C++ & Object Design

Using Test Driven Development

Introductions

- Form groups of approximately 3 – 5 people

- In your groups:

  - Create a list of what you'd like to learn in this class

  - Pick your group's "top 3"

  - For each group member, get a name and one interesting fact

  - Prepare to write your group's "top 3"

  - Prepare to present your group member's name & fact

Schedule

- Start/Stop

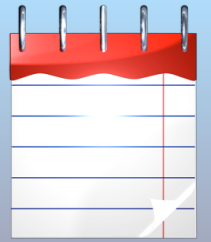- Breaks

- Exercises

Save your fingers

- Do the research yourself...

  - You will learn/retain/understand more if you pair

  - Your productivity will increase

- But...

  - Consider physical configuration

  - Alternate who types

  - Both must be capable of work

  - Don't mix "expert" with "beginner"

Getting Setup

Obligatory Hello World

- Well, not exactly ...

  - Get the environment setup

  - Mechanics

  - Getting your first test running

  - Include directories

  - Library directories and libs

  - Building libs

Your environment

- Your environment should be setup

  - You'll be opening one of two different workspaces

- Details for environment setup are:
  http://schuchert.wikispaces.com/cpptraining

Mechanics: Files...

- We'll need a main (RunAllTests.cpp)

```cpp
#include <CppUTest/CommandLineTestRunner.h>

int main() {
    const char *args[] = { "", "-v" };
    return CommandLineTestRunner::RunAllTests(2, args);
}
```

- We'll need a test (FooShould.cpp)

```cpp
#include <CppUTest/TestHarness.h>

TEST_GROUP(FooShould) {
};

TEST(FooShould, Pass) {
    LONGS_EQUAL(1, 1);
}
```

8

Mechanics: Include & Lib

- The compiler needs to see this include

```
CppUTestBase/include
```

- The compiler needs to see this library directory

```
CppUTestBase/lib
```

- The compiler needs to see this library

```
CppUTest
```

- A more typical main:

```cpp
#include <CppUTest/CommandLineTestRunner.h>

int main(int argc, char **argv) {
    return CommandLineTestRunner::RunAllTests(argc, argv);
}
```
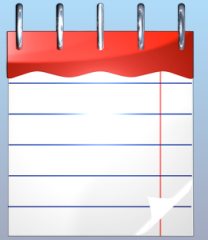
Finally. Run it. Debug It.

- When you run it:

```
.
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)

Press any key to continue . . .
```
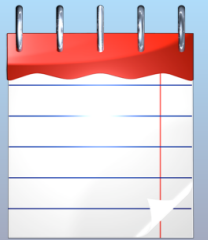
- Now, run in debug, make sure it breaks just before main

Recap

Auto Test Discovery	
Class	
CommandLineTestRunner	
CommandLineTestRunner.h	
Compilation Unit	
Ctrl-B	
Ctrl-F11	
Include Directory	
libCppUTest.a	
Library Directory	
LONGS_EQUAL	

Recap

Object Module	
Order of Includes	
RunAllTests	
standard library	
std	
struct	
template class	
TEST	
TEST_GROUP	
TestHarness.h	
vector	

The Dice Game

13

What's ahead?
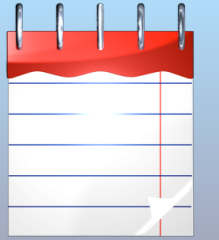
- Creating classes (source and header files)

- Subclassing

- Interfaces

- Guts of C++

- Unit testing

- Test driven development

Project 0: Simple Dice Game

- Simple dice game...

```
┌─────────────┐   1   ┌─────────────┐   2   ┌─────────────┐
│  DiceGame   │──────▶│     Cup     │──────▶│     Die     │
└─────────────┘       └─────────────┘       └─────────────┘
```

- Rules

  - You win 1 if the roll is > 7

  - You push if the roll is 7

  - You lose 1 if the roll is < 7

A Test: DieShould.cpp

01: Must include the class(es) you're using in the test

03: Order of includes might be important here. CppUTest overrides new and delete, which causes problems with classes in the standard library if you use a version prior to 2.3. Include it last.

05 – 06: Define a struct for a test fixture. This creates a base struct (class) with "Die" in its name.

06: You are creating a struct, don't forget the ; at the end

08: Create a test. This is a class that inherits from the struct created on lines 05 - 06. The class name includes "InitialValueInRange1to6" in its name.

10: Test assertion, the expression must evaluate to true or the test will fail. In C/C++ this means 0 or not 0.

```
01: #include "Die.h"
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(DieShould) {
06: };
07:
08: TEST(DieShould, InitiallyBeBetween1And6) {
09:     Die d;
10:     CHECK(
           d.faceValue() >=1 &&
           d.faceValue() <= 6
         )
11: }
```

16

Terms & Concepts

#include with ""	
CHECK	
CppUTest	
struct versus class	
TEST	
TEST_GROUP	
TestHarness.h	

Basics of a Class: Header

01 – 03, 21: Guard against multiple include
01:: If compiler supports, really only include once
02 – 03, 21: When include 2x, only process once

05: Begin class definition – you can because it is followed by { instead of ;

06: Accessible to all clients

07: Declare a method
07: Method is const, meaning it does not change object's state

08: Finish class definition.

08: Don't forget that closing ; – modern compilers can often give you a good error. Or, you might get 100 errors and then the compiler will just give up.

10: Finish off #ifndef at top

```
01:  #pragma once
02:  #ifndef DIE_H_
03:  #define DIE_H_
04:
05:  class Die {
06:  public:
07:    int faceValue() const;
08:  };
09:
10:  #endif
```

18

Terms & Concepts

#define	
#endif	
#ifndef	
#pragma once	
class	
const member function	
Declaration	
Definition	
member function	
public:	

Basics of a Class: Source

01: Include class' header file first (by convention)

03: Define the Die::faceValue method. You know this is a definition because it has a { instead of a ;

03: The test only requires a value between 1 and 6, so this will do for now.

```
01: #include "Die.h"
02:
02: int Die::faceValue() const {
03:     return 1;
04: }
```

Terms & Concepts

Member function definition	
Scope ::	

Create It

- Create a new project: DiceGame

  - Add each of the source files: DieShould.cpp, Die.h, Die.cpp, RunAllTests.cpp

  - Run your program a few times.

- Unit tests should produce no output. Why?

- Feel free to remove:

  - FooShould.cpp

  - VectorShould.cpp

Growing Behavior

- Our Die is hard-coded. We need it to do more. How can we get this to happen?

    - Write tests that force more behavior.

- Production code should become more general as the tests become more specific.

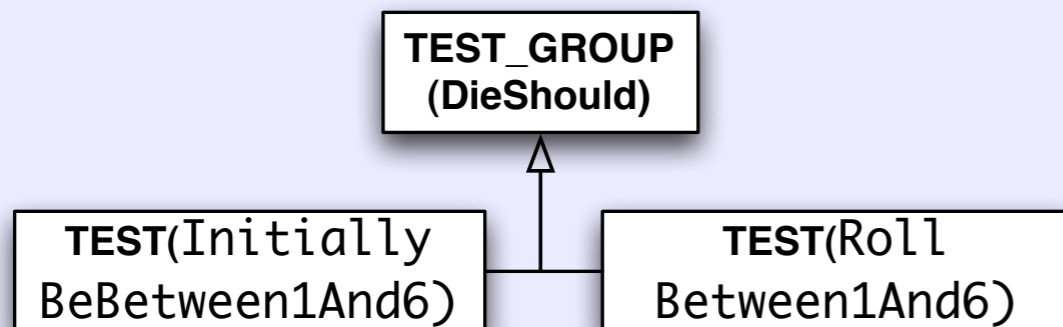# Update DieShould

13: Introduce a new test

14: Notice the duplication?

16: Call a new method, roll.

17 – 18: Make sure the faceValue is between 1 and 6. Do you prefer this over the single-line version on line 10?

14, 17 – 18: Notice the duplication?

Before refactoring this test to remove the duplication (violation of the DRY principle), we should get back to "green".

```
01: #include "Die.h"
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(DieShould) {
06: };
07:
08: TEST(DieShould, InitiallyBeBetween1And6) {
09:    Die d;
10:    CHECK(
          d.faceValue() >=1 &&
          d.faceValue() <= 6
       );
11: }
12:
13: TEST(DieShould, RollBetween1And6) {
14:    Die d;
15:    for(int i = 0; i < 10000; ++i) {
16:       d.roll();
17:       CHECK(d.faceValue() >= 1);
18:       CHECK(d.faceValue() <= 6);
19:    }
20: }
```

```
        TEST_GROUP
        (DieShould)
            △
            │
TEST(Initially      TEST(Roll
BeBetween1And6)     Between1And6)
```

Add Roll to Die

07: Declare a new method. Simply adding this will get the code to compile but not link.

```
01:  #pragma once
02:  #ifndef DIE_H_
03:  #define DIE_H_
04:
05:  class Die {
06:  public:
07:      int roll();
08:      int faceValue() const;
09:  };
10:
11:  #endif
```

02: Define the roll method. Nobody is using the return value. Should this even return a value? (Command-query separation.)

```
01: #include "Die.h"
02: int Die::roll() {
03:    return 9999;
04: }
05:
06: int Die::faceValue() const {
07:    return 1;
08: }
```

25

Update The Die

- Add the test

  - Build it to see the failure.

- Update Die's header file

  - Build to see the linking failure

- Update Die's source file

  - Run it to see the tests pass

- Experiment

  - Remove the ; at the end of class Die{} & build

  - Fix that and then do the same thing with TEST_GROUP

26

DRY Violation

- DRY – Don't Repeat Yourself

- There's a bit of duplication in DieTest

  - We are now going to refactor the test

- Refactoring definition:
  Changing the structure of the code (hopefully improving it) without changing its behavior.

DieShould

06: Add a member field to the TEST_GROUP. This object will be available to each TEST method. Each test method will get a fresh instance of one of these.

07 – 10: Declare a member function that will verify the range of the die. This method will also be available to each TEST method.

Note: How do TEST methods access member fields and member functions? The TEST macro creates a sub-struct of the struct created by TEST_GROUP. Inheritance makes them available.

14, 20: Call the method to verify the die's value.

```cpp
01: #include "Die.h"
02:
03: #include <CppUTest/TestHarness.h>
04:
05: TEST_GROUP(DieShould) {
06:     Die d;
07:     void verifyDieValue() {
08:         CHECK(d.faceValue() >= 1);
09:         CHECK(d.faceValue() <= 6);
10:     }
11: };
12:
13: TEST(DieShould, InitiallyBeBetween1And6) {
14:     verifyDieValue();
15: }
16:
17: TEST(DieShould, RollBetween1And6) {
18:     for(int i = 0; i < 10000; ++i) {
19:         d.roll();
20:         verifyDieValue();
21:     }
22: }
```
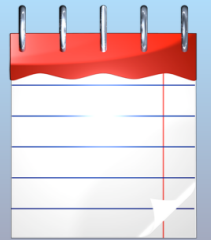
Update DieTest

- Remove the duplication in DieTest.

  - Make sure your tests all pass.

- It is worth keeping the tests clean?

Terms & Concepts

As # tests increases, production code should	
DRY	
Initialization Before Tests	
Keeping Tests Clean	
Order of Tests	
Test Fixture	
Test Fixture Fields	
Test Fixture Methods	

Roll Didn't Do Much

- The roll() method didn't do anything.

    - Is that so bad? We did extend the class' interface

- We need something to improve the implementation.

- What could we do to accomplish that?

    - Write another test!

Value Distribution

02: Use the standard library class array. It holds int and it has 6 elements.

02: Note, this class is in <array>. You'll need to include that header file. This header file can be included anywhere, what is order unimportant?

03: Initialize the contents of the array to 0.

04: Roll the die 600,000 times, each time incrementing the count in the values array with an index equal to the faceValue (-1) by one.

09: Iterate over the entire array.

09: std::array<int, 6>::iterator is long, we'll fix that in a second. In any case, iter is that type. It turns out that it is just a pointer to an int, so it is an index into the array.

10: Initialize iter to the first element in the array

11: So long as you are not at the end of the array, keep going on.

12: Increment the counter. Using pre-increment on iterators is important. It has to do with r-values versus l-values. For now, just always do this.

13 – 14: Make sure the range for each value rolled is within the range 95,000 and 105,000. Can we make this range any tighter?

```cpp
01: TEST(DieShould, DistributeValuesWell) {
02:    std::array<int, 6> values;
03:    values.fill(0);
04:    for(int i = 0; i < 600000; ++i) {
05:      d.roll();
06:      ++values[d.faceValue() - 1];
07:    }
08:
09:  for(std::array<int, 6>::iterator
10:       iter = values.begin();
11:       iter != values.end();
12:       ++iter) {
13:    CHECK(*iter > 95000);
14:    CHECK(*iter < 105000);
15:    }
16: }
```

Wait, that's a lot...

- Here's an improved version we should instead use.

```cpp
#include <array>                                    //  <tr1/array>
typedef std::array<int, 6> RollArray;              // std::tr1::array
typedef RollArray::iterator iterator;

TEST(DieShould, DistributeValuesWell) {
    RollArray values;
    values.fill(0);                                // values.assign(0);
    for (int i = 0; i < 600000; ++i) {
        d.roll();
        ++values[d.faceValue() - 1];
    }

    for (iterator iter = values.begin(); iter != values.end(); ++iter) {
        CHECK(*iter > 95000);
        CHECK(*iter < 105000);
    }
}
```

- The typedef creates synonyms.

Almost Final Header File

07: Introduce a constructor (ctor). This will be called when creating an instance of Die. We'll use it to initialize the member field value.

12: Introduce a member field. This is a primitive so by default it will not be initialized.

```
01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:    Die();
08:    int roll();
09:    int faceValue() const;
10:
11: private:
12:    int value;
13: };
14:
15: #endif
```

34

Almost Final Source File

01: Include class' header file first (by convention)

03: Include other headers used by the implementation. In this case, we are using the new random number generation featured in tr1.

05: Define ctor, using member-wise initialization list

Note: Generally, define constructors (and destructors). Not doing so can lead to larger .o's and longer linking times as the compiler will generate them, probably inline.

08: Using things in the tr1 namespace,. Bring those into the top level scope for this method only.

09: mt19937 – engine producing random numbers.

10: uniform_int<int> attempts to produce an even distribution over the range provided (1, 6) given an engine.

09 – 10: Static on these lines cause a single variable to be created the first time the method is called. Line 9 creates an instance of an engine using the no-arg ctor. Line 10 creates an instance of uniform_int<int> passing in the range 1 – 6. This is a c-style way of getting a single instance.

12: Produce the next random value and store it.

