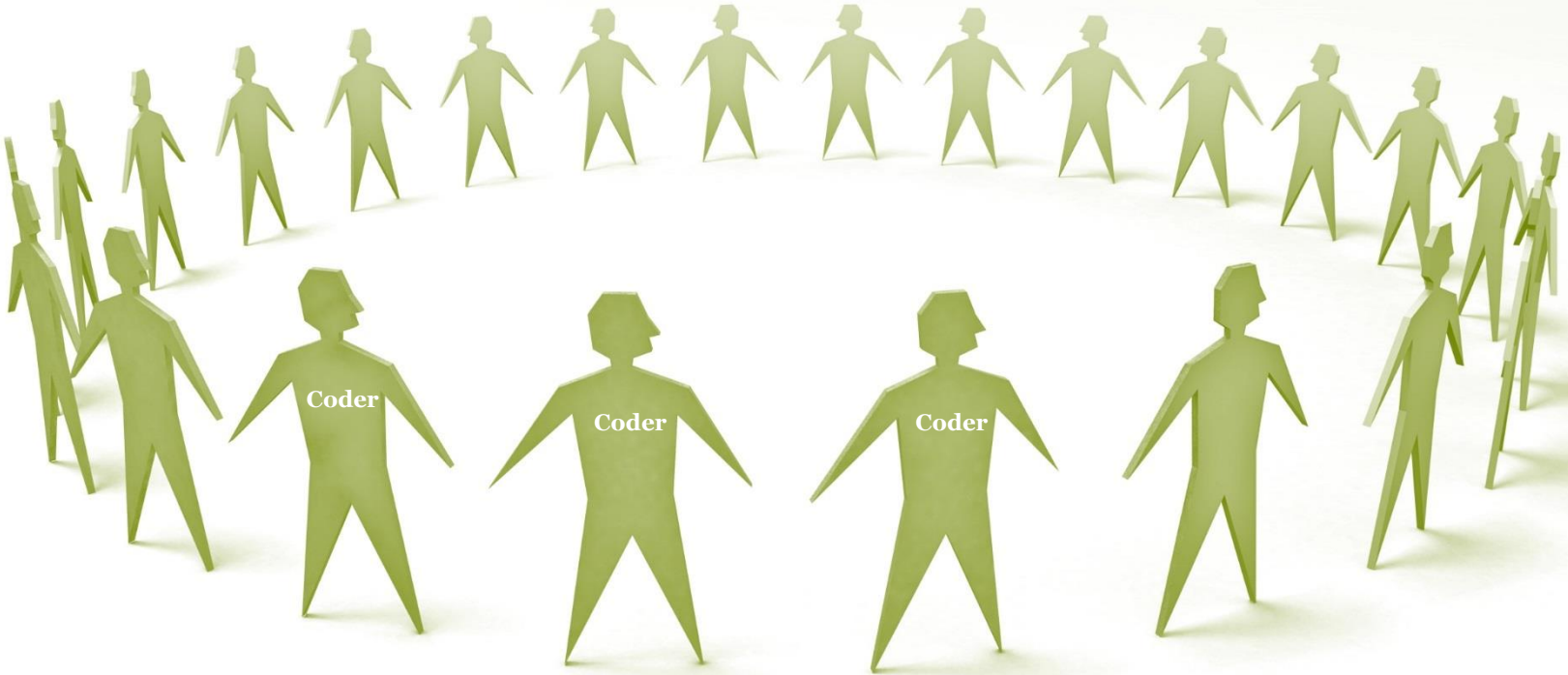
Building Software Capital

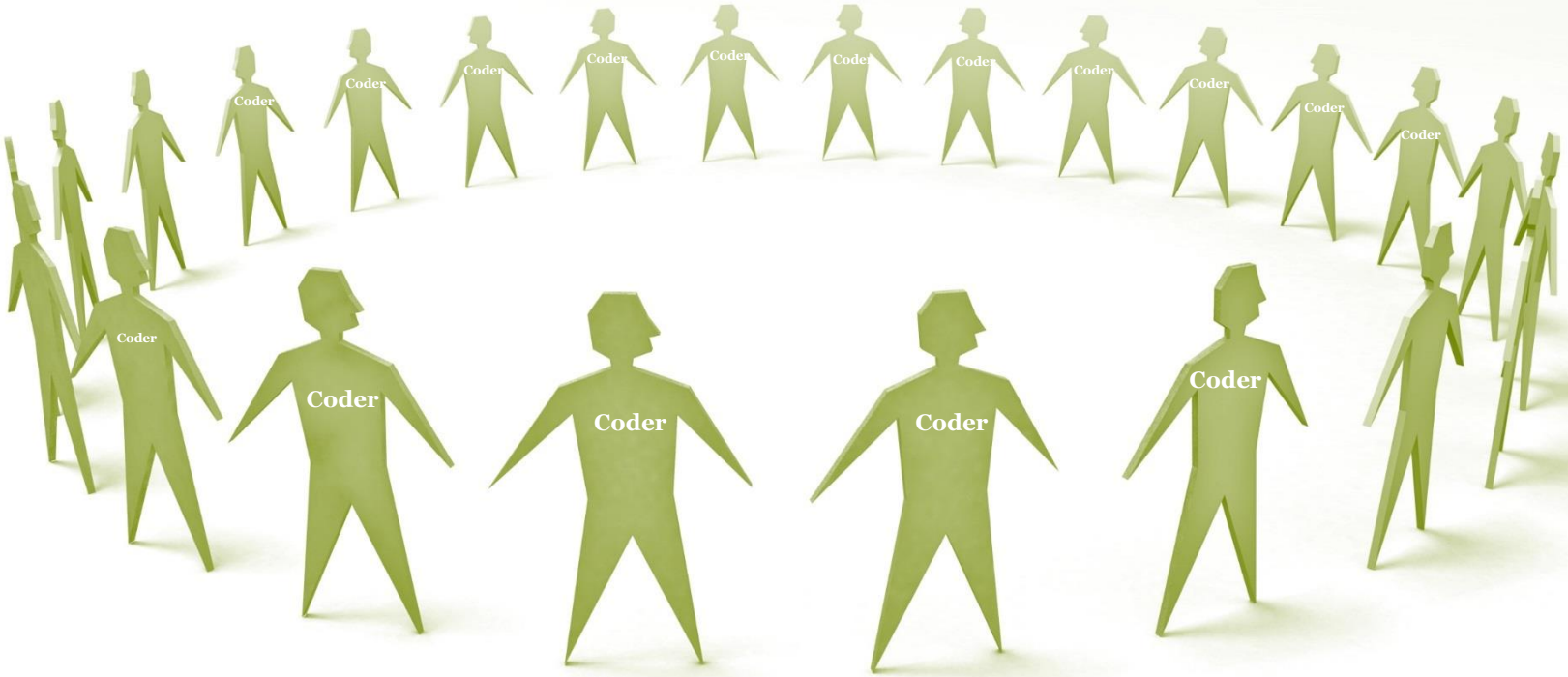
How to Write the Highest Quality Code and Why

