BACK TO THE WELL

Principles of Software Design

Chuck Allison

Better Software Conference, November 2011

(slides available at freshsources.com/bs)

BANGKOK (Reuters – May 12, 2003) – Security guards smashed their way into an official limousine with sledgehammers on Monday to rescue Thailand's finance minister after his car's computer failed… All doors and windows had locked automatically when the computer crashed, and the air-conditioning stopped, officials said. "We could hardly breathe for over 10 minutes… It took my guard a long time to realize that we really wanted the window smashed so that we could crawl out. It was a harrowing experience."

Internal Software Quality

- “The General Principle of Software Quality is that improving quality reduces development costs”
  — McConnell, *Code Complete 2nd Ed.*

- “The purpose of design is to create a clean and relatively simple internal structure”
  — Stroustrup, *C++ Prog. Lang., 2nd Ed*.

Attributes of Quality Software

- Usable
- Reliable
- Adaptable
- Cost effective
- ***How are these attributes attained?***

Quality is No Accident

- It is *designed* into software
- It is *maintained* by refactoring and other agile practices

- But it all starts with ***Design***
- “Reliability cannot be retrofit” – *P. J. Plauger*

This Morning's Agenda

- What is Design?
- Fundamental Principles of Software Design
- The SOLID Principles of OO Design
- Applying Good Design Principles

What Is Design?

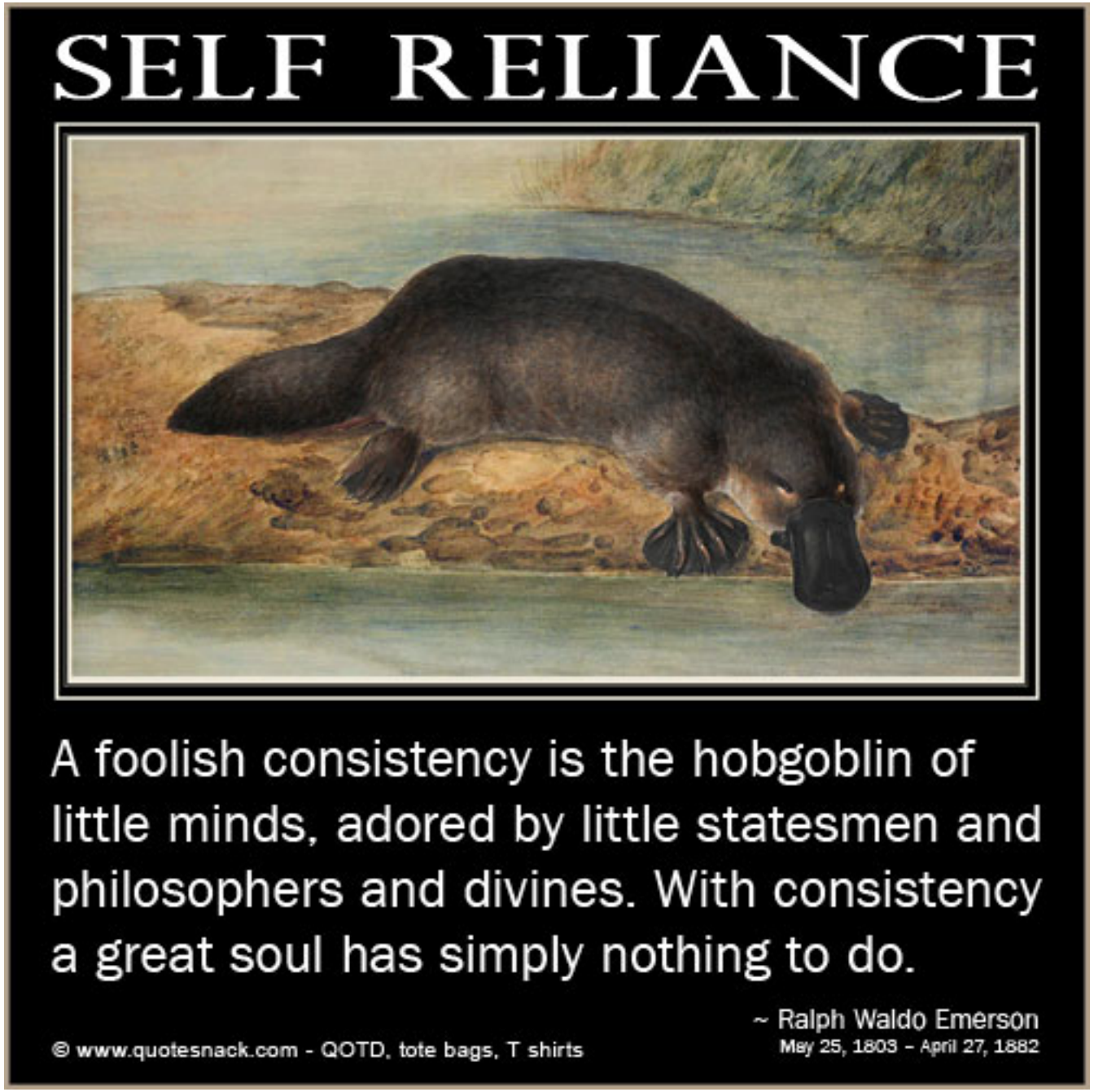
- 1. to prepare the preliminary sketch or the plans for a work to be executed, especially to plan the form and **structure** of: *to design a new bridge*.

- **2**. to plan and fashion **artistically** or **skillfully**.

- 3. to intend for a definite purpose: *a scholarship designed for foreign students*.

About Software Design

- Based on *timeless design principles*
- Methodologies come and go
  - "Methodologies are increasingly seen as helpful guides to capable people who could do most of it without them, rather than as the step-by-step cookbook that *removes responsibility* for understanding what you are doing and why." – *CSC Report*
- The ultimate expression of design is found in *code*


SELF RELIANCE
A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do.
~ Ralph Waldo Emerson
May 25, 1803 – April 27, 1882
© www.quotesnack.com - QOTD, tote bags, T shirts

Disclaimer

- Design is a Creative, Imprecise, Incremental Art
  - “Discovery of the meaningful abstractions in a given domain is an evolutionary process” (— Booch)
  - Has both *tactical* (*physical*) and *strategic* (*architectural*) components
- Design Principles are sometimes *at odds* with each other
  - There is no One Right Answer
  - This is Good Thing
- You, the Designer, *resolve* the conflict
  - by *balancing the competing forces* at play according to timely, local and global needs

Cogent Quote

"I am quite convinced that in fact computing will become a very important science. But at the moment we are in a very primitive state of development; we don't know the basic *principles* yet and we must learn them first. If universities spend their time teaching the state of the art, they will not discover these principles and that, surely, is what academics should be doing."

– *Christopher Strachey*, 1969

DESIGN FUNDAMENTALS

“Fundamentals Never Go Out Of Style”

– *Grady Booch* (CIO.com 2008)

- Create Crisp and Resilient *Abstractions*
- Maintain a Good *Separation of Concerns*
- Focus on *Simplicity*

"The acts of the mind, wherein it exerts power over simple ideas, are chiefly *these three*:

1) Combining several simple ideas into on *compound* one, and thus all complex ideas are made.

2) The second is bring two ideas, whether simple or complex, together, and setting them, by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of *relations.*

3) The third is *separating* them from all other ideas that accompany them in their real existence: this is called *abstraction*, and thus all *general ideas* are made."

— John Locke (1690)

“The acts of the mind, wherein it exerts power over simple ideas, are chiefly these three:”

1) Combining existing parts into a new “whole” (composition)

2) Relationships (coupling)

3) Naming and encapsulating