12: Return that value. What about calling faceValue here? Violation of DRY? Command-query separation?

```cpp
01: #include "Die.h"
02:
03: #include <tr1/random>
04:
05: Die::Die() : value(1) {}
06:
07: int Die::roll() {
08:     using namespace std::tr1;
09:     static mt19937 engine;
10:     static uniform_int<int> uniform(1, 6);
11:
12:     return value = uniform(engine);
13: }
14:
15: int Die::faceValue() const {
16:     return value;
17: }
```
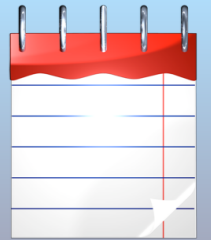
Get it all updated

- Update the test source file

- Update the Die header file
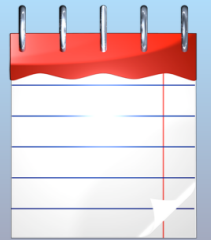
- Update the Die source file

- Was this a big leap?

Terms and Concepts

<tr1/random>	
array	
begin	
command-query separation	
Constructor(ctor)	
end	
iterator	
l-value	
Member field	
mt19937	
namespace	

Terms and Concepts

no-arg ctor	
operator()	
pre-increment	
private:	
r-value	
std::tr1	
template class	
tr1	
typedef	
uniform_int<int>	
using	

Final Header File

08: Dtor, make virtual by default (the tool does).

17: Part of canonical form: hide copy ctor by default
18: Part of canonical form: hide assignment operator by default

Note: "Part of canonical form" – suggests doing something explicit rather than letting the compiler do the work. This can mean making them private and unimplemented OR implementing them.

17, 18: Take in and return Die by reference. A reference cannot be null. In this case, the reference is const, can only call const methods.

18: Die reference is passed in. You can only call const methods (e.g., faceValue).

17 – 18: The limits on the passed in Die are theoretical. We won't be writing code for these methods anyway.

```cpp
01: #pragma once
02: #ifndef DIE_H_
03: #define DIE_H_
04:
05: class Die {
06: public:
07:     Die();
08:     virtual ~Die();
09:
10:     int roll();
11:     int faceValue() const;
12:
13: private:
14:     int value;
15:
16: private:
17:     Die(const Die&);
18:     Die& operator=(const Die&);
19: };
20:
21: #endif
```

Final Version of Source

06: Define dtor — not doing may lead to larger .o's

Generally: Define ctor's and the dtor. Not doing so can lead to larger .o's and longer linking times.

Notice: There are no definitions of the Copy Constructor and the Assignment Operator. That's OK. So long as those methods are never called, the linker will not complain.

If we had not added them to the definition of the class, then those methods would exist and be available.

```cpp
01: #include "Die.h"
02:
03: #include <tr1/random>
04:
05: Die::Die() : value(1) {}
06: Die::~Die() {}
07:
08: int Die::roll() {
09:    using namespace std::tr1;
10:    static mt19937 engine;
11:    static uniform_int<int>
        uniform(1, 6);
12:
13:    return value = uniform(engine);
14: }
15:
16: int Die::faceValue() const {
17:    return value;
18: }
```
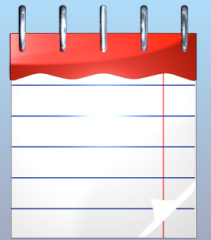
40

Final Die Version

- Make these final updates. While you are at it, add one more test:

```
TEST(DieShould, RollDoublesOftenEnough) {
    int doubles = 0;
    int lastValue = -1;
    for(int i = 0; i < 6000; ++i) {
        d.roll();
        if(d.faceValue() == lastValue)
            ++doubles;
        lastValue = d.faceValue();
    }

    CHECK(doubles > 950);
}
```
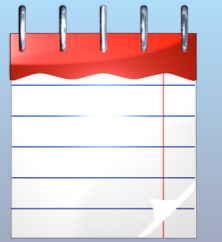
- What do you think of this test?

- What do you think of all the Die tests?

Terms & Concepts

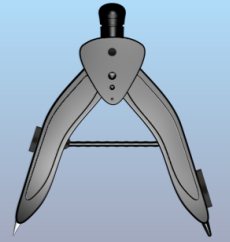|	
Assignment Operator	
Copy Constructor	
Destructor (dtor)	
Operator Overloading	
reference	
virtual	
Virtual dtor	

What's Coming Up?

- Dependency Injection/Inversion of Control

- Test Doubles

- Mechanics of dynamic binding in C++

- (Light)Pointers and references

- Subclassing & command-query separation

- (More On)Standard library iterators
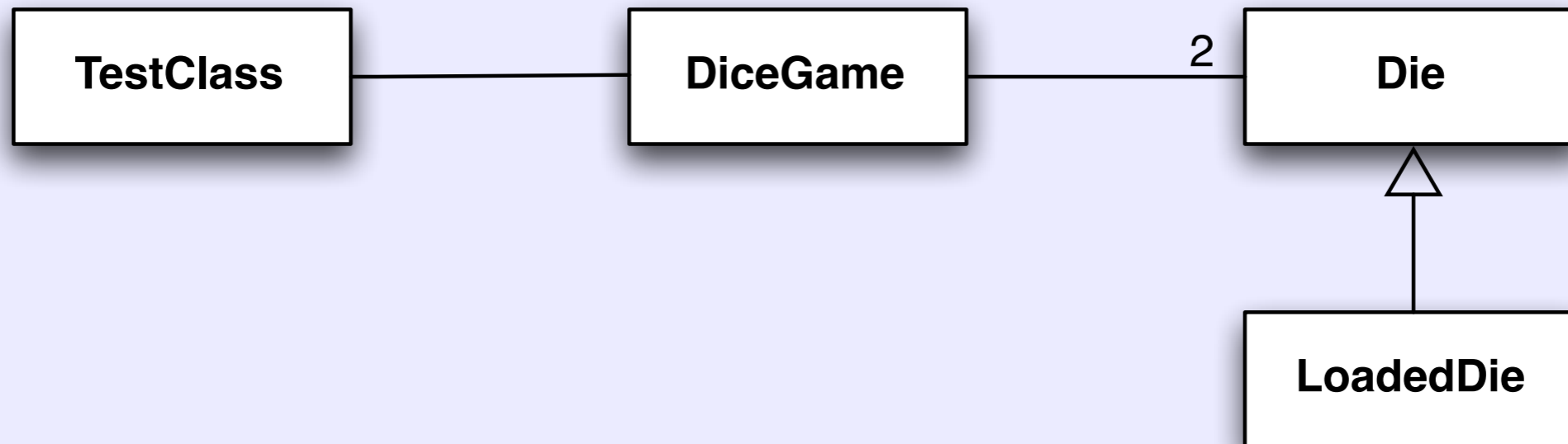
The Game
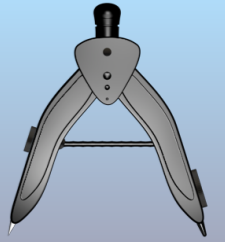
- What of the game's rules? How to control it?

- Write tests confirming the boundary conditions

  - < 7 – lose

  - =7 – push

  - >7 – win
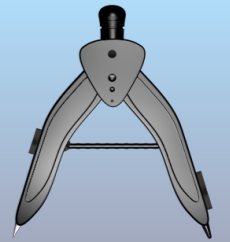
- Dice game uses 2 die objects
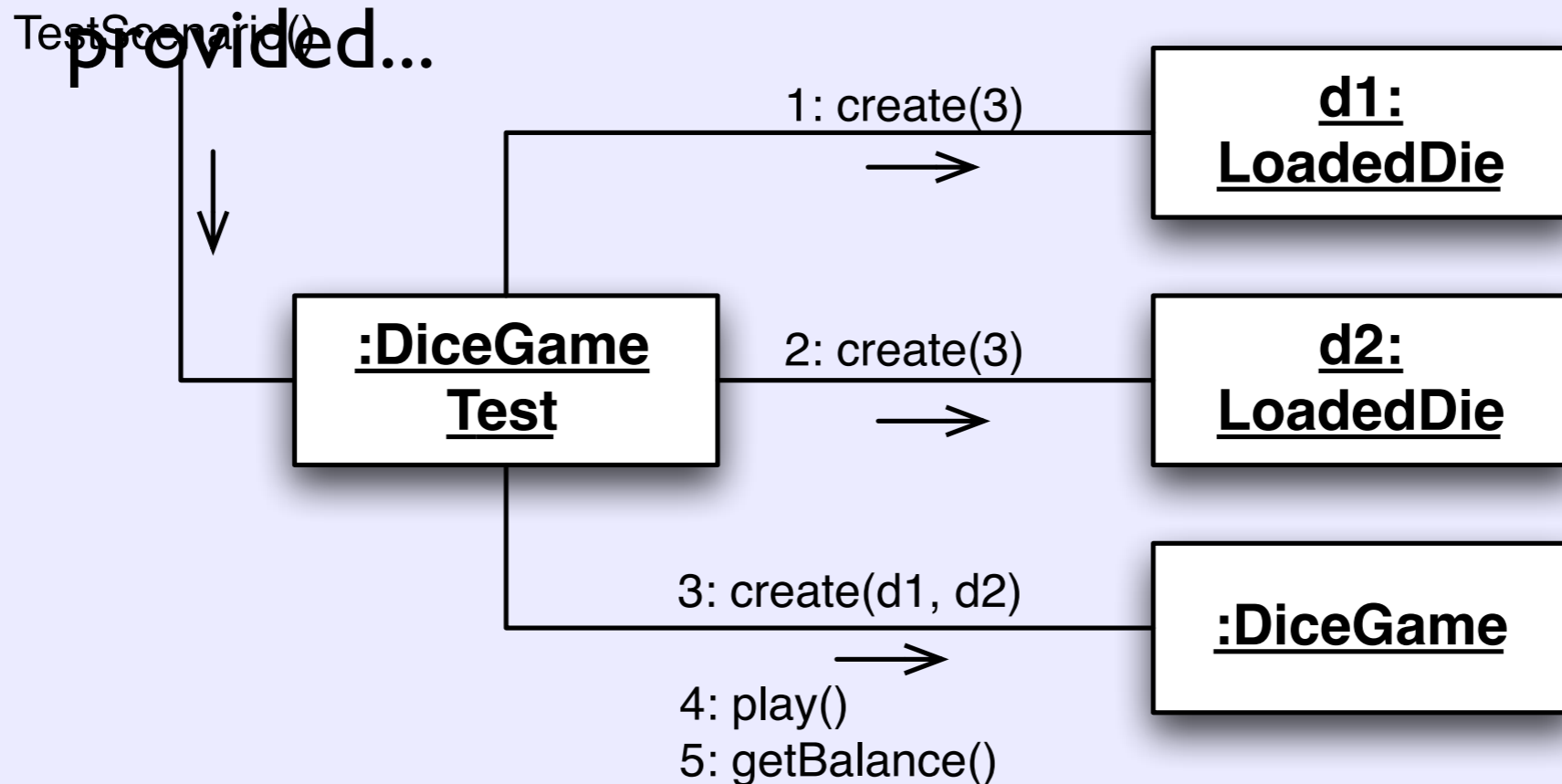
Controlling Test

- What about this?

```
┌──────────────┐          ┌──────────────┐    2    ┌──────────────┐
│  TestClass   │──────────│  DiceGame    │─────────│     Die      │
└──────────────┘          └──────────────┘         └──────────────┘
                                                           △
                                                           │
                                                    ┌──────────────┐
                                                    │  LoadedDie   │
                                                    └──────────────┘
```

- Even so, how can you get DiceGame to use LoadedDie?

Dependency Injection

- Create DiceGame, allow its Die objects to be provided...

TestScenario()

1: create(3) → **d1: LoadedDie**

:DiceGame Test

2: create(3) → **d2: LoadedDie**

3: create(d1, d2) → **:DiceGame**

4: play()
5: getBalance()

- What does this allow?

- How fundamental is this principle?

What are the moving parts?
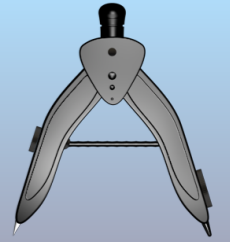
- For this to work

  - Die *should* have a virtual dtor

  - Die *must* have at least one other virtual method

  - That virtual method *must* be overridden by LoadedDie

  - That virtual method *must* be called by DiceGame

  - DiceGame *must* hold either pointers or references to Die

LoadedDieShould

- Let's start with a simple test:

```cpp
#include "LoadedDie.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(LoadedDieShould) {
};

TEST(LoadedDieShould, ReturnValueProvided) {
    LoadedDie die(5);
    CHECK(die.faceValue() == 5);
}
```

- Create this test and the LoadedDie class.

Create LoadedDie

- We have 2 options (we'll eventually use both)...

  - Extract Interface

  - Subclass concrete class

- Subclass concrete class

- What about

  - Copy ctor

  - Assignment op?

```cpp
#pragma once
#ifndef LOADEDDIE_H_
#define LOADEDDIE_H_

#include "Die.h"

class LoadedDie: public Die {
public:
    LoadedDie(int loadedValue);
    int faceValue() const;

private:
    int loadedValue;
};

#endif
```
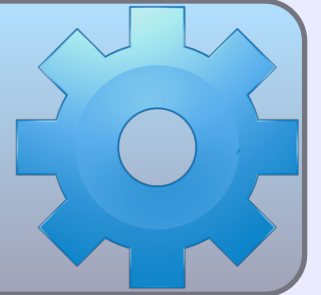
Write LoadedDie

```cpp
#include "LoadedDie.h"

LoadedDie::LoadedDie(int loadedValue)
  : loadedValue(loadedValue) {
}


int LoadedDie::faceValue() const {
  return loadedValue;
}
```

Experiment

- Why does this fail?

```
#include "LoadedDie.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(LoadedDieShould) {
};

TEST(LoadedDieShould, ReturnValueProvided) {
    LoadedDie die(5);
    CHECK(die.faceValue() == 5);
    Die &d = die;
    CHECK(d.faceValue() == 5);
}
```
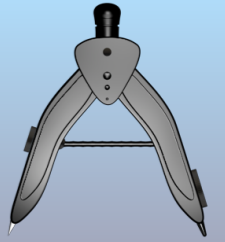
Get it working
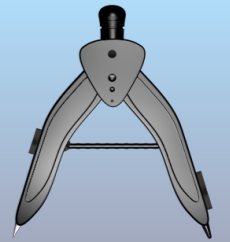
- To fix this, simply add "virtual"

```
class Die {
public:
    Die();
    virtual ~Die();
    int roll();
    virtual int faceValue() const;
}
```

LoadedDie...

- What of overriding faceValue versus roll?

  - How can derived class access value in Die?

  - Issues with subclassing concrete class

  - Command Query Separation – roll returns an int

Review Die Update

- We allowed for polymorphism – dynamic binding:

```cpp
class Die {
public:
    Die();
    virtual ~Die();
    int roll();
    virtual int faceValue() const;
}
```

- Other things to consider

  - Why virtual dtor?

  - Why not virtual roll?

  - What's the danger of one versus the other?

  - How fragile is subclassing a concrete class in general?

  - Command-query separation helps, we'll fix Die accordingly.