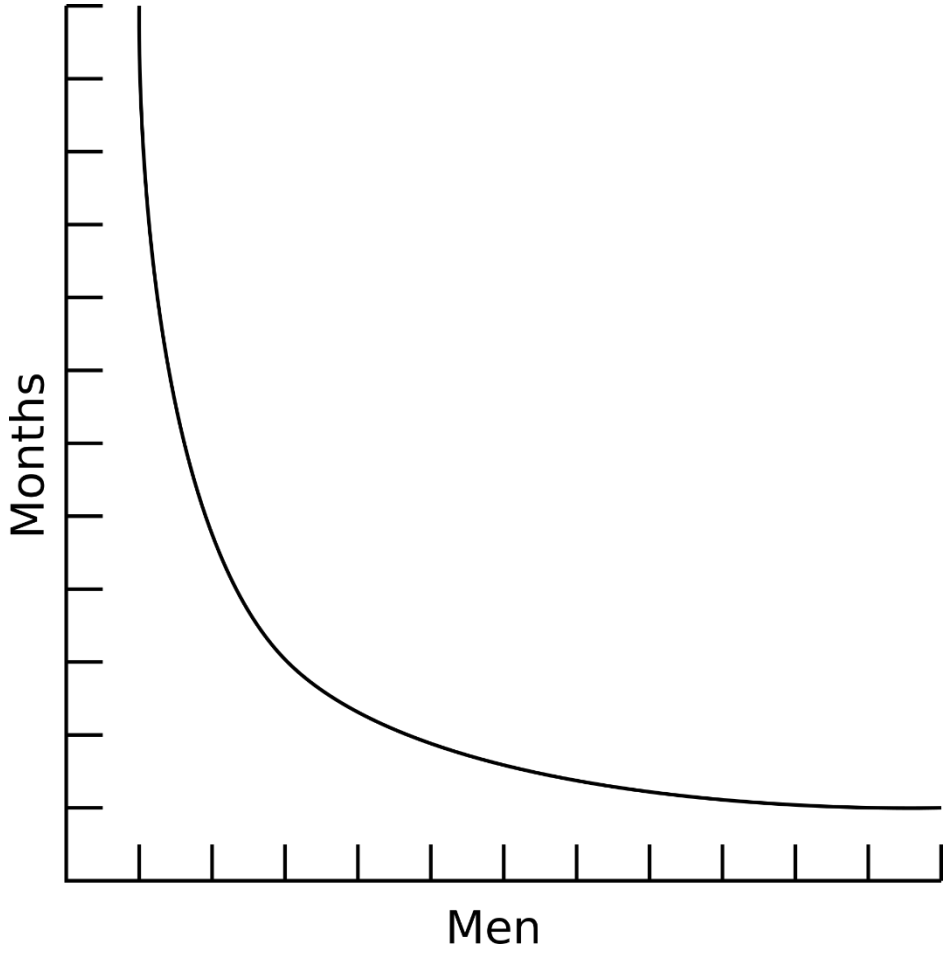
David Sankel
9/21/2016

Bloomberg

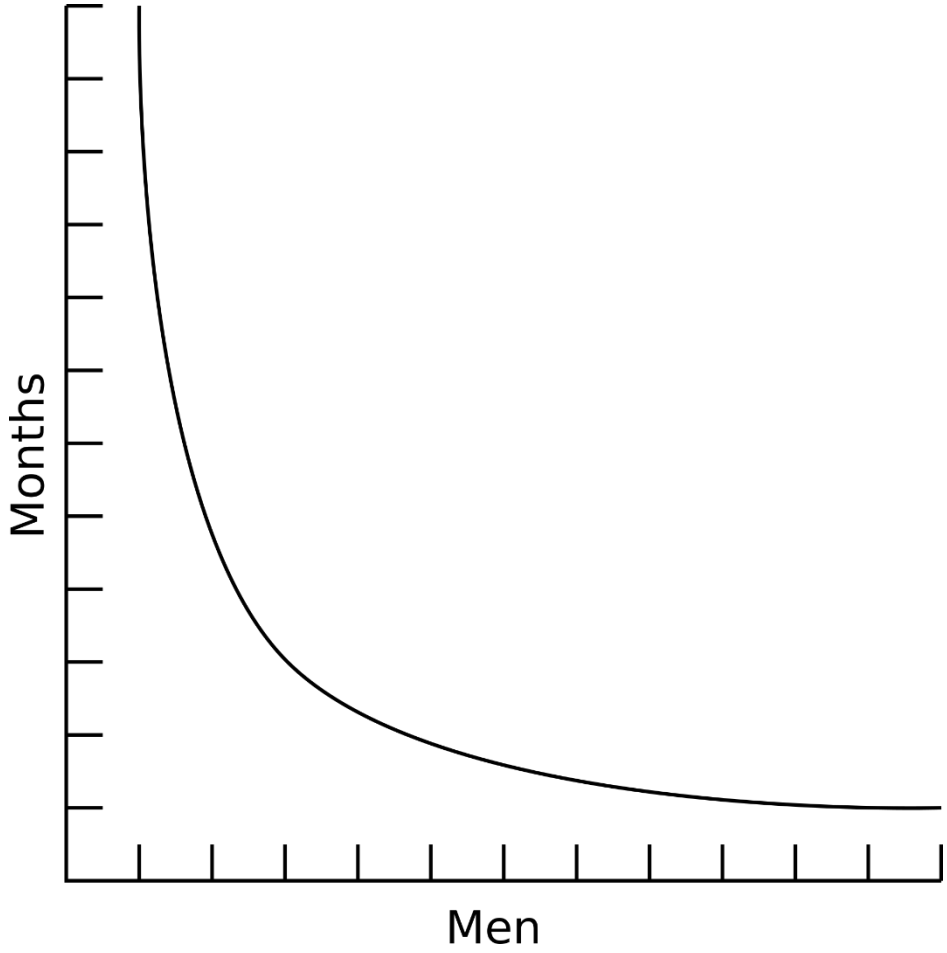
CppCon 2016

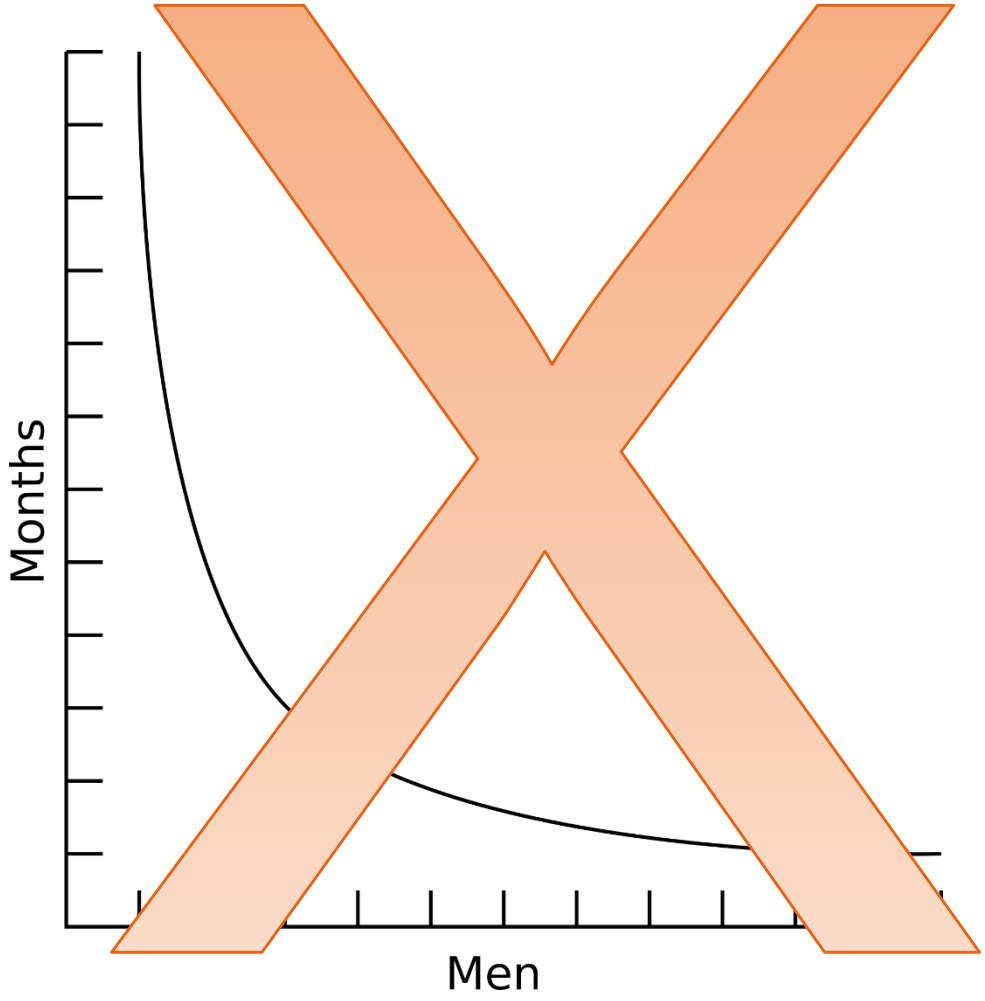

Coder
Coder
Coder

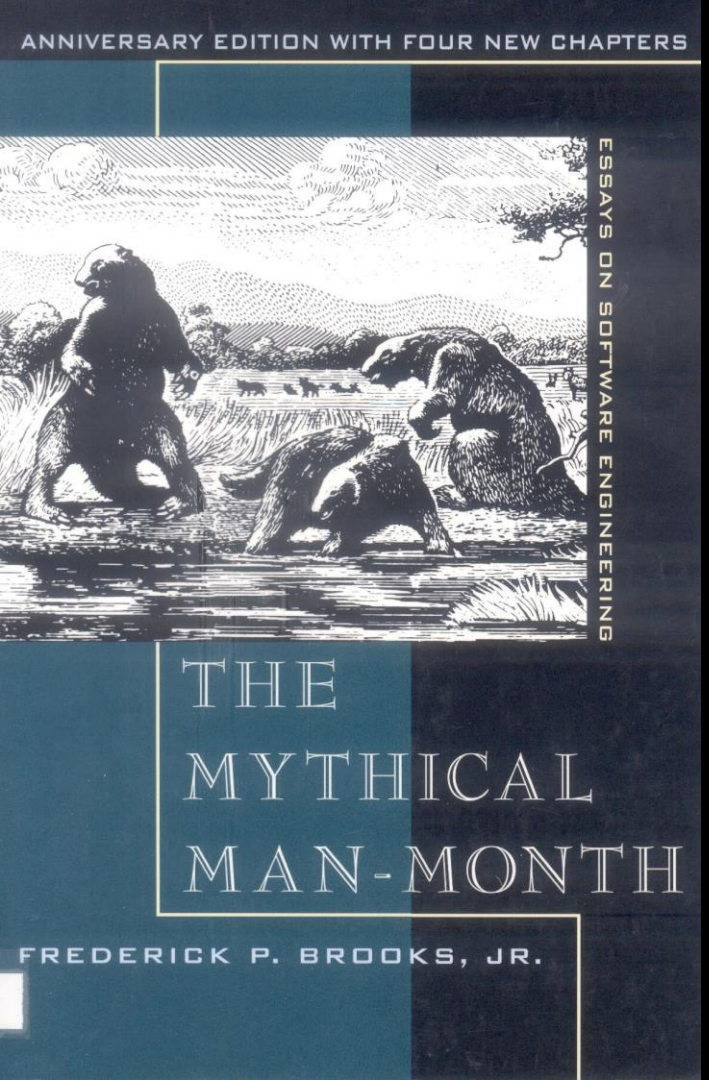

Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder


Months
Men


RailsConf 2006


Months
Men

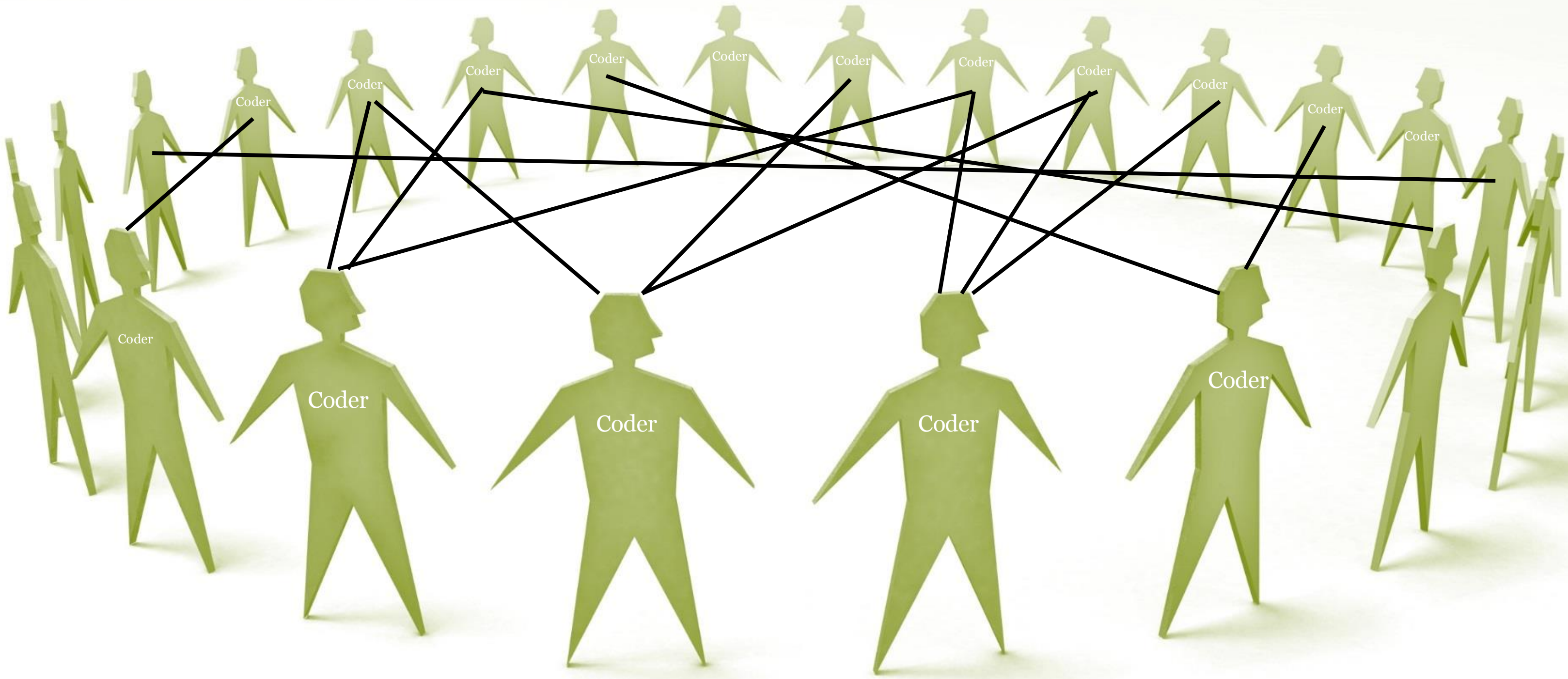

Months
Men

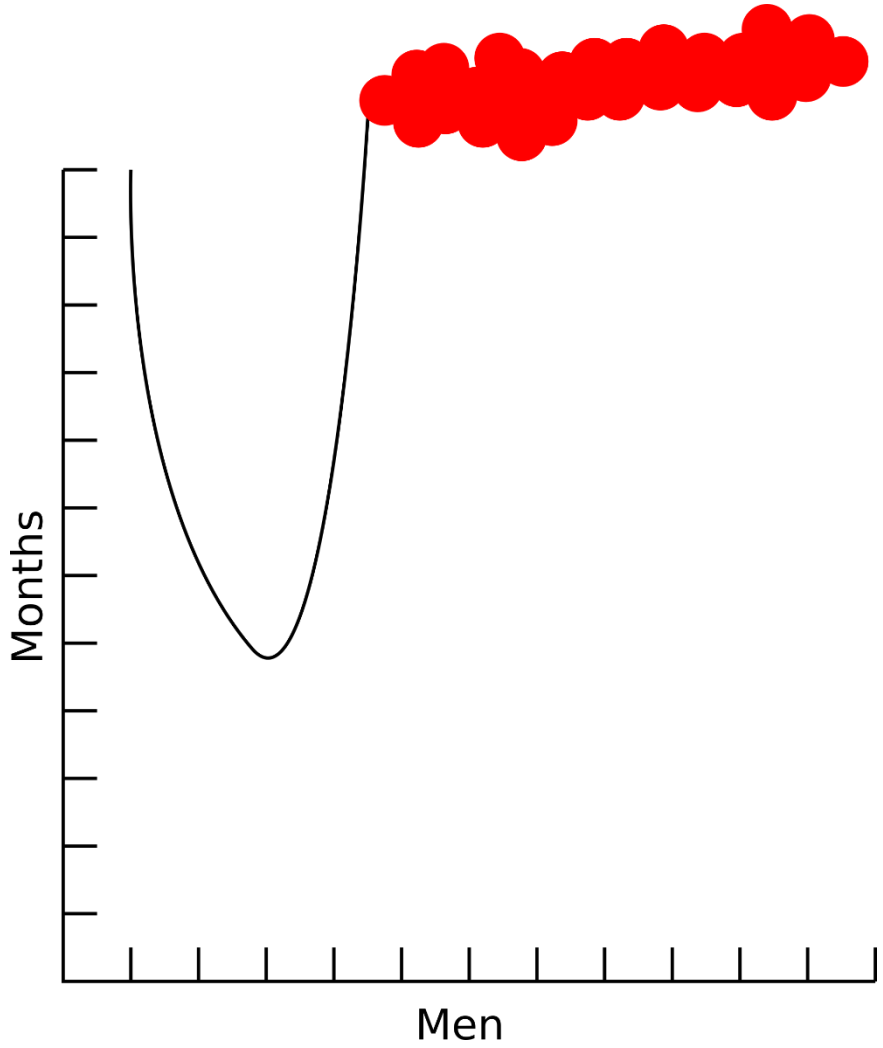


The Mythical Man Month

- Frederick P. Brooks
- 1975


Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder


Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder
Coder

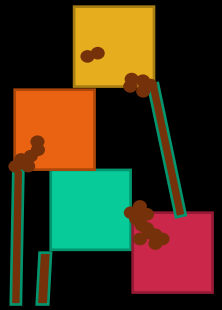

Months
Men

Goal: Improve Time to Market

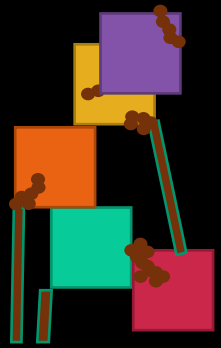
- Optimize Team Size
- Hire Top Developers
- ?

What about Reuse?

Dev: Hey, it works :/



Dev: We really should fix this someday.