Bite the Bullet: Fix Die

- Update roll to have a void return

  - Do you need to update LoadedDie

  - Header files and source files

  - Get the code compiling

  - Get the tests passing

55

What's on Deck?
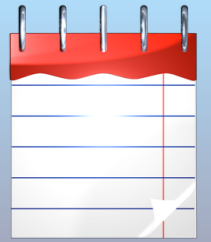
- Test Driven Development

    - Creating DiceGame from scratch

    - Moving from hard-coded to coded with unit tests

    - Refactoring by extracting classes

    - Making test setup easier by changing API's

    - Inversion of Control/Dependency Injection

Test-Driven Walkthrough

- Create a failing test (DiceGameShould.cpp):

```cpp
#include "LoadedDie.h"
#include "DiceGame.h"
#include <CppUTest/TestHarness.h>

TEST_GROUP(DiceGameShould) {};

TEST(DiceGameShould, DecreaseBalanceForLoss) {
    LoadedDie *d1 = new LoadedDie(3);
    LoadedDie *d2 = new LoadedDie(3);
    DiceGame game (d1, d2);
    game.play();
    LONGS_EQUAL(-1, game.getBalance());
}
```

- This does not compile:

  - No DiceGame

  - Several methods need be created...

Work Involved

- Create DiceGame

  - Add missing methods

- Add missing includes, etc.

- Update main()?

58

Dice Game (header)

05: Use the std::vector class to store 2 Die pointers.

06: Forward-declare class rather than include its header file. Reduces compilation time and unnecessary recompilation. Possible because collection holds pointers, not objects.

10: Create a typedef for vector rather than using the raw type. We are putting pointers rather than objects into the vector. Cannot store references (templates cannot use references), must use pointers or references for dynamic binding. Since we are only using pointers, we can simply forward declare the Die class. Prefer forward declaration, even going so far as to change the structure of the class to allow for it. (Can also forward declare for references, but that doesn't solve the problem of not being able to put references in templates.)

11: Promote the nested iterator typedef up one level to shorten the amount of (physical) typing required to use it.

13: Allow the two Die objects to be passed in. While there's nothing to indicate this (yet), the DiceGame will take ownership of the memory associated with the parameters passed in to its ctor.

14: Unless there's a compelling reason to not do it, we'll make our destructors virtual.

15 – 17: The methods used in the test.

20: Define a field, each instance of DiceGame will have its own collection of Die objects, which will actually be std::vector<Die*>

24 – 25: Hide the copy ctor and assignment operator.

```cpp
01:  #pragma once
02:  #ifndef DICEGAME_H_
03:  #define DICEGAME_H_
04:
05:  #include <vector>
06:  class Die;
07:
08:  class DiceGame {
09:  public:
10:     typedef std::vector<Die*> DiceCollection;
11:     typedef DiceCollection::iterator iterator;
12:
13:     DiceGame(Die *d1, Die *d2);
14:     virtual ~DiceGame();
15:     void play();
16:     int getBalance() const;
17:
18:  private:
19:     DiceCollection dice;
20:     int balance;
21:
22:  private:
23:     DiceGame(const DiceGame&);
24:     DiceGame& operator=(const DiceGame&);
25:  };
26:
27:  #endif
```

Dice Game (cpp)

01: By convention, include class' header file first.

02: The header file forward declares Die. The source file uses the class, so Die's header file must be included.

04: Use member-wise initialization to set balance to 0. Otherwise it would be uninitialized (by definition).

05 – 06: Add the two die objects into the vector

10: Iterate over each element of the collection of Die objects. This is a standard form. Start at begin(), continue while not at end(). Pre-increment (do not post-increment) current. Why?

11: current is an iterator, which, for a vector, is simply a pointer to what the vector contains. The vector contains Die*, so iterator is actually Die**. Dereference it to get back a Die*, which then is deleted.

05, 06, 10: How does the method know which instance of DiceGame it's using?

```cpp
01: #include "DiceGame.h"
02: #include "Die.h"
03:
04: DiceGame::DiceGame(Die*d1, Die*d2)
        :balance(0) {
05:   dice.push_back(d1);
06:   dice.push_back(d2);
07: }
08:
09: DiceGame::~DiceGame() {
10:   for(iterator current = dice.begin();
      current != dice.end();
      ++current)
11:     delete *current;
12: }
```

Dice Game (cpp)

We only have one test, this is all we need to get that one test passing.

How are we going to improve the implementation? Write more tests.

```cpp
13: void DiceGame::play() {
14:     --balance;
15: }
16:
17: int DiceGame::getBalance() const {
18:     return balance;
19: }
```

61

Get it all Working

- You have quite a bit of work ahead of you:

  - Create the test source file

  - Create DiceGame class

  - Get it all compiling and the one new test passing

- Review your work, any duplication?

Improving Implementation

- Always loosing is not much fun...

- Adding a new test can help fix that:

```
TEST(DiceGameShould, IncreaseBalanceForWin) {
  // ...
}
```

- Then get both tests to pass at same time.

- Add another:

```
TEST(DiceGameShould, LeaveBalanceAloneForPush) {
  // ...
}
```

- And then make sure all three tests are passing.

Hints...

- When you write a test for balance increasing, you'll need to roll the dice and get the sum:

```
for(iterator i = dice.begin(); i != dice.end(); ++i)
    (*i)->roll();
```
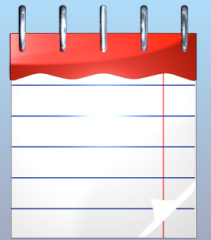
```
int total = 0;
for(iterator i = dice.begin(); i != dice.end(); ++i)
    total += (*i)->faceValue();
```

Now Experiment

- Here are a few things to try. For each, make the change, run your tests, note the results and then restore. Run each of these independently (use RCS if you have it)

1. Delete the body of DiceGame::~DiceGame

2. Remove virtual from the declaration of Die::~Die

3. Remove virtual on the declaration of Die::faceValue()

4. Remove "const" on the declaration & definition of LoadedDie::faceValue

5. Add virtual to declaration of Die::roll()

- Record your observations for each of these

65

Observations

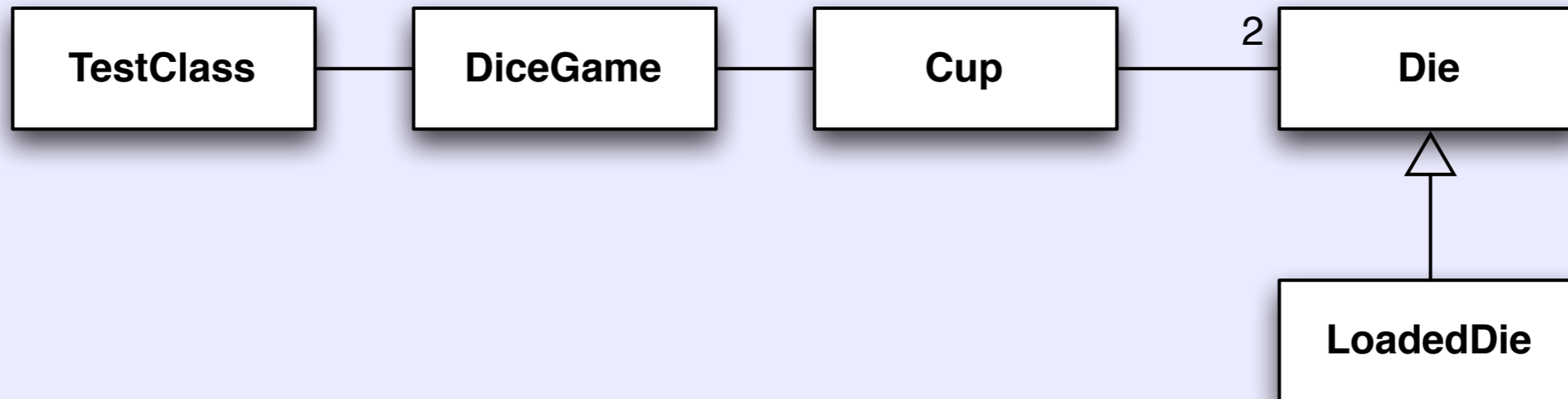|	
Delete body of DiceGame::~DiceGame	
Remove virtual from Die::~Die	
Remove virtual from Die::faceValue()	
Remove const on LoadedDie::faceValue	
Add virtual to Die::roll()	

What's Coming Up?
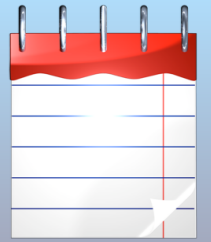
- Refactoring

- Extract class refactoring

- shared_ptr

- Factory

- Dependency Inversion Principle

What of the Cup Class?

- So far, no Cup class. Will it improve our code?

```
┌───────────┐     ┌───────────┐     ┌───────────┐   2  ┌───────────┐
│ TestClass │─────│ DiceGame  │─────│    Cup    │──────│    Die    │
└───────────┘     └───────────┘     └───────────┘      └───────────┘
                                                             △
                                                             │
                                                       ┌───────────┐
                                                       │ LoadedDie │
                                                       └───────────┘
```

- We'll probably want to change the DiceGame ctor.

- All the collection logic will be pushed into it

- Would change the test a bit.

- How can we slowly migrate to it?

Refactoring

- Refactoring – Changing the structure of code without changing its behavior.

    - We might use tests to make sure we don't change the behavior.

    - Though our current test suite is maybe a bit weak.

    - Even so, let's give it a try.

- We're going to several seemingly unnecessary steps:

    - Keep the code compiling most of the time.

    - Keep the tests passing most of the time.

Refactoring: Cup Header

- ## Create Cup Header:

```cpp
#pragma once
#ifndef CUP_H_
#define CUP_H_

#include <vector>
class Die;

class Cup {
public:
  typedef std::vector<Die*> DiceCollection;
  typedef DiceCollection::iterator iterator;
  typedef DiceCollection::const_iterator const_iterator;

  Cup(Die *d1, Die *d2);
  virtual ~Cup();

  void roll();
  int total() const;

private:
  DiceCollection dice;

private:
    Cup(const Cup&);
    Cup& operator=(const Cup&);
};

#endif
```

70

Refactoring: Cup Source

This code is copied (not moved) from DiceGame

By using "extract class" or many little "extract methods" operations, you can safely refactor since you are mostly maintaining the structure of the code.

We've already seen a "const method", when you iterate over a standard C++ collection in a const method, you need to use a const_iterator instead of a regular iterator.

Once you have the header and source, verify that your system still compiles and your tests still pass.

```cpp
#include "Cup.h"
#include "Die.h"

Cup::Cup(Die *d1, Die *d2) {
  dice.push_back(d1);
  dice.push_back(d2);
}

Cup::~Cup() {
  for(iterator current = dice.begin();
      current != dice.end();
      ++current)
    delete *current;
}

void Cup::roll() {
  for(iterator current = dice.begin();
      current != dice.end();
      ++current)
    (*current)->roll();
}

int Cup::total() const {
  int total = 0;

  for(const_iterator current = dice.begin();
      current != dice.end();
      ++current)
    total += (*current)->faceValue();

  return total;
}
```

Get to Compiling

- You've added code that's a copy of existing code

    - Get your code compiling

    - Make sure your tests still run

- No new code, so this is a restructuring

Question: should you write unit tests for it?
Your Answer:

Update DiceGame, Part 1

Update the header file to use Cup instead of std::vector. New and changed lines larger than unchanged lines.

As you are doing this, ask yourself, could we have taken a smaller step first?

Notice that this is header is smaller? Makes sense, much of the work has been done in Cup instead of DiceGame.

```cpp
#pragma once
#ifndef DICEGAME_H_
#define DICEGAME_H_

class Die;
class Cup;

class DiceGame {
public:
    DiceGame(Die *d1, Die *d2);
    virtual ~DiceGame();

    void play();
    int getBalance() const;

private:
    Cup *dice;
    int balance;

private:
    DiceGame(const DiceGame&);
    DiceGame& operator=(const DiceGame&);
};

#endif
```
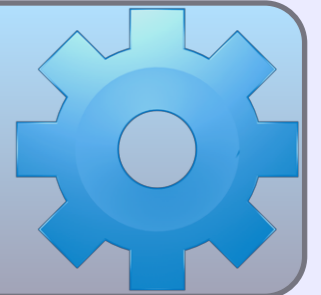
Update DiceGame, Part II

Dice Game forward declared Cup, so it must include Cup.h if it plans to use it.

Notice the order of the member-wise initialization list? What determines the order of execution?

Notice simplified dtor? The work is the same, we've pushed it into the Cup class.

How about play()? It's pretty clear. How about taking the conditionals and moving them into a private method called determineResult()?

```cpp
#include "DiceGame.h"
#include "Cup.h"

DiceGame::DiceGame(Die *d1, Die *d2)
  : dice(new Cup(d1, d2)), balance(0) {
}

DiceGame::~DiceGame() {
  delete dice;
}

void DiceGame::play() {
  dice->roll();

  if(dice->total() < 7)
    --balance;
  if(dice->total() > 7)
    ++balance;
}

int DiceGame::getBalance() const {
  return balance;
}
```

Taking Small Steps

- We extracted a class by

  - First copying code into new class

  - Getting it to compile

  - Updating original code to use new code

  - Keeping DiceGame's public API the same

- Is there a "next step" or are we done?

Memory Allocation

- Problem

  - How can a client know that the Cup ctor expects dynamically allocated objects?

  - What of the Cup class calling delete in its dtor?

An Option

- std::shared_ptr

  - Shared ownership of pointers

  - Multiple, reference-counted

  - When copies, reference count increased

  - Will fail with circular references

77

Failing Example

- Memory leak...

```cpp
#include <CppUTest/TestHarness.h>

TEST_GROUP(shared_ptr) {
};

TEST(shared_ptr, FailsWithout) {
    int *v = new int;
}
```

```
TEST(shared_ptr, FailsWithout)

../SharedPtrExample.cpp:6: error: Failure in TEST(shared_ptr,
FailsWithout)

    Memory leak(s) found.

Leak size: 4 Allocated at: ../SharedPtrExample.cpp and line: 7. Type:
"new" Content: "
"

Total number of leaks:  1
```

Fixed...

- Using an std::shared_ptr:

```cpp
#include <memory>

#include <CppUTest/TestHarness.h>

TEST_GROUP(shared_ptr) {
};

TEST(shared_ptr, Fixed) {
  std::shared_ptr<int> v(new int);
}
```

```cpp
#include <shared_ptr.h>
```

```
OK (9 tests, 9 ran, 20020 checks, 0 ignored, 0 filtered out, 112 ms)
```

Fix DiceGame...

- ## Update Header:

```cpp
#include <memory>

class DiceGame {
  ...


private:
  std::shared_ptr<Cup> dice;
```

- ## Update Source:
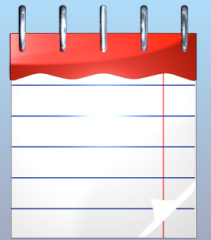
```cpp
DiceGame::~DiceGame() {
}
```

Update Cup...

- Fix Cup to use std::shared_ptr

  - Update all the necessary files

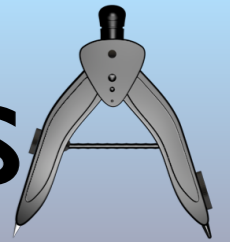 - Remember to remove body of dtor in Cup

  - Get your tests running:

```
typedef std::shared_ptr<Die> sp_Die;
typedef std::vector<sp_Die> DiceCollection;
dice.push_back(sp_Die(aDie));
```
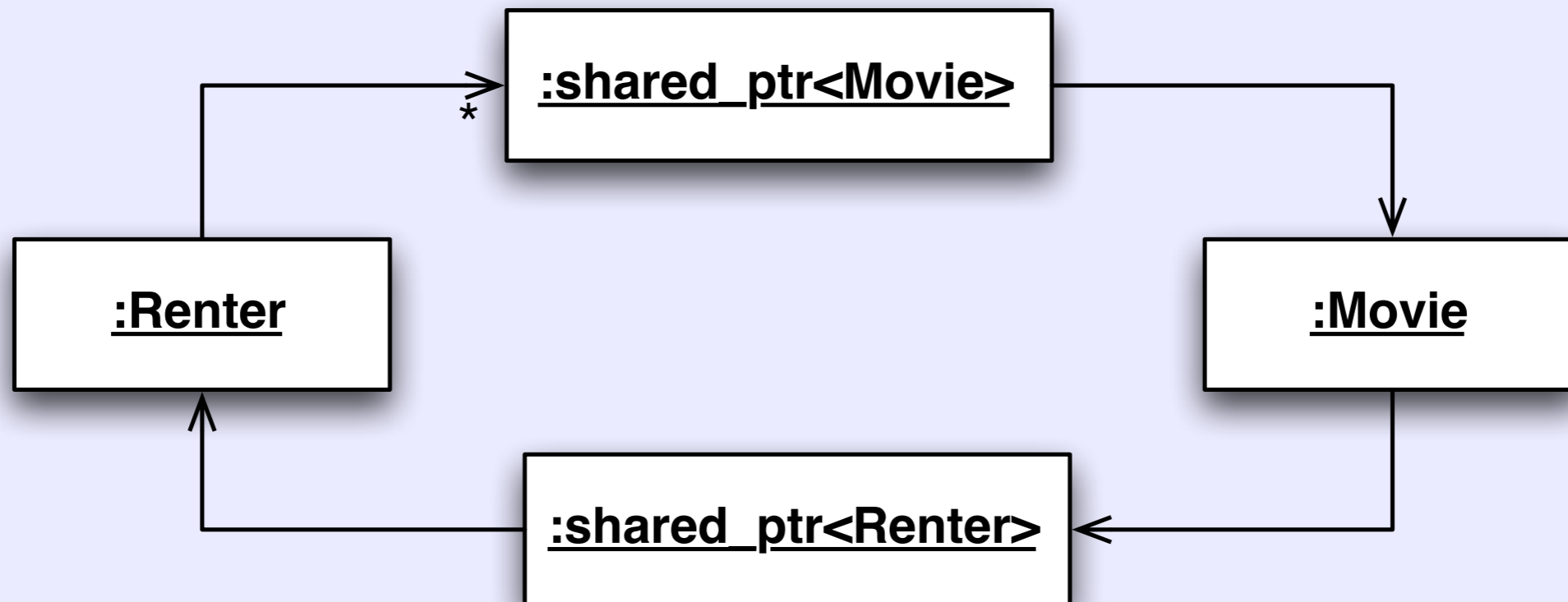
81

Check List

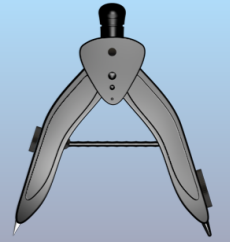|	
Extract class	
std::shared_ptr	

Warning: Circular References

- shared_ptr<>'s fail on circular references:

Here's a failing example

```cpp
#include <memory>

struct Movie;
typedef std::shared_ptr<Movie> sp_Movie;

struct Renter {
  sp_Movie movie;
};

typedef std::shared_ptr<Renter> sp_Renter;
struct Movie {
  sp_Renter checkedOutBy;
};
```
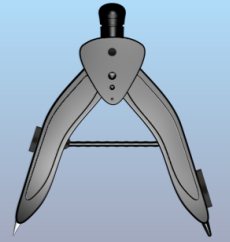
```cpp
struct RentAMovieSystem {
  sp_Renter createRenter() {
    return sp_Renter(new Renter);
  }
  void rentAnyMovieTo(sp_Renter &renter) {
    Movie *someMovie = new Movie;
    renter->movie.reset(someMovie);
    someMovie->checkedOutBy = renter;
  }
};
```

```cpp
#include <CppUTest/TestHarness.h>

TEST_GROUP(CircularReference) {
};

TEST(CircularReference, Broken) {
  RentAMovieSystem system;
  sp_Renter renter = system.createRenter();
  system.rentAnyMovieTo(renter);
}
```
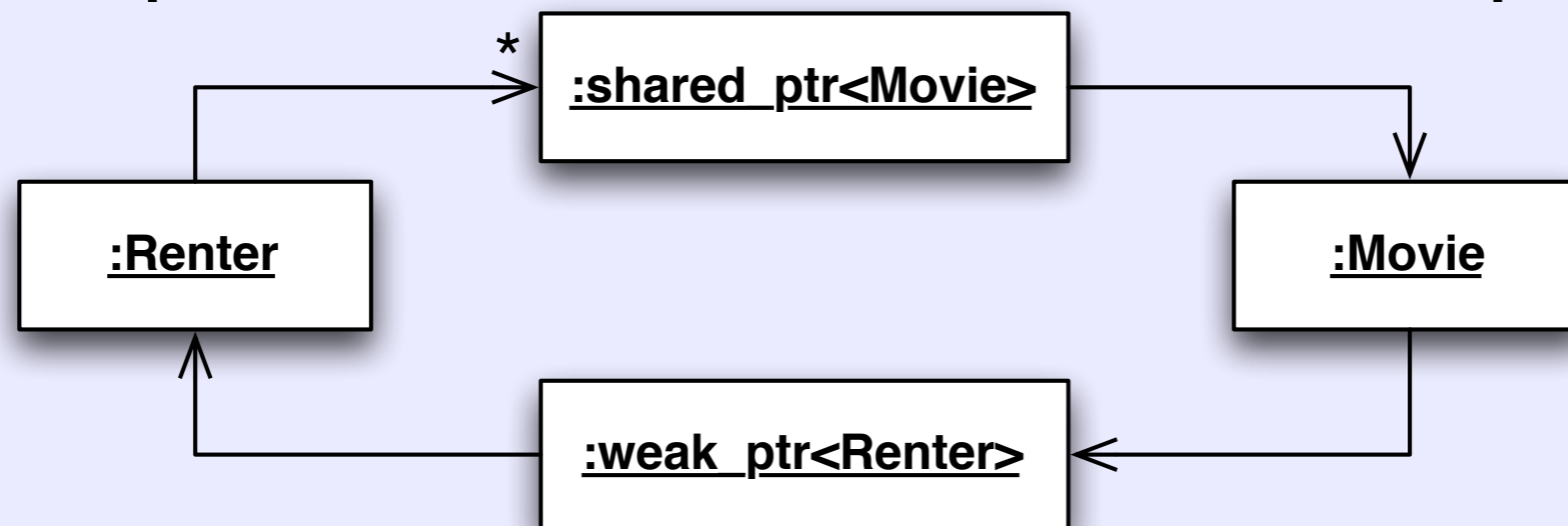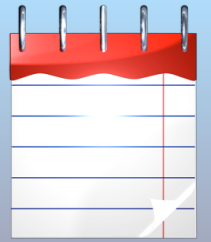
Possible Solutions

- Option 0: Remove Circular Reference.

- Option 1: Use Weak Pointer On "Dependent" Side

```
          *  ┌──────────────────────┐
        ┌───▶│ :shared_ptr<Movie>   │─────────┐
        │    └──────────────────────┘         ▼
┌───────────┐                          ┌───────────┐
│ :Renter   │                          │ :Movie    │
└───────────┘                          └───────────┘
        ▲    ┌──────────────────────┐         │
        └────│ :weak_ptr<Renter>    │◀────────┘
             └──────────────────────┘
```

- Option 2: Use Normal Pointer Instead

```
          *  ┌──────────────────────┐
        ┌───▶│ :shared_ptr<Movie>   │─────────┐
        │    └──────────────────────┘         ▼
┌───────────┐                          ┌───────────┐
│ :Renter   │◀─────────────────────────│ :Movie    │
└───────────┘                          └───────────┘
```
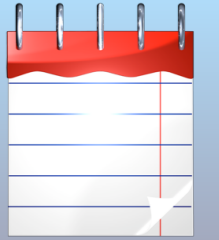
std::weak_ptr

- Weak pointers can be set from shared pointers:

```
struct Movie {
  std::weak_ptr<Renter> checkedOutBy;
};
```

- Note, for this to work:

  - Only one shared_ptr on one new Renter

  - That one shared_ptr used to create weak reference

- Can you change design to remove circularity?
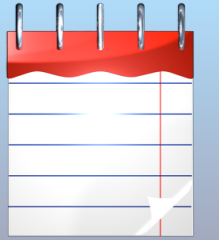
Raw Pointer

- Update struct:

```
struct Movie {
  Renter* checkedOutBy;
};
```

- And assignment:

```
void rentAnyMovieTo(sp_Renter &renter) {
  Movie *someMovie = new Movie;
  renter->movie.reset(someMovie);
  someMovie->checkedOutBy = renter.get();
}
```

What's Coming Up?

- for_each

- accumulate

- bind to call member functions

- pointers to member functions

- _1

88

A Few Built-In Algorithms

- Remember this:

```cpp
void Cup::roll() {
   for(iterator current = dice.begin();
        current != dice.end();
        ++current)
     (*current)->roll();
}
```

- How about this instead?

```cpp
#include <algorithm>

static void rollIt(Cup::sp_Die &die) {
   die->roll();
}

void Cup::roll() {
   std::for_each(dice.begin(), dice.end(), rollIt);
}
```

And total()...

- ## Instead of this...

```cpp
int Cup::total() const {
  int total = 0;

  for(const_iterator current = dice.begin();
      current != dice.end();
      ++current)
    total += (*current)->faceValue();

  return total;
}
```
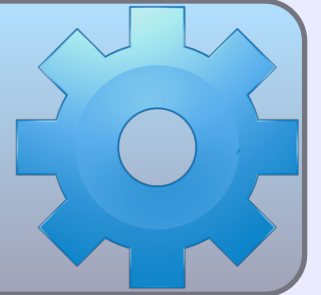
- ## Try this...

```cpp
#include <numeric>
static int sumIt(int currentSum, const Cup::sp_Die &die) {
    return currentSum + die->faceValue();
}


int Cup::total() const {
 return std::accumulate(dice.begin(), dice.end(), 0, sumIt);
}
```

Update Cup

- Use for_each and accumulate

- Compare & contrast

    - Do you prefer one over the other?

    - What about the file-level functions?

Try This Instead

- Having to write a method that calls through to a member method is a hassle. Thus, <tr1/functional>:

```cpp
#include <tr1/functional>

using namespace std::tr1;
using namespace std::tr1::placeholders;

void Cup::roll() {
  std::for_each(
      dice.begin(),
      dice.end(),
      bind(&Die::roll, _1)
  );
}
```

What of accumulate?

- We can also update total:

```cpp
int Cup::total() const {
  return std::accumulate(
    dice.begin(),
    dice.end(),
    0,
    std::tr1::bind(
      std::plus<int>(),
      _1,
      bind(&Die::faceValue, _2)
    )
  );
}
```

- You may be thinking &^!@#???

- http://schuchert.wikispaces.com/cpptraining.SummingAVector

Check List

_1, _2	
<algorithm>	
<numeric>	
accumulate	
bind	
for_each	
function pointer	
Pointer to member function	
std::tr1::placeholders	

94

Time Check

- The remainder of this is covered: http://schuchert.wikispaces.com/cpptraining.CppAndOodTheLeastYouNeedToKnow

- Depending on time, we'll probably switch to the next project.
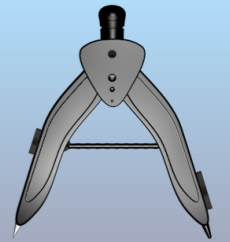
Making Test Writing Easier?

- Hide game internals a bit...

startTest()

1: create()

f: DieFactory

:TestClass

2.1: d1 := build()
2.2: d2 := build()

g: DiceGame

2: create(f)

2.3.1: create();
2.3.2: push_back(d)
2.3.3: push_back(d)

:Cup

vector<sp_Die>

2.3: create(d1, d2)
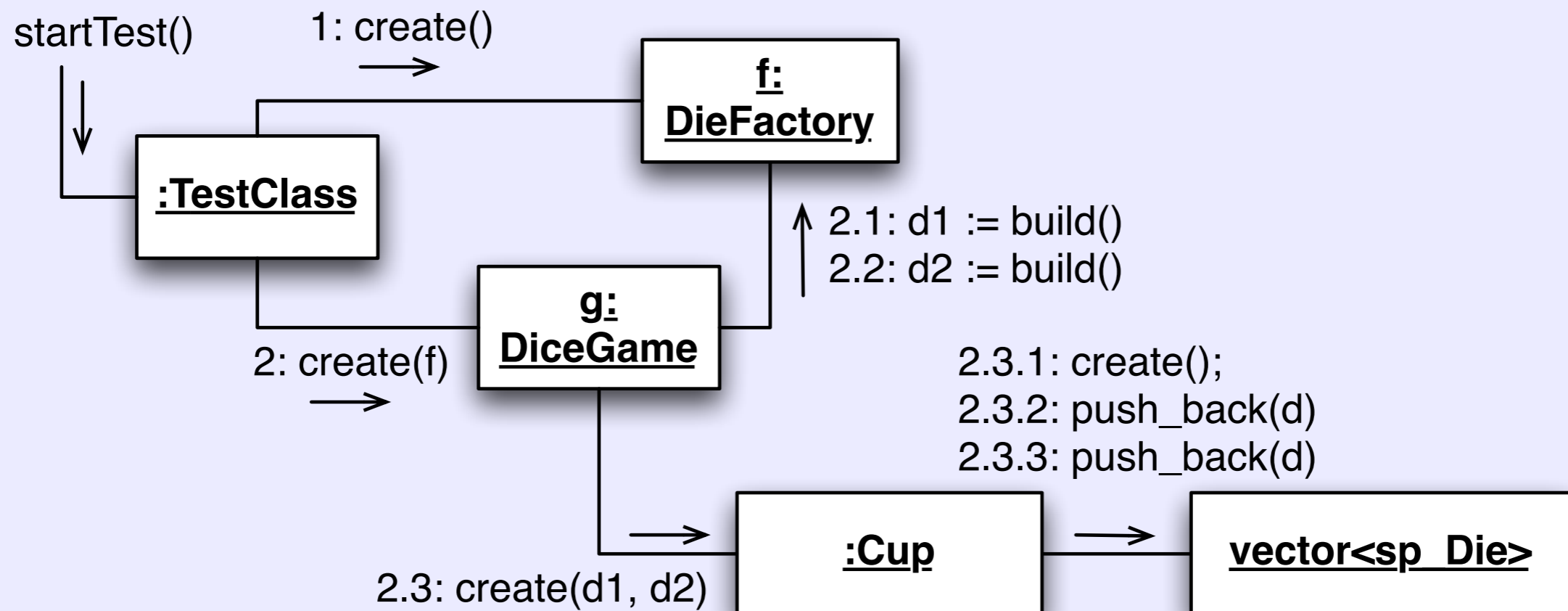
Oh Wait... Testability

- We still need loaded dice...