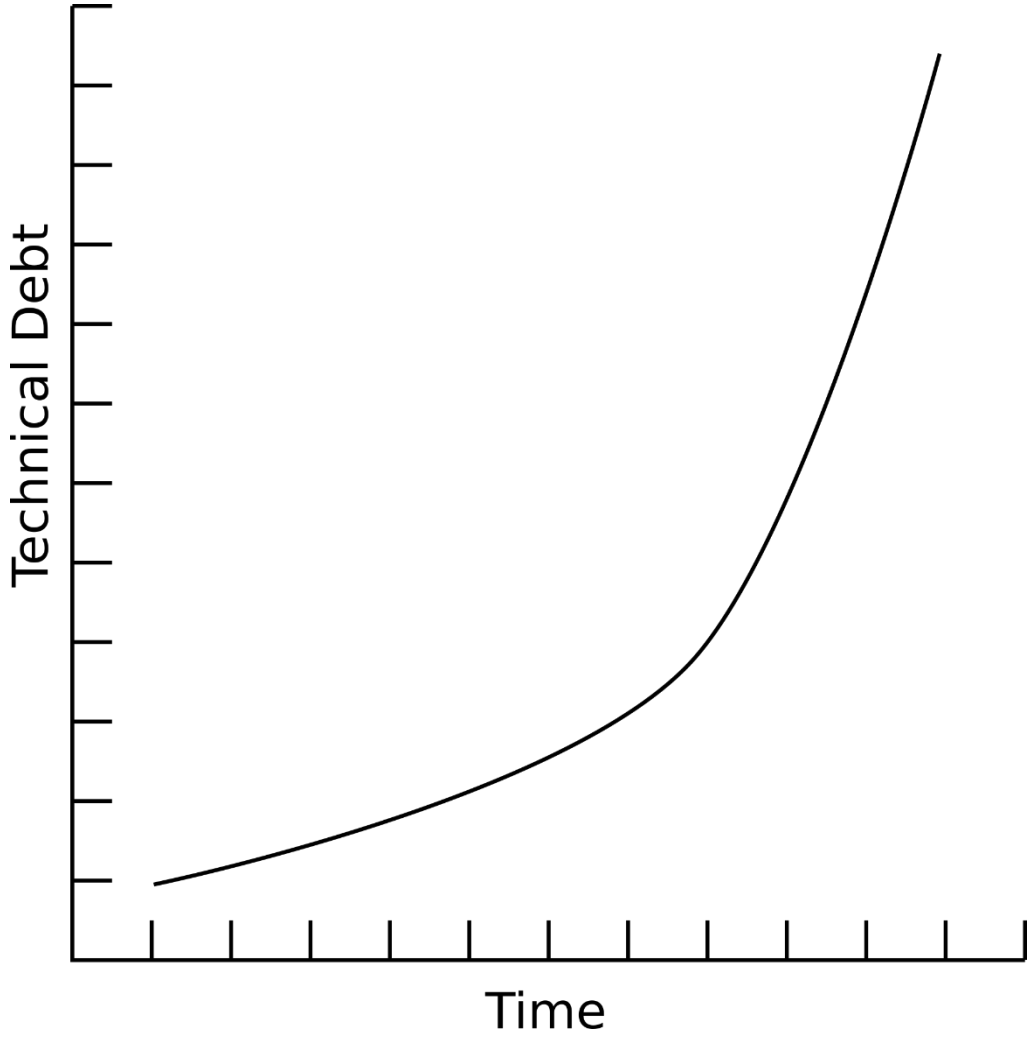


Dev: Can we rewrite?

Man: NO!

Dev: Look, we really need to rewrite.

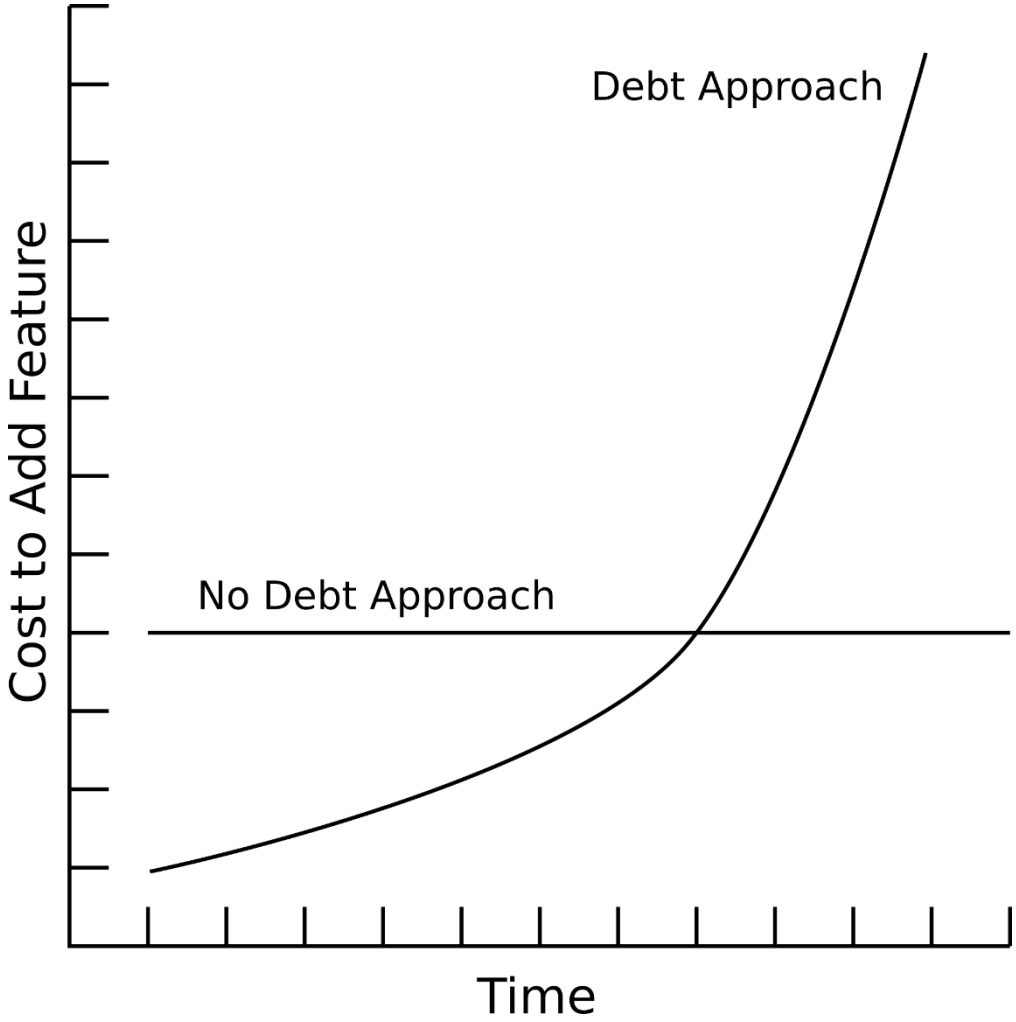
Man: Okay, fine.


Technical Debt
Time

Unchecked Technical Debt

- Grows Exponentially
- Increases time to add new features.
- Eventually cost of adding feature supersedes cost of rewrite.

The greedy algorithm does not win in the long run!


Debt Approach
No Debt Approach
Cost to Add Feature
Time

Technical Debt vs. Software Capital

Technical Debt

- Easy to create
- Cheap
- Reused “by the gun”
- Narrow focus
- Ugly
- Incomplete
- Increases time to market

Software Capital

- Hard to create
- Expensive
- Voluntarily Reused
- Wide Focus
- Beautiful
- Complete
- Decreases time to market

What is Software Capital?