- Test will select which factory

- Factory selects die

- DiceGame gets dice from factory

IDieFactory

DieFactory

LoadedDie Factory

Die

LoadedDie
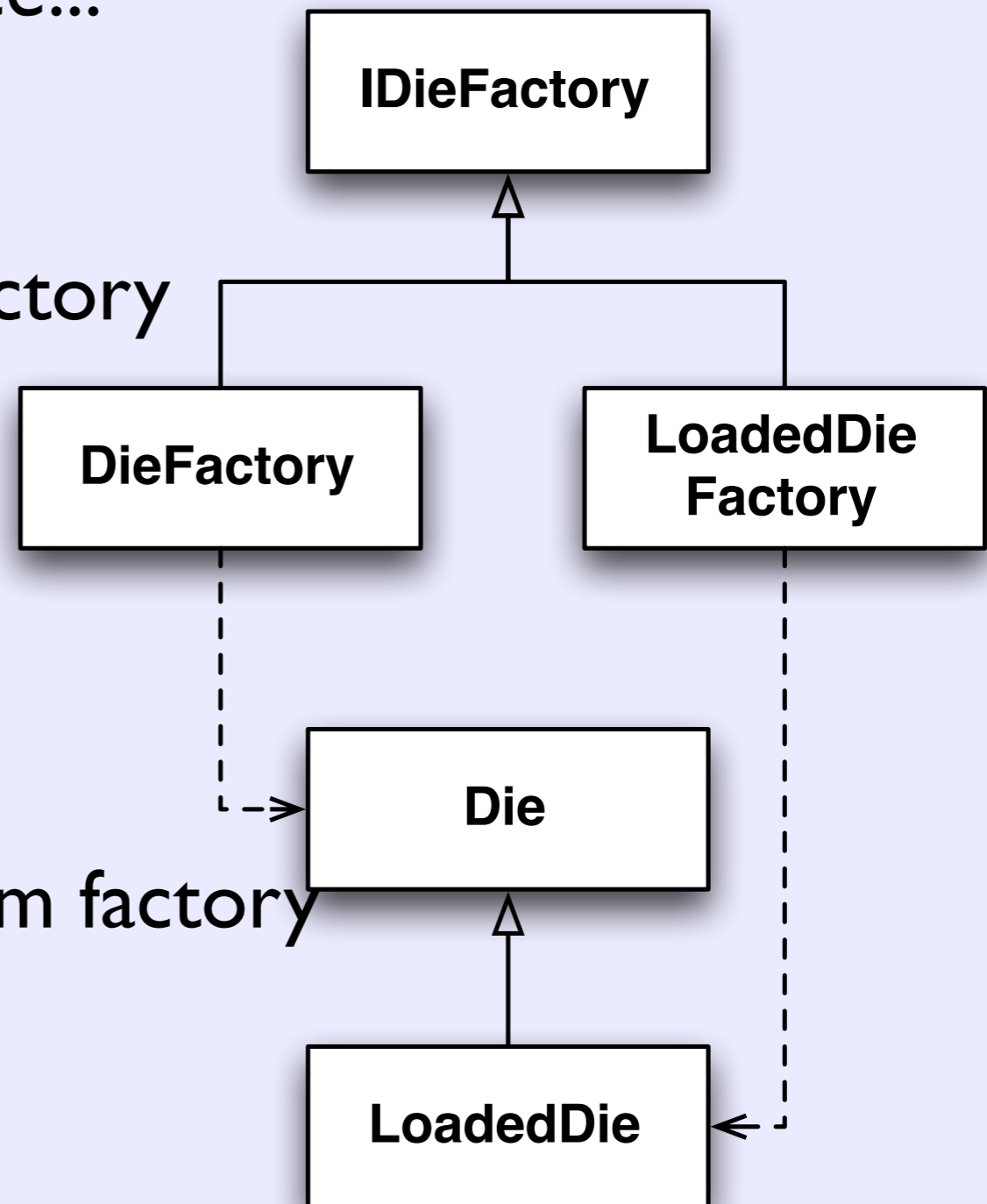
The 4-Contact Points
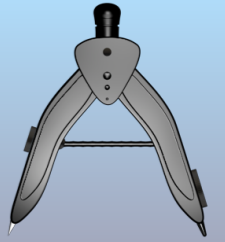
- There are 4 things you might be doing

    - Writing test code

    - Writing production code

    - Refactoring test code

    - Refactoring production code

- Do one of these at a time and finish before moving on

http://tinyurl.com/4-contact-points

New Test...

- We're not going to change everything at once...

```cpp
#include "LoadedDieFactory.h"
#include "Die.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(LoadedDieFactoryShould) {
};

TEST(LoadedDieFactoryShould, ReturnLoadedDie) {
    LoadedDieFactory factory(5);
    sp_Die d = factory.build();
    LONGS_EQUAL(5, d->faceValue());
}
```

LoadedDieFactory

- Notice the typedef? Seems it might be a good idea to make it its own beast.

- Otherwise, anything new here?

```cpp
#pragma once
#ifndef LOADEDDIEFACTORY_H_
#define LOADEDDIEFACTORY_H_

class Die;
#include <memory>
typedef std::shared_ptr<Die> sp_Die;

class LoadedDieFactory {
public:
  LoadedDieFactory(int value);
  virtual ~LoadedDieFactory();
  sp_Die build();

private:
  int faceValue;
};

#endif
```

LoadedDieFactory

- Only slightly tricky part is the loaded value to return...

```cpp
#include "LoadedDieFactory.h"
#include "LoadedDie.h"

LoadedDieFactory::LoadedDieFactory(int value)
   : faceValue(value) {
}


LoadedDieFactory::~LoadedDieFactory() {
}


sp_Die LoadedDieFactory::build() {
  return sp_Die(new LoadedDie(faceValue));
}
```

Build Test & Class

- Time to get this working, create

    - LoadedDieFactoryShould.cpp

    - LoadedDieFactory.h

    - LoadedDieFactory.cpp

Cup Needs Updating

- Now you need to be able to build a Cup differently...

```cpp
#include "Cup.h"
#include "LoadedDieFactory.h"

#include <CppUTest/TestHarness.h>

TEST_GROUP(CupShould) {
};

TEST(CupShould, BeConstructableWithSharedPointers) {
    LoadedDieFactory factory(3);
    Cup cup(factory.build(), factory.build());
    LONGS_EQUAL(6, cup.total());
}
```

Cup's new constructor

- Need a declaration

```
class Cup {
public:
   ...

   Cup(sp_Die d1, sp_Die d2);
```

- Then a definition

```
Cup::Cup(sp_Die d1, sp_Die d2) {
   dice.push_back(d1);
   dice.push_back(d2);
}
```

Get ye a new cup

- Create the CupShould.cpp

    - Why did you not already have one of those?

    - What about having it now?

- Update the Cup class, get your test passing

Instantiating DiceGame

- Now update one existing test and get it back to

```cpp
#include "LoadedDie.h"
#include "DiceGame.h"
#include "LoadedDieFactory.h"
#include <CppUTest/TestHarness.h>

TEST_GROUP(DiceGameShould) {};

TEST(DiceGameShould, DecreaseBalanceForLoss) {
  LoadedDieFactory factory(3);
  DiceGame game (factory);
  game.play();
  LONGS_EQUAL(-1, game.getBalance());
}
```

New DiceGame Construction

- The new method declared (old one still there for

```cpp
class LoadedDieFactory;

class DiceGame {
public:
  DiceGame(LoadedDieFactory &factory);
  DiceGame(Die *d1, Die *d2);
```

```cpp
#include "LoadedDieFactory.h"

DiceGame::DiceGame(LoadedDieFactory &factory):balance(0) {
  sp_Die d1 = factory.build();
  sp_Die d2 = factory.build();
  dice.reset(new Cup(d1, d2));
}
```

Make your updates

- Update this one test

    - Get everything to green

    - Update the next test: IncreateBalanceForWin

- Oops! There's one test we cannot change...

    - Review LeaveBalanceAloneForPush

    - How are we going to fix that?

One way... LoadedDieFactory

- Of course, start with a test:

```
TEST(LoadedDieFactoryShould, BeAbleToTakeTwoValues) {
  LoadedDieFactory factory(3, 4);
  sp_Die d1 = factory.build();
  sp_Die d2 = factory.build();
  LONGS_EQUAL(3, d1->faceValue());
  LONGS_EQUAL(4, d2->faceValue());
}
```

And an updated header

- Add an overloaded constructor

  - Be able to return up to 2 unique values

```cpp
class LoadedDieFactory {
public:
  LoadedDieFactory(int firstValue, int secondValue);
  LoadedDieFactory(int value);
  virtual ~LoadedDieFactory();
  sp_Die build();

private:
  int values[2];
  int lastIndex;
};
```

And the new definitions...

```cpp
LoadedDieFactory::LoadedDieFactory(int value) : lastIndex(-1) {
  values[0] = value;
  values[1] = value;
}

LoadedDieFactory::LoadedDieFactory(int firstValue, int secondValue)
  : lastIndex(-1) {
  values[0] = firstValue;
  values[1] = secondValue;
}

LoadedDieFactory::~LoadedDieFactory() {
}

sp_Die LoadedDieFactory::build() {
  lastIndex = (lastIndex + 1) % 2;
  return sp_Die(new LoadedDie(values[lastIndex]));
}
```

111

Update it

- Get the factory updated

  - Once you have the updated factory, fix the last test

- Now for cleanup

  - Remove the old constructor from DiceGame

  - In fact, remove all references to Die in the h & .cpp

  - Opinions?

Improved?

- Was this too much?

  - Should we have left it as is, taking two die objects?

  - Could we anticipate this and design accordingly?

- Oh, did you notice there's still more to do?

There's no DieFactory

- One issue with Test Doubles...

  - Can you build a real system?

- We cannot

  - There's no DieFactory

  - We have a few more moving parts to finish this effort

Extract an interface...

- Even though C++ does not have interfaces...

```
#pragma once
#ifndef IDIEFACTORY_H_
#define IDIEFACTORY_H_

class Die;
#include <memory>
typedef std::shared_ptr<Die> sp_Die;

class IDieFactory {
public:
  virtual ~IDieFactory() = 0;
  virtual sp_Die build() = 0;
};

#endif
```

One Method Implemented

- Even though destructor pure virtual...

```
#include "IDieFactory.h"

IDieFactory::~IDieFactory() {
}
```

- Makes sure whole hierarchy cleans up well

  - But subclasses not forced to implement

  - Best of both worlds

Then change LoadedDieFactory

- This is it...

```
#include "IDieFactory.h"

class LoadedDieFactory : public IDieFactory {
```

- If you forget public, it won't be substitutable

  - Common mistake

  - Default is private

Now DieFactory

- This test is a bit different

```cpp
#include "DieFactory.h"
#include "Die.h"
#include <typeinfo>
#include <CppUTest/TestHarness.h>

TEST_GROUP(DieFactoryShould) {
};

TEST(DieFactoryShould, ReturnOnlyDie) {
   DieFactory factory;
   sp_Die die = factory.build();

   CHECK(typeid(Die) == typeid(*die.get()));
}
```

118

DieFactory...

```cpp
#pragma once
#ifndef DIEFACTORY_H_
#define DIEFACTORY_H_

#include "IDieFactory.h"

class DieFactory
  : public IDieFactory {
public:
  DieFactory();
  virtual ~DieFactory();
  sp_Die build();
};

#endif
```

```cpp
#include "DieFactory.h"
#include "Die.h"

DieFactory::DieFactory() {
}

DieFactory::~DieFactory() {
}

sp_Die DieFactory::build() {
  return sp_Die(new Die);
}
```

DieFactory

- Time to create the test and files for DieFactory

  - What do you think of typeid?

  - Isn't this type-checking?

  - If you have some spare time, google

    - dynamic_cast

    - static_cast

    - const_cast

    - reinterpret_cast

Finally, a Smoke Test...

- Can the game be built correctly?

```cpp
#include "DiceGame.h"
#include "DieFactory.h"
#include <stdio.h>
#include <CppUTest/TestHarness.h>

TEST_GROUP(DiceGameSmokeTest) {
};

TEST(DiceGameSmokeTest, StandardUse) {
  DieFactory factory;
  DiceGame game(factory);

  for(int i = 0; i < 28219; ++i) game.play();

  char balance[32];
  snprintf(balance, 32, "Balance = %d", game.getBalance());
  UT_PRINT(balance);
}
```

- This isn't a unit test... Just what is it?

Wrap it up!

- Write this smoke test

  - It won't compile

  - So fix that

  - What's the output you get?

  - Where does this test belong?

  - What did it force you to do?

- If you do decided to add such a test to your unit test suite, remove the output.

Check List
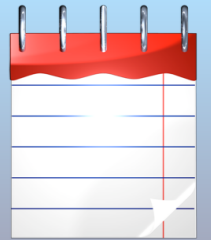
Abstract Factory	
Factory	
Interface Inheritance	
Order of creation	
Order of destruction	
Pure virtual dtor	

Final Recommendations

You've got a good start...

- You have used quite a bit of C++, there's much more

  - Books to add to your library:
    Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
    More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions
    Accelerated C++: Practical Programming by Example
    Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)
    Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library

- Practice, practice, practice

  - Consider working through Monopoly again

  - Code Katas: http://schuchert.wikispaces.com/Katas

- A final question:

Rpn Calculator

126

A Brief History of Rpn

- Created in 1920's by Jan Łukasiewicz

  - Doesn't require "look ahead", easier to implement

  - No operator precedence, execute as soon as encountered

  - AKA post-fix (normal is in-fix, functions are closer to pre-fix)

- HP 9100A, 1968

  - 3-level stack (X, Y, Z)

  - 16 storage registers

  - Several functions

  - 40 Pounds

  - Under USD $5,000

Project Description

- Develop a programmable Rpn calculator

    - Basic operations such as add, subtract, multiply, divide

    - Other functions such as sum, factorial, prime factors

    - Stack operations such as dup, drop, rotate up/down

    - Easy to add new operations

    - Create macros by combining existing operations

    - Macros execute as if they are built-in operators

    - Deviations

        - Allow larger that 4-entry stack

        - Use integer math for simplicity

What's Coming Up

- Analysis to Design

- Test Driven Development

- Overview

  - GRASP, SOLID, Code Smells, Legacy Refactoring, Test Doubles

  - The 4 Actions

Iteration Specifications

- Here are some examples to describe this iteration

Given	When	Then
30 4	+	34
30 4	-	26
4 6	*	24
8 2	/	4
5	!	120
1	-	-1
	+	0
	$%^unknown*&^	<error>

Example Use

- Given a new calculator

1. where the user first enters 30

2. and then enters 4

3. and finally selects add

4. then the result should be 34

0	T		0	T		0	T		0	T
0	Z		0	Z		0	Z		0	Z
0	Y		0	Y		30	Y		0	Y
0	X		30	X		4	X		34	X

1: enter 30 →   2: enter 4 →   3: add →

4: then

Subtraction

- What happens with subtraction and only 1 number?

  - Start with a new calculator

0	T
0	Z
0	Y
0	X

  - Enter a single number

0	T
0	Z
0	Y
1	X

  - Subtract is defined as Y - X, which is 0 - 1 in this case:

0	T
0	Z
0	Y
-1	X

Consider Multiple Consumers

- Discuss

```
+---------+   +---------+   +---------+   +---------+
|   GUI   |   |   TUI   |   | Rest API|   |  Tests  |
+---------+   +---------+   +---------+   +---------+
```

```
        +-------------------+
        |   RpnCalculator   |
        +-------------------+
```
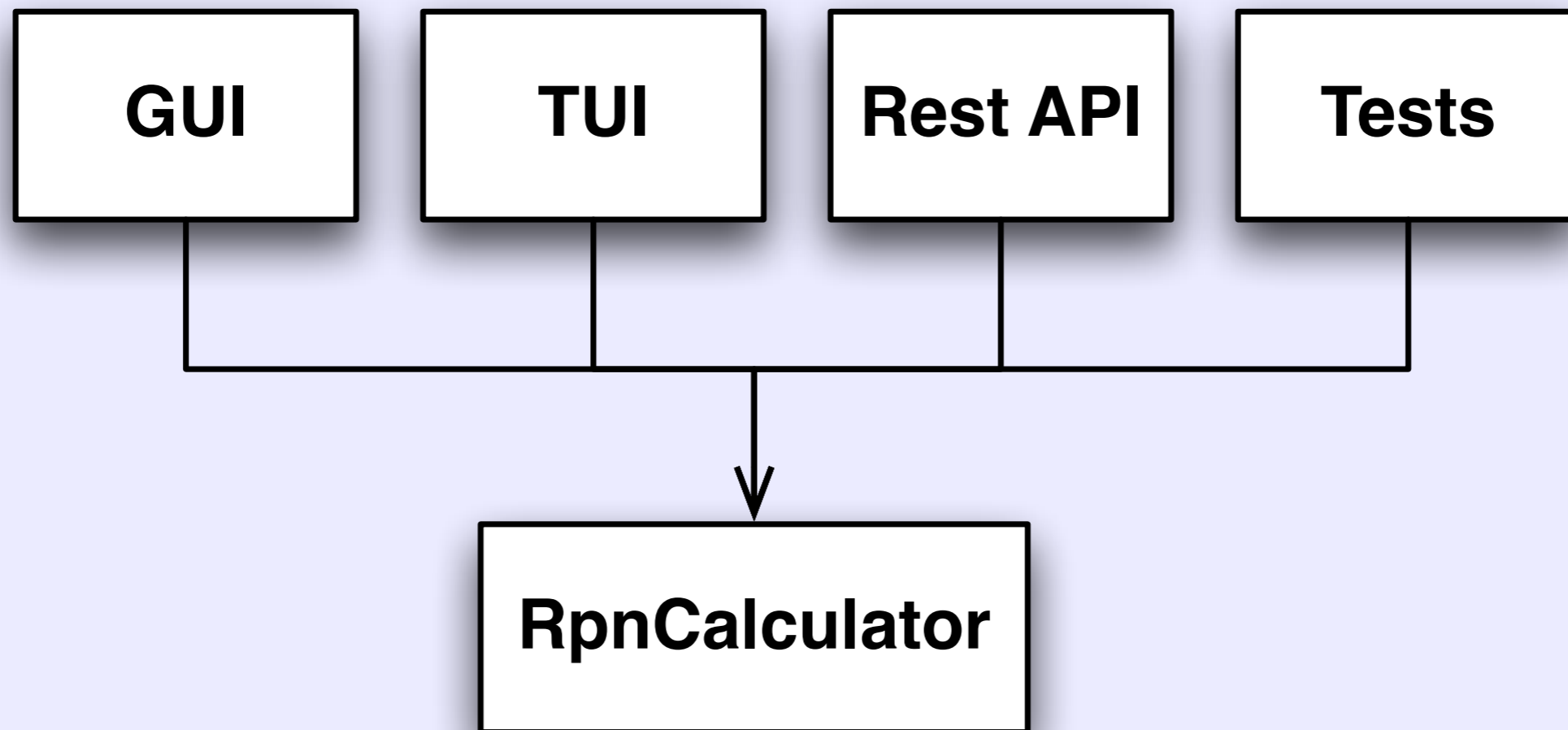
Sequence Diagrams

- Develop Sequence Diagrams for
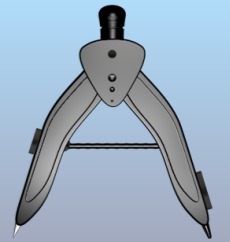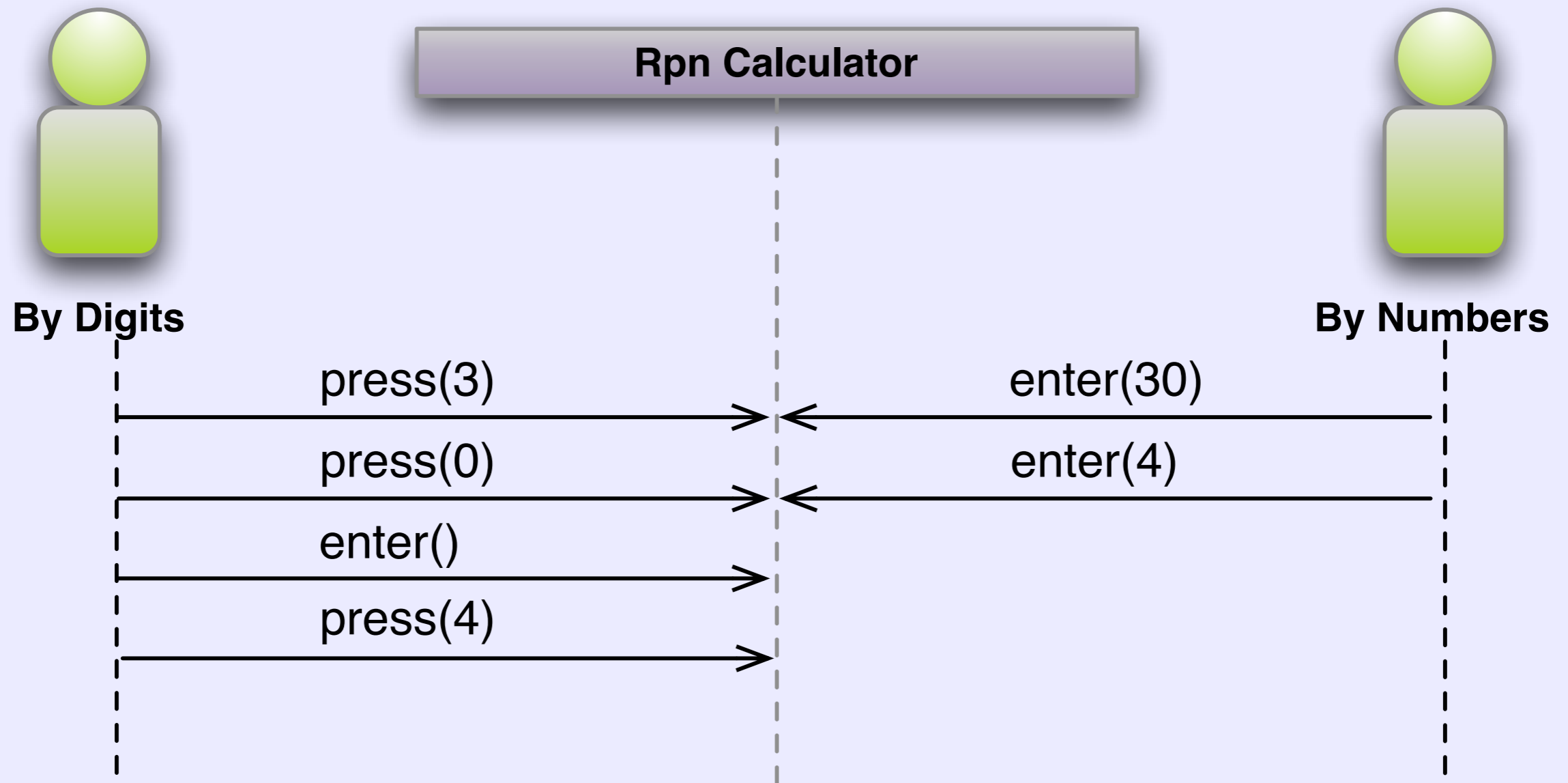
  - Add

  - Factorial

- Two Levels of Detail (1st system, then object)

  - System level (actor to system)

  - Object level (detailed interaction)

- As you do ...

  - Consider several kinds of clients: web, gui, text, test

  - Hint, stacks are in the domain (see any HP calculator manual)

  - Yes, that much detail

134

Key Design Decision

- By digits or by numbers?

By Digits

Rpn Calculator

By Numbers

press(3)   enter(30)

press(0)   enter(4)

enter()

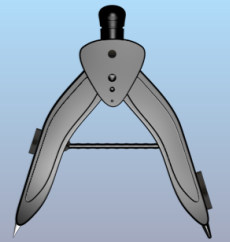press(4)

- By numbers - to "enter key" or not?

Key Design Decision

- Method per operator or parameterized?

Rpn Calculator

By Method   **By Parameter**

add()	perform("+")
subtract()	perform("-")
multiply()	perform("*")
divide()	perform("/")
factorial()	perform("!")
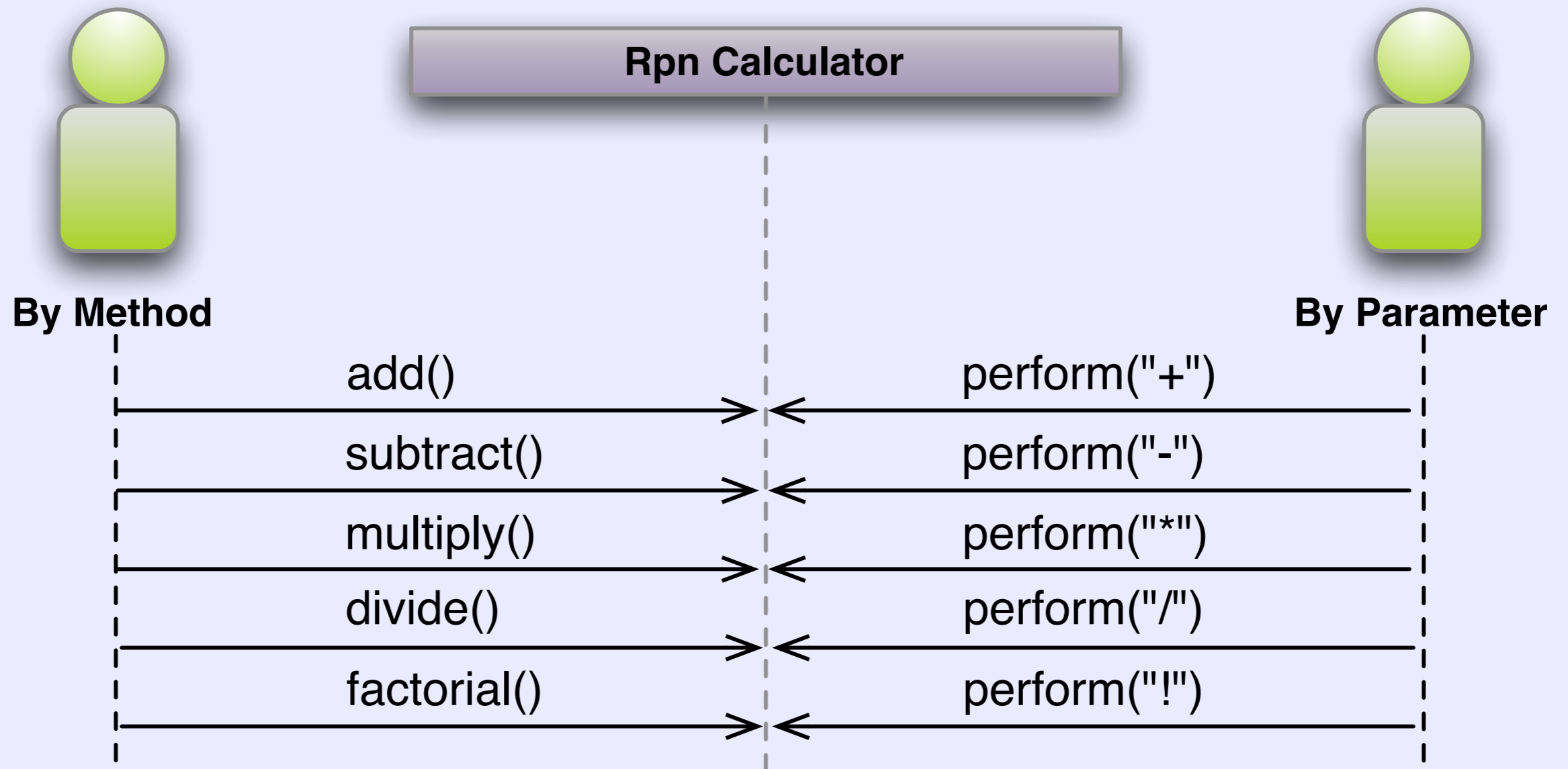
- What would the Open/Closed Principle Suggest?