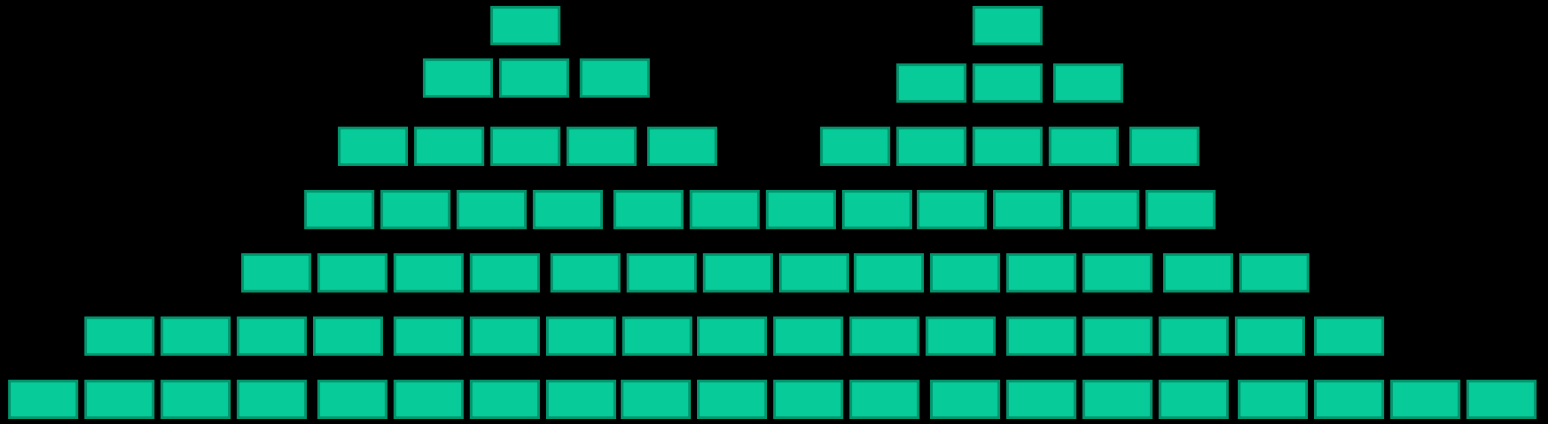
std::vector

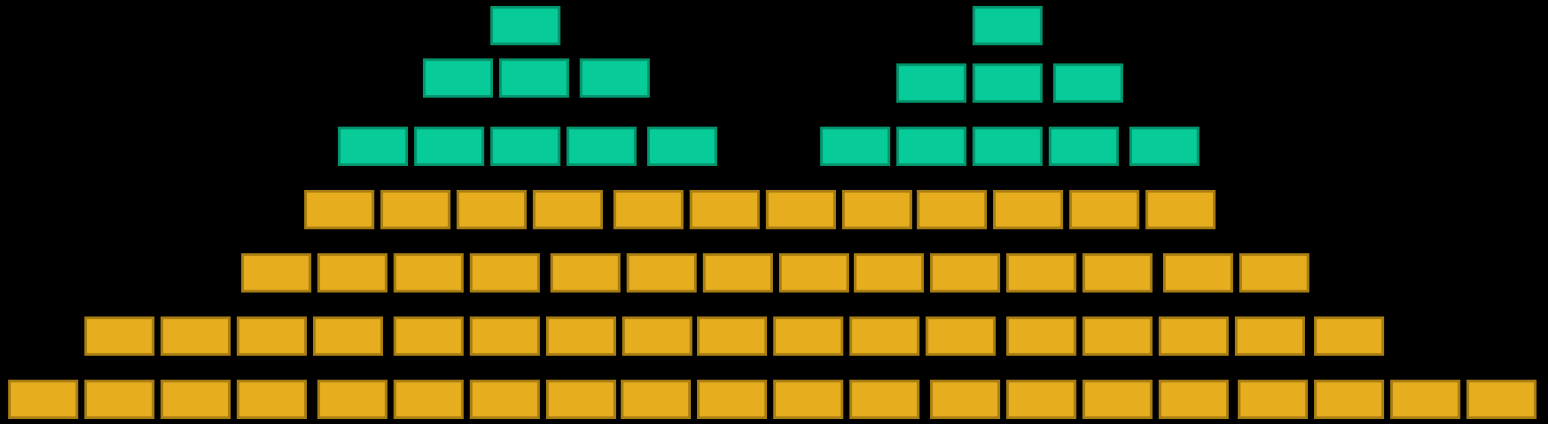
Software Capital

- Useful
- Clean
- Complete
- Reliable
- General
- Documented
- Bug Free
- Efficient
- Reusable

Software Capital

- Dean Zarras coined term in 1996 in Software Capital—Achievement and Leverage. goo.gl/y2EHPs
- Defined as “the cumulative technology that can be re-deployed to new situations”.
- Opposite of technical debt.
- Pays dividends instead of costs interest.





Okay, but how do we build software capital?

Principle 1: Code Reviews

Get an outside opinion.

Code Reviews: Get an outside opinion

- How we see our children vs. how we see others'.
- Is that interface really intuitive?
- One code review will provide 90% of the feedback of a dozen.

Code Reviews: Find Bugs

- Hidden bugs are unaccounted-for technical debt.
- Closer we are to creating the bug, the cheaper it is to fix.

Code Reviews: Standards and Socialization

- Do the best thing everywhere.
  - Interaction creates the standards.
- Intense knowledge transfer.

Code Reviews: Accountability

- Documentation?
- Is this unit tested enough?
- Standards?
- Now two people are responsible for this code.

Code Review Tips

- Choose anyone on the team.
- One round of feedback is usually enough.
- Works extremely well with those who love to learn.
- Tooling can help, but email works.
- “Did you consider…” language.

Principle 2: Standards

Standards: Why?

- Reduced Variance -> Improved Productivity
- Professionalism
- Consistency improves ability to navigate code.
- Makes tooling possible

Standards: What goes in?

- Formatting
- Idioms
- Documentation requirements
- Organization
- Best Practice

Standards: tooling is a must

- Use clang-format and remove formatting as a code-review concern.
- Retrofit old code with new standards whenever feasible.
  - The “code base” is a thing that can be operated on.
  - Use clang as a library to refactor your code.
- clang-tidy to detect and fix certain violations
  - ie. Header guards and include order
  - add common violations

Standards: What criteria?

- Objective criteria always trumps subjective.
- Concentration on reuse. Who is going to reuse this?
- Don’t waste time on trivialities.

Principle 3: Unit Testing

Unit Testing: Why?

- Kill bugs before they cause problems.
- Future proofing against new bugs.
- Gives impression of dependability.
- Safe refactoring.

Unit Testing: Common Excuses

- This is GUI code
  - With a modularized GUI, you can test the pieces.
- This depends on disk/network/etc.
  - Use dependency injection.
- I already know the code is correct
  - We need to be flagged when someone breaks it.
- I need to ship this thing
  - Code needs unit tests to get past code review.

Unit Testing: Design for Testability

```
void retrieveData( const Server & server ) {
  server.connect();
  server.sendRequest( ServerRequest::kRetrieveData,
    []( const Error & error, const Payload & payload ) {
      if( error ) {
        // handle error
      }
      else {
        // verify the payload is correct and handle it if it is.
      }
    }
 );
}
```

Unit Testing: Design for Testability

```
void retrieveData( const Server & server ) {
  server.connect();
  server.sendRequest( ServerRequest::kRetrieveData,
    []( const Error & error, const Payload & payload ) {
      if( error ) {
        // handle error
      }
      else {
        DataParse parseResult = DataParser::parse( payload );
        if( !parseResult.hasError() ) {
          // send on the parsed structure
        }
        else {
          // handle error
        }
      }
    }
 );
```

Unit Testing: Design for Testability

```
// This class implements an abstract mechanism for talking with...
class AbstractServer {
public:
  // Connect to the server. In the exceptional case that there is a problem
  // connecting, throw a ...
  virtual void connect()=0;

  // Send the specified 'request' to the server and call the specified
  // 'callback' with...
  virtual void sendRequest(
    const ServerRequest request,
    std::function<void (const Error&, const Payload&) > callback )=0;
};
```

Unit Testing: Design for Testability

```
class TestServer {
public:
  void connect() override;

  virtual void sendRequest(
    const ServerRequest request,
    std::function<void (const Error&, const Payload&) > callback ) override;

  void throwExceptionOnConnect( bool );
  void setRequestResponseError( const Error& );
  void setRequestResponsePayload( const Error& );
};
```

Unit Testing: Design for Testability

```
TestServer testServer;
testServer.throwExceptionOnConnect(true);
CHECK_THROWS( retrieveData(testServer), ConnectionError );
testServer.setRequestResponseError( /*...*/ );
// etc.
```

Unit Testing: Tooling

- Continuous Integration
- Try Server

Principle 4: Contracts

Contracts: What are they?

What is a car?

Contracts: What are they?

- Precise and complete specification of guaranteed user-visible behavior.
- Excludes implementation detail.
- The “what” and not the “how”.

An example

void sort(std::vector<int> & intVector);

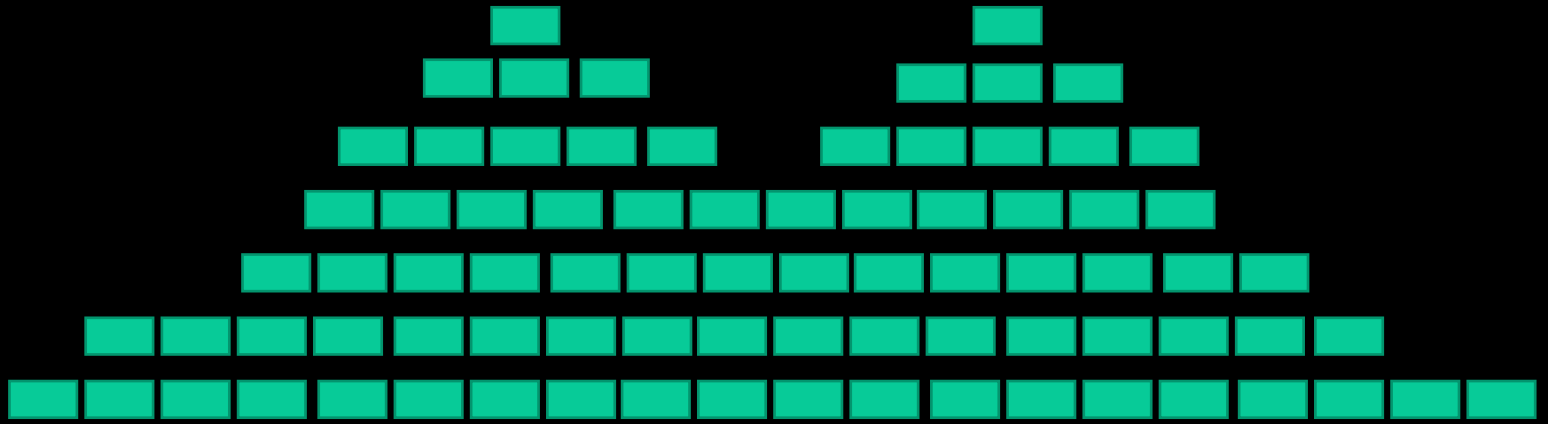
Put the specified ‘intVector’ in order from lowest to highest.

or

Put the specified ‘intVector’ in order from lowest to highest. The algorithm runs in O(n log n) time using O(n) space.

More guarantees imply more use cases.
Fewer guarantees imply more implementation flexibility.

The code is the documentation doesn’t work for large projects



Why Contracts

- Enables objective way to define a bug: Bugs are broken contracts.
- Allows for tiers of abstraction which saves time.
- Enables within-contract improvements to code that is highly reused.
- Provides guidelines for what to unit-test.
- Makes bad interfaces stand out.

Contract Specifications

double sqrt(double number);

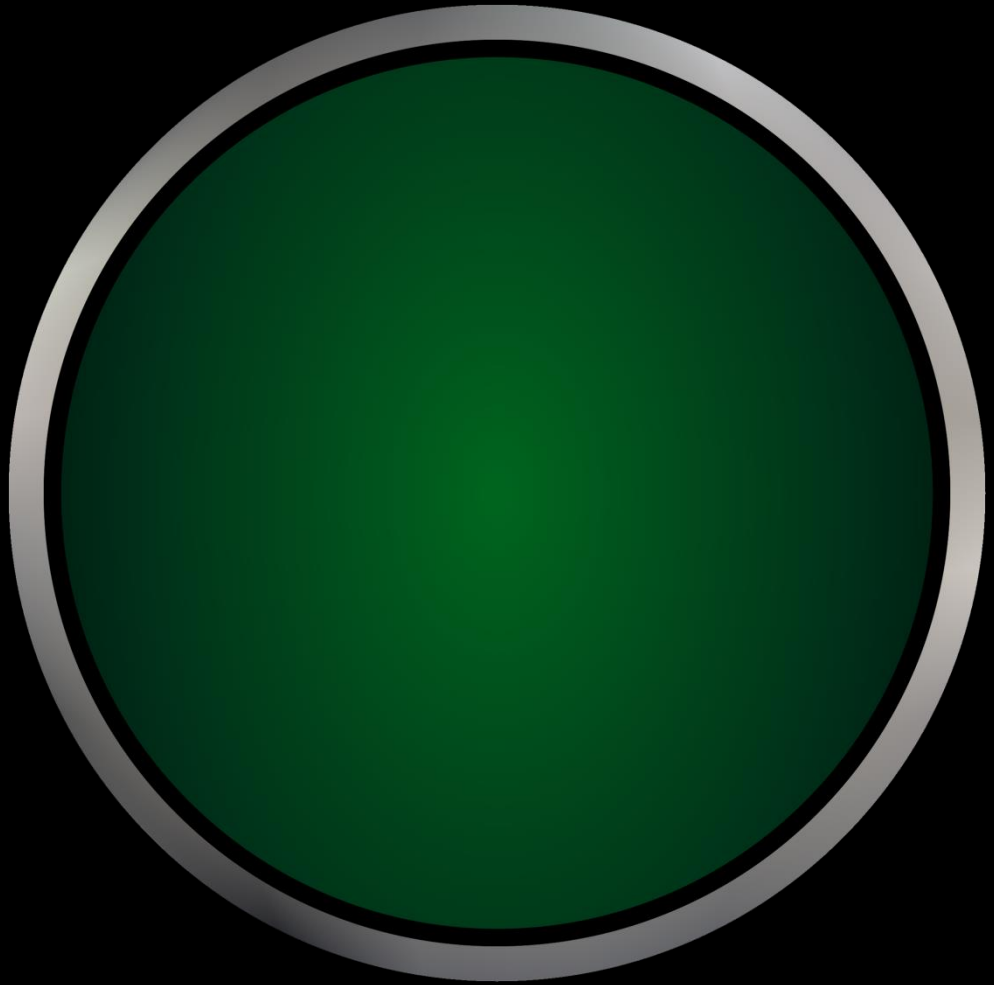
Return the square root of the specified ‘number’. The behavior is undefined unless ‘number > 0.0’.

- Consistent
  - Use a schema
  - Code reviews
- Convenient
  - In the header
  - Separate inline functions
  - Human readable -> No legalese or markup

BDE Contract Specification

- Bloomberg
- Part of BDE coding standards
- https://github.com/bloomberg/bde/wiki/CodingStandards.pdf
- Creative Commons License
- Copy-paste-modify for your company

Principle 5: Good Interfaces

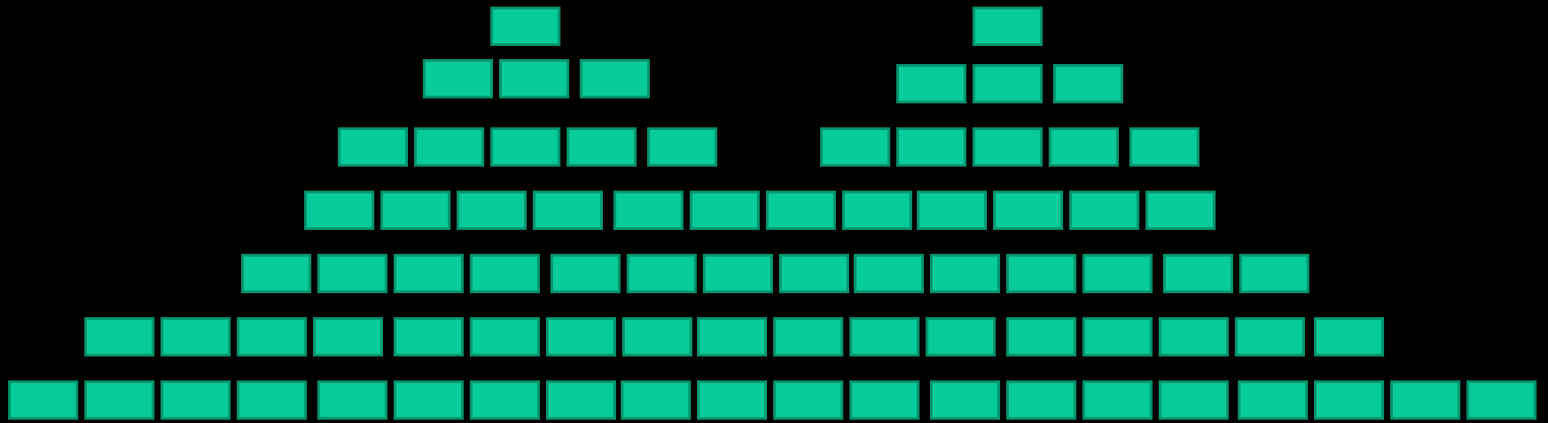


Good Interfaces

- Art Form
- General in the right ways
- Simple contracts
- Manageable pieces
- Built on recognizable patterns
- Naming is important

Math can be a guide to good interfaces!

Organization



Three Levels

- Component
  - .h/.cpp combo
- Package
  - Library
  - Executable
- Package Group

Three Levels

<group>/<package>/<component>.h

<group>/<package>/<component>.cpp

```
namespace <group> {
namespace <package> {
class <component> {
};
}
}
```

- Physical/logical correspondence
- Required include for class is always clear

Logical Organization

- A class's member functions are only those that require private access.
- Other useful functions go in corresponding utility component.

```
// cwf/base/Circle.h

namespace cwf { namespace base{

class Circle {
public:
  double getRadius();
  Point getCenter();
private:
  // etc.
};

}}
```

```
// cwf/base/CircleUtil.h

namespace cwf { namespace base{

class Circle;

class CircleUtil {
public:
  static double circumference( const Circle& );
};

}}
```

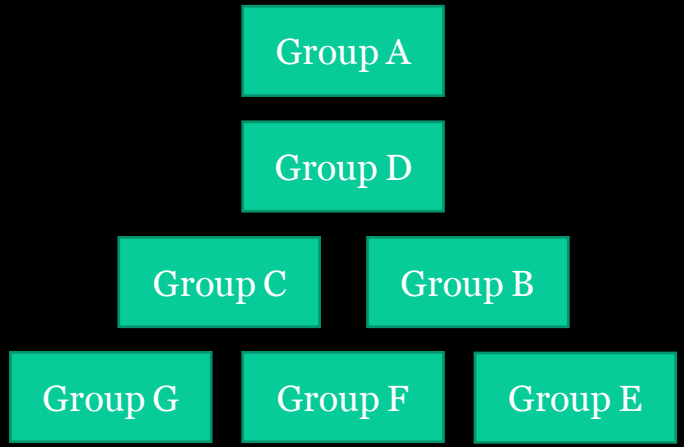
Multiple classes in same component

- Circular dependencies.
- Friends should stick together.
- Use <component>_<piece> as the name for each such class.

Rules for Organization

- Generally one class per component.
- Component consists of a single header and ‘.cpp’ file.
- Packages consist of logically related components with similar dependencies.
- No circular dependencies between components.
- No circular dependencies between packages.
- No circular dependencies between package groups.

What's the big picture for your company?



More on Organization…

Read *Large-Scale C++ Software Design* by John Lakos.

Principle 6: Innovation

Great Innovations

- C++11
  - Smart pointers
  - R-value semantics
  - Lambda functions
- C++17
  - Optional
  - Variant
  - …

The innovation tax

New stuff is great, but…

- It almost always adds complexity.
- Innovation is the cause for code rot.
- If you don’t pay your taxes, you’ll incur technical debt.

Paying the innovation tax

- Training
  - Conferences
  - Books
  - Code Reviews
- Modernizing the Code base
  - A priority
  - Automation can help a lot

Principle 7: Infrastructure

Infrastructure

- Version Control
- Continuous Integration
- Build System
- Core Libraries
- Standards
- Sweeping codebase changes

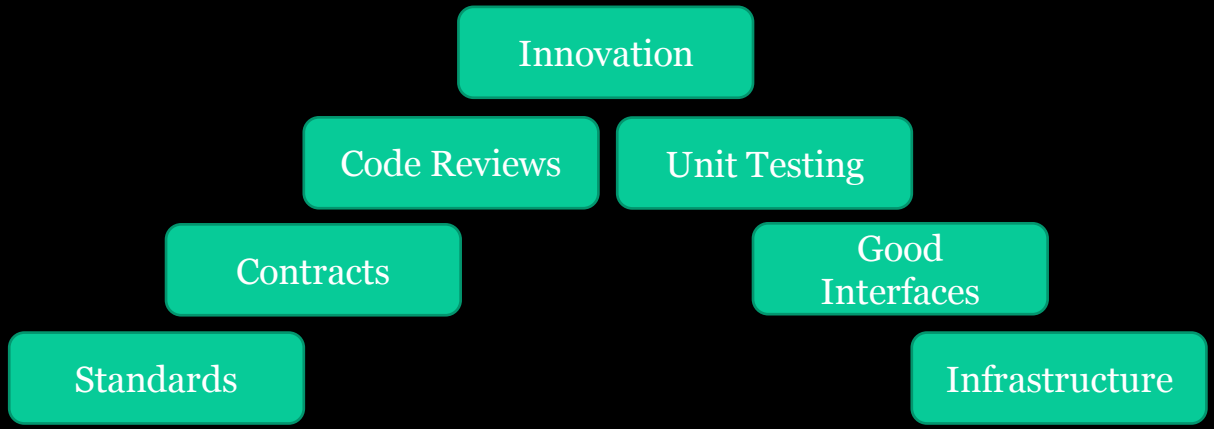
Small Change = Big Impact

Infrastructure: Owning the big picture

- This is not an easy task.
- Highest level of technical expertise required.
- Not a job for the intern.
- Do it right.

Software Capital

- The real value of your organization.
- Key to sustainable competitive advantage.



Bloomberg