One Possible Approach

- This will get the job done

  - it's more that Extreme TDD would suggest

	Rpn Calculator	s: Stack	Add

enter(30)

push(30)

enter(4)

push(4)

perform("+")

findOperatorNamed("+")

execute(s)

rhs := pop()

lhs := pop()

push(lhs + rhs)
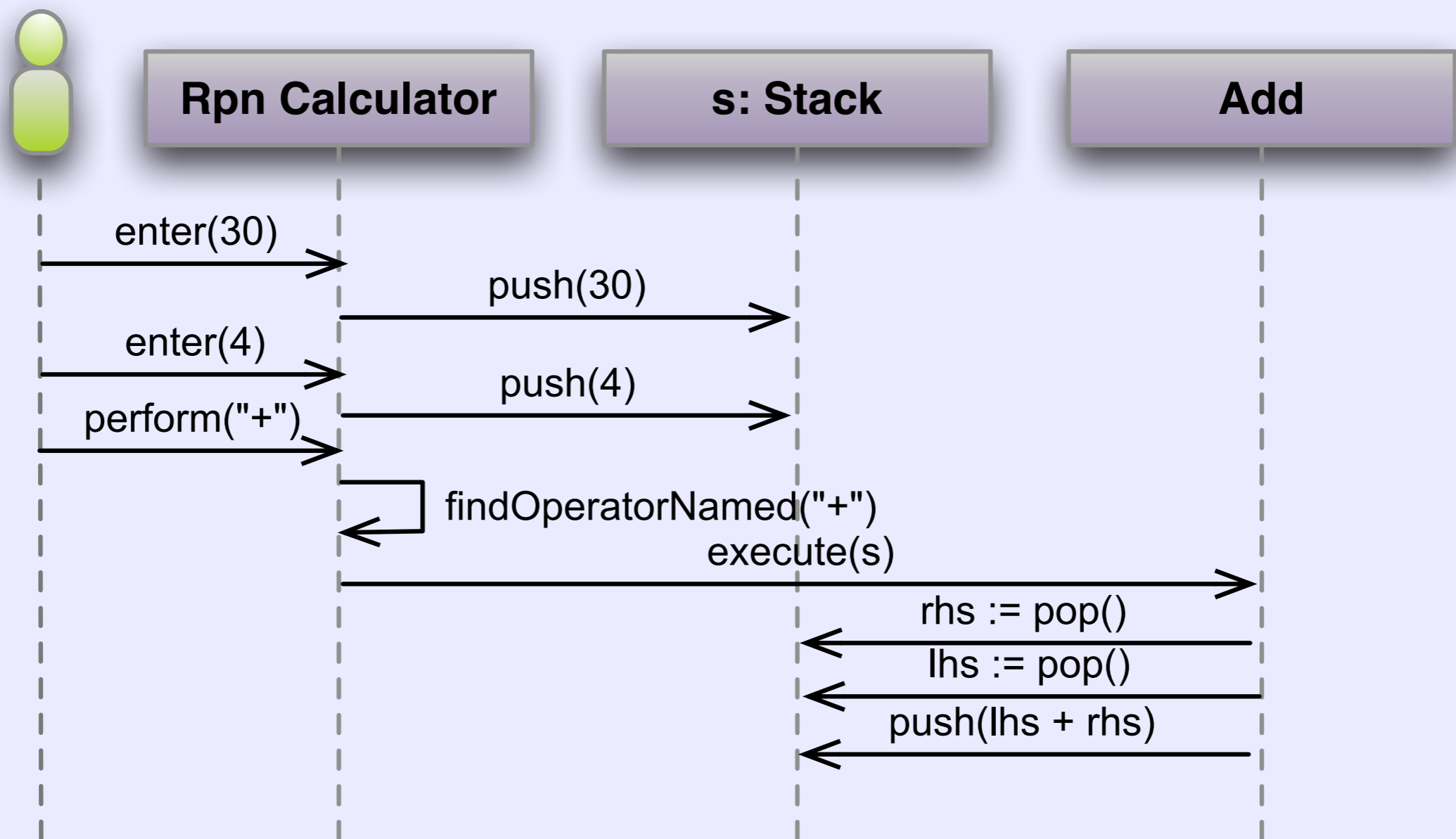
Test Driven Development
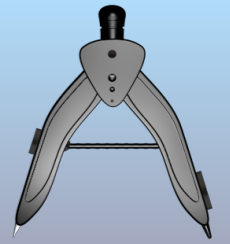
- We'll be applying (modified) laws of TDD

    - Write no production code without a failing test

    - Write a failing test

    - Get the test to pass

- Another way to think of it:

    - Red — Write Failing Test

    - Green — Get Failing Test to Pass

    - Blue — Refactor

(Red) Writing a Failing Test
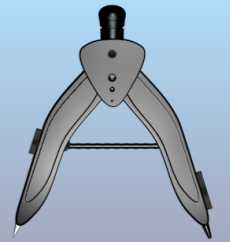
- For now, start with your SSD'd

  - Create tests for each SSD, one at a time

  - Use common setup to guide creating fixtures

  - Use API calls on a controller object

  - Get the code to compile with the test failing

  - Test "too big"? Break it into parts and work on that part.

    - If so, do you think a dynamic diagram might help the design?

Getting to (Green)

- Given a failing test

  - Write just enough code to get it to pass

    - Might be hard-coded return

    - Might be more general

- Write the simplest thing that could possibly work

- Observations

  - Code should be more general as tests added.

  - Simple is not the same as stupid!

  - Do not make other tests fail along the way

(Blue)Refactor
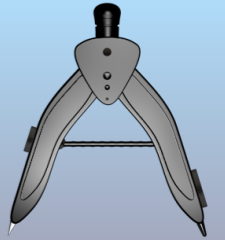
- Review production code and tests

  - Look for code smells

  - Consider removing them

  - Consider generalizing based on what's coming up

  - All generalizations add cost, make sure the'll payoff

Time to Code

- Create a new project

  - Set up: include & library directory / add library

  - Make sure to set: refresh automatically, save before build

- Test your way into

  - Getting add to work

  - Getting subtract to work

  - Getting factorial to work

Well... Hum
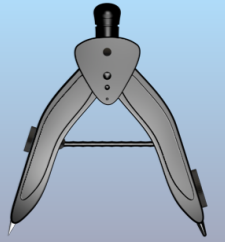
- There are problems with the code

    - Duplication between Add and Subtract

    - Selecting the operator's not going to grow well

- Other things you notice?

143

Pay down the debt
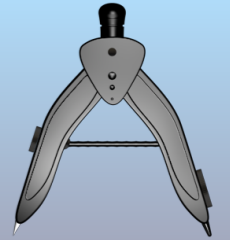
- Time to pay down our design debt

  - Design & implement duplication removal

  - Improve operator selection

  - Anything else we've identified.

After all of that refactoring

RpnStack

1

RpnCalculator

<<factory>>
MathOperator Factory

1

<<interface>>
<<strategy>>
IMathOperator

execute(:RpnStack):void

*

<<template method>>
Binary Math Operator

execute(:RpnStack):void
doExecute(:int, :int):int

Factorial

Add

Subtract

Multiply

Divide
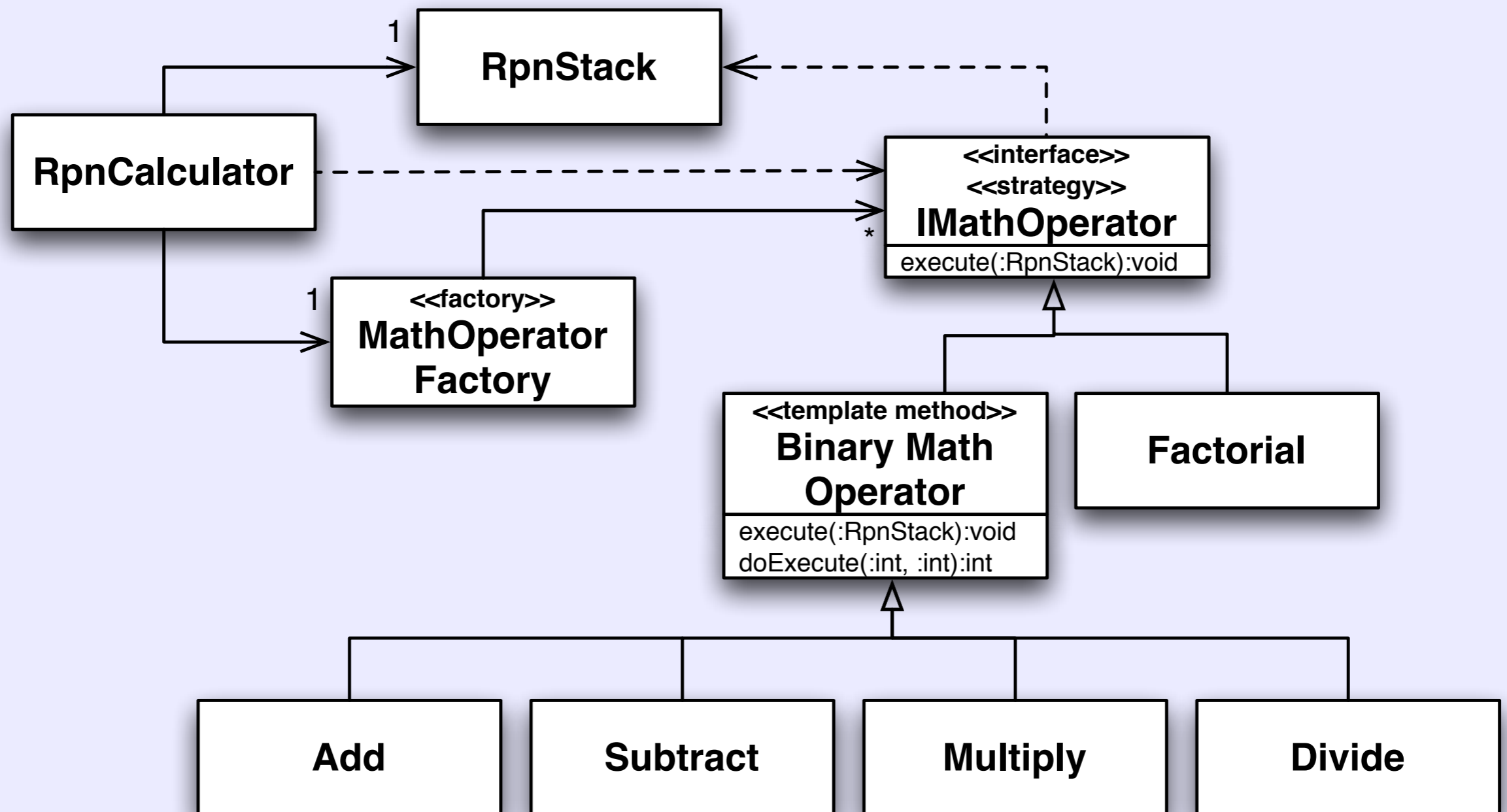
Strategy

- Replace interchangeable algorithms with hierarchy
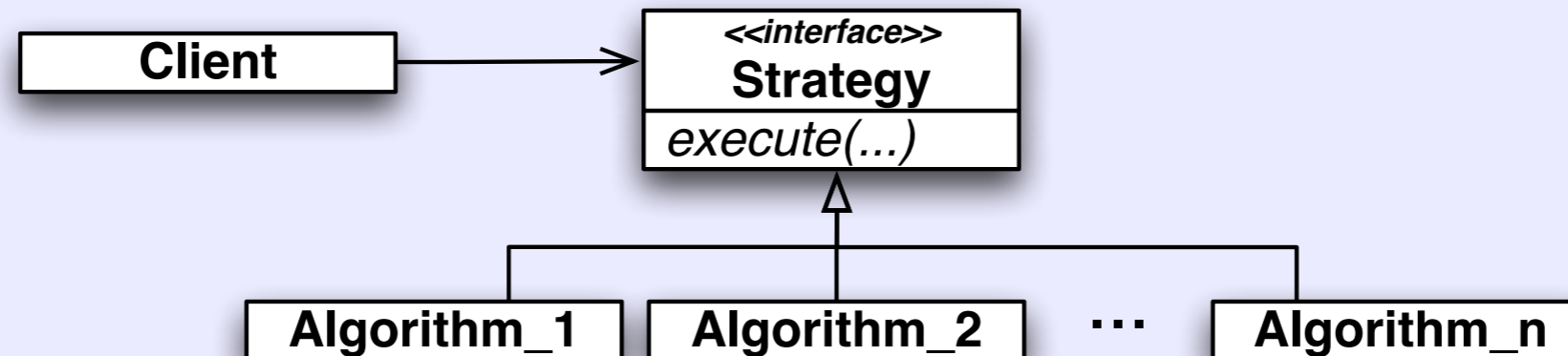
- Keys:

  - Base interface

  - Single method - client does not select what to call

- Examples

  - Billing Strategy: retail, business, industrial

  - Monopoly: Reading card from community chest or chance

Template Method
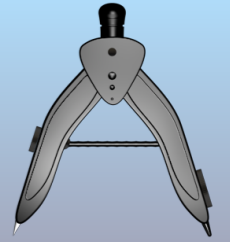
- Base algorithm w/extension points in child classes

```
                                        <<abstract>>
   Client      ────────────▷       TemplateMethod
                                   baseAlgorithm(...)
                                   extensionPoint(...)
                                           △
          ┌────────────────────────────────┼────────────────────────────┐
    Version_1               Version_2               ...        Version_n
   extensionPoint(...)     extensionPoint(...)                extensionPoint(...)
```
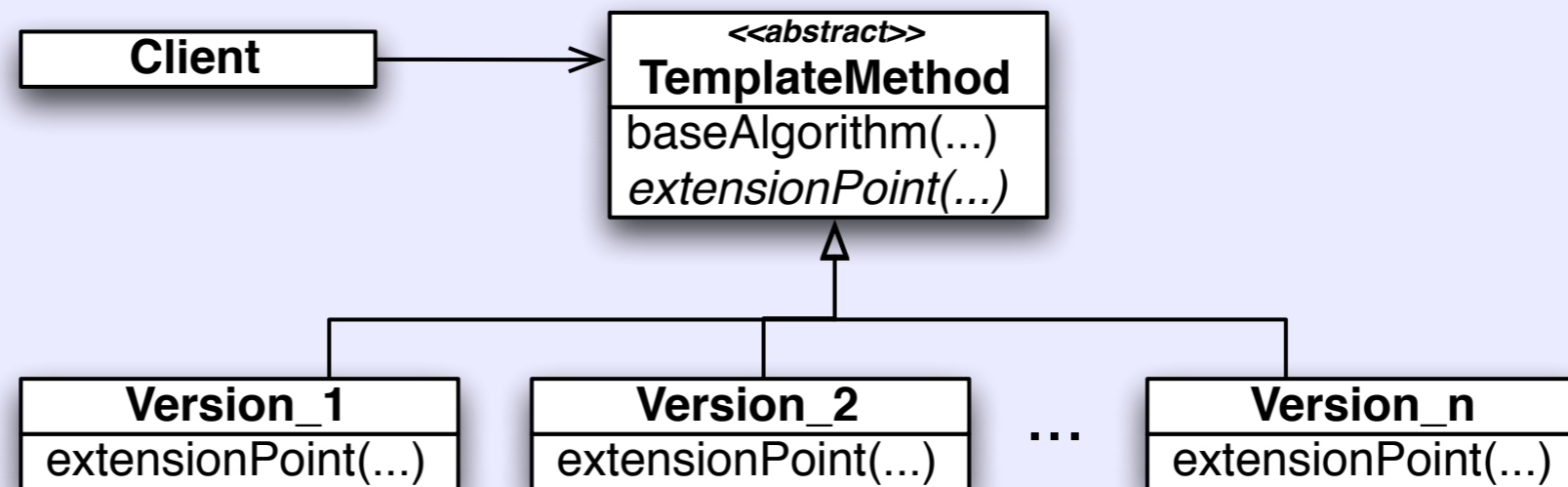
- Keys

  - Base method calls abstract methods

  - Abstract methods implemented in derived class

- Examples

  - Protocol Enforcement

  - Monopoly: Rent calculation

Abstract Factory
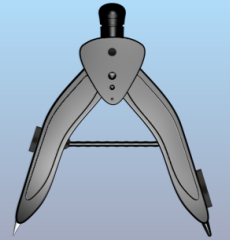
- Describe an interface for building objects

```
  ┌──────────┐         ┌─────────────────┐
  │  Client  │────────▷│  <<interface>>  │
  └──────────┘         │     Factory     │
                       ├─────────────────┤
                       │    build(...)   │
                       └─────────────────┘
                                △
              ┌─────────────────┼─────────────────┐
      ┌──────────────┐  ┌──────────────┐    ┌──────────────┐
      │  Factory_1   │  │  Factory_2   │ ... │  Factory_n   │
      └──────────────┘  └──────────────┘    └──────────────┘
```
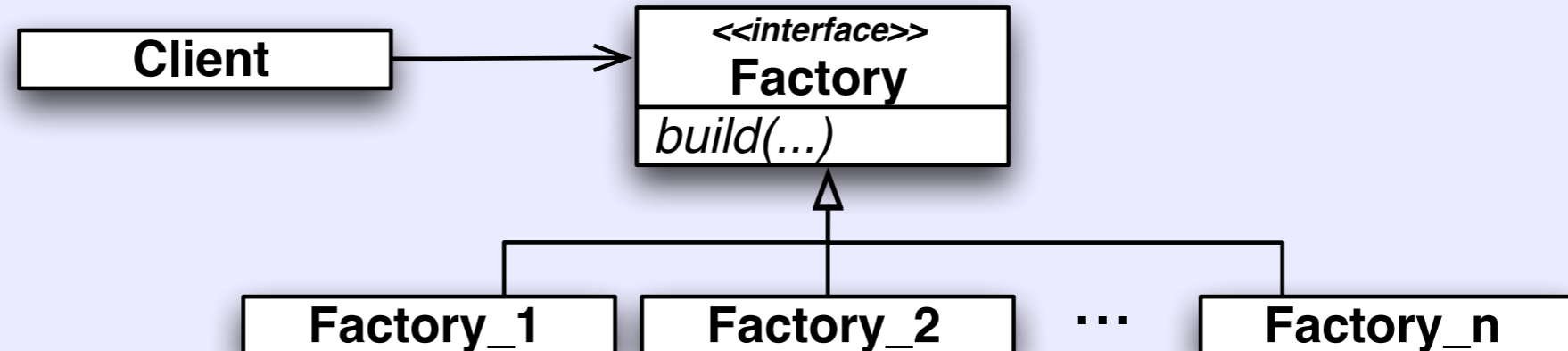
- Key

  - Maps some identifier to an object

  - May create new objects, copy existing ...

Iteration Specs

- Here are some examples to describe this iteration

```
3<enter>
5<enter>
2
sum
10
```

Given	When	Then
1	primeFactors	
2	primeFactors	2
3	primeFactors	3
4	primeFactors	2 2
5	primeFactors	5
6	primeFactors	2 3
7	primeFactors	7
8	primeFactors	2 2 2
9	primeFactors	3 3
10	primeFactors	2 5
11	primeFactors	11
12	primeFactors	2 2 3
64	primeFactors	2 2 2 2 2 2
75 8	primeFactors	75 2 2 2

Extra Credit

Given	When	Then
1 2	swap	2 1
15 4	drop	15
42	dup	42 42
3 4 2	ndup	3 4 3 4

You're on your own

- You have some time

  - Use TDD to develop Sum

  - Use TDD to develop Prime Factors

  - Oh, how did you test that they are available to the calculator?

- You have an Operator Factory...

  - What happens if you accidentally replace existing operator?

Final Iteration

- Here are some examples to describe this iteration

```
start
+
*
-

save ams
3<enter>
5<enter>
2<enter>
13
ams
-72
```

```
start
primeFactors
sum
save sumOfPrimeFactors
12
sumOfPrimeFactors
7
```

Extra Credit

```
start
2
*
save times2
6<enter>
times2
12
```

```
start
2
ndup
<
if
drop
else
swap
drop
then
save min
4
6
min
4
-1
min
-1
```

Programmable

- Google the composite pattern
  http://schuchert.wikispaces.com/
  cpptraining#LinksToDesignPatterns

  - Use it as part of your design for programmability

  - Design an appropriate solution

  - End to end

  - Does this functionality belong in the calculator?

Composite Object

- A composite behaves as something else...

```
                              <<interface>>        *
  Client  ────────────────→     Strategy  ←──────────┐
                               execute(...)          │
                                    △                │
                        ┌───────────┼───────────┐    │
                   Algorithm_1  Algorithm_2  ...  Composite
```
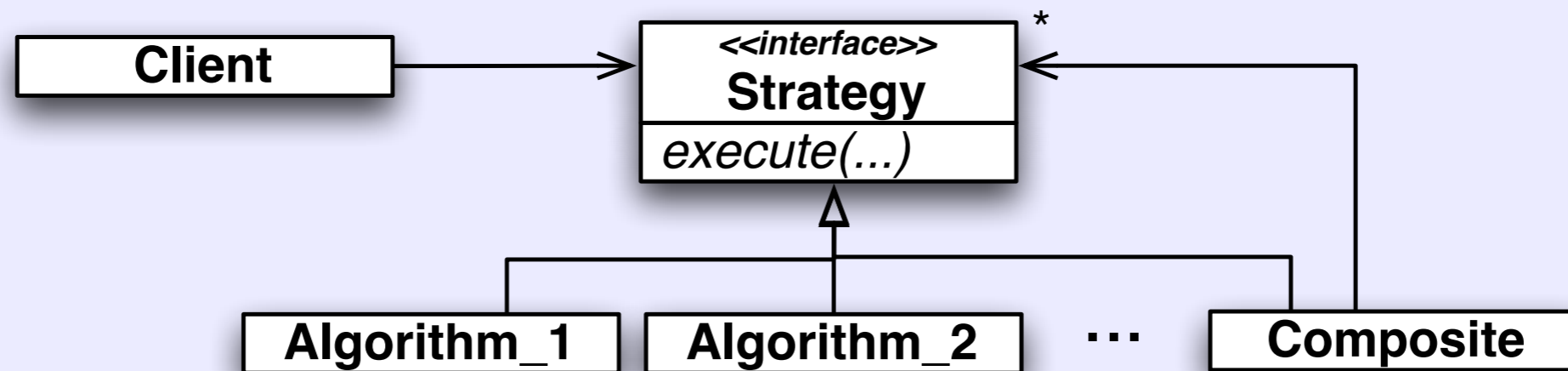
- And holds one to zero or more of those

```
  :Composite ──→ :Collection ──→ :Add
                            ──→ :Multiply
                            ──→ :Subtract
```
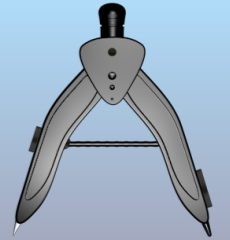
Composite

- Sending message to Composite...

Composite	Add	Multiply	Subtract

execute(s: Stack)

execute(s: Stack)

execute(s: Stack)

execute(s: Stack)
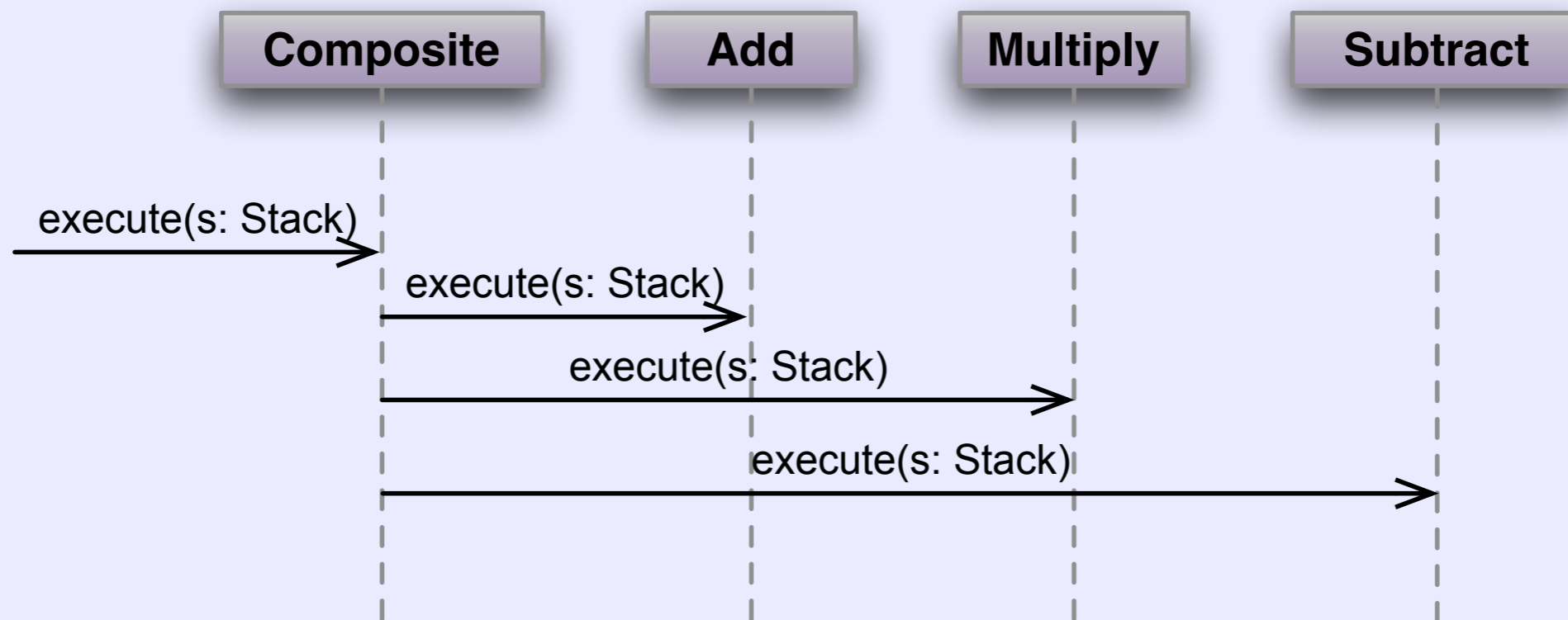
- Sends a message to each of its contained objects

Project Recap

- Started with a small solution

  - Now it is programmable

  - Looking back, did it seem possible? Rushed?

- How do you do this going forward?

- What's next to learn?

Design, Design, Design

- Here's a staring list to help with OOD

GRASP	Craig Larman
SOLID	Robert Martin
Code Smells	Martin Fowler
WELC	Michael Feathers
Test Doubles	Several
Coding Katas	Several
Design Patterns	Gang of 4

- http://schuchert.wikispaces.com/TddIsNotEnough

GRASP

INFORMATION EXPERT	Assign responsibility to the thing that has the information.
CONTROLLER	Assign system operations (events) to a non-UI class. May be system-wide, use case driven or for a layer.
LOW COUPLING	Try to keep the number of connections small. Prefer coupling to stable abstractions.
HIGH COHESION	Keep focus. The behaviors of a thing should be related. Alternatively, clients should use all or most parts of an API.
POLYMORPHISM	Where there are variations in type, assign responsibility to the types (hierarchy) rather than determine behavior externally,
PURE FABRICATION	Create a class that does not come from the domain to assist in maintaining high cohesion and low coupling.
PROTECTED VARIATIONS	Protect things by finding the change points and wrapping them behind an interface. Use polymorphism to introduce variance.

SOLID Principles

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

S	SINGLE RESPONSIBILITY	Single Reason to Change
O	OPEN/CLOSED	Open for extension closed to change
L	LISKOV SUBSTITUTION	Derived types substitutable for base types
I	INTERFACE SEGREGATION	Interfaces should be focused (small) & client specific
D	DEPENDENCY INVERSION	Dependencies should go from concrete to abstract

Package Cohesion/Coupling

- Guidelines for package cohesion

REP	Release/Reuse Equivalency	What you release is what you reuse.
CCP	Common Closure	Classes that change together should be packaged together
CRP	Common Reuse	Classes that are used together should be packaged together

- Guidelines for package coupling

ADP	Acyclic Dependencies	No cycles in your dependencies
SDP	Stable Dependencies	Dependencies should go from less to more stable. Depend on stable things
SAP	Stable Abstractions	Abstraction increase with stability

A Few Code Smells

- A few of Martin's code smells:

Poor Names	Name suggests wrong intent
Long methods	More than 1 thing/multiple levels of abstraction
Large classes	More than one concept/multiple levels of abstraction
Long parameter list	Too many arguments to keep straight (> 3)
Duplicated code	Same or similar code appears in more than one place
Divergent change	The class/method changes for dissimilar reasons
Shotgun Surgery	Single change affects multiple classes/methods
Feature Envy	One class uses another class' members
Switch Statements	Duplicated switches/if-else's over same criterion

- http://c2.com/cgi/wiki?CodeSmell

Some Legacy Refactorings

- From Working Effectively with Legacy Code

ADAPT PARAMETER	326	Change parameter to an adapter when you cannot use extract interface
BREAK OUT METHOD OBJECT	330	Convert method using instance data into a class with a ctor and single method
ENCAPSULATE GLOBAL REFERENCES	339	Move access to global data into access via a class to allow for variations during test
EXTRACT AND OVERRIDE CALL	348	Turn chunk of code into overridable method and then subclass in test
EXTRACT AND OVERRIDE GETTER	352	Turn references into hard-coded object into call to getter and then subclass
EXTRACT INTERFACE	362	Extract interface for concrete class, then use interface. Override in test.
INTRODUCE INSTANCE DELEGATOR	317	Add instance methods calling static methods. Call through instance, which test subclasses.
PARAMETERIZE CONSTRUCTOR PARAMETERIZE METHOD	379 383	Examples of Inversion of Control (IoC)
SUBCLASS AND OVERRIDE METHOD	401	Test creates subclass & passes it in/requires some IoC
SPROUT METHOD SPROUT CLASS	59 63	Create a method or class out of existing code.

Test Doubles

- Gerard Meszaros
  http://xunitpatterns.com/Test%20Double%20Patterns.html

DUMMY	Empty implementation. Not called or don't care if it is
STUB	Canned replies – "snapshot in time"
SPY	Watches and Records
FAKE	Partial Simulator
MOCK	Has & Validates expectations
SABOTEUR	Designed to always fail, e.g., always throws an exception.

Design Patterns

- **From:** Design Patterns: Elements of Reusable Object-Oriented Software

STRATEGY	Define a function or algorithm as a class. Form a wide but shallow hierarchy of different algorithms.
TEMPLATE METHOD	Write an algorithm in a base class with extension points represented as abstract methods. Subclass and override.
ABSTRACT FACTORY	A base interface for creating one or a family of objects through a standard API. Create implementations for each family of objects that need to be created.
COMPOSITE	A class that implements some other interface and also holds onto zero or more instances of that same interface.
STATE	Similar to strategy, though the states are interdependent. States can cause a so-called context to change from one state to another during its lifetime.

Additional Resources

- ## Video Series

C++	Dice Game	http://vimeo.com/album/254486
C#	Shunting Yard	http://vimeo.com/album/210446
JAVA	Rpn Calculator	http://vimeo.com/album/205252
IPHONE	iPhone & TDD	http://vimeo.com/album/1472322

- ## Mocking

JAVA	Mockito	http://schuchert.wikispaces.com/Mockito.LoginServiceExample
C#	Moq	http://schuchert.wikispaces.com/Moq.Logging+In+Example+Implemented

- ## Other

JAVA	FitNesse	http://schuchert.wikispaces.com/FitNesse.Tutorials
RUBY	Several	http://schuchert.wikispaces.com/ruby.Tutorials
JAVA	UI	http://schuchert.wikispaces.com/tdd.Refactoring.UiExample

Thank

Fin.

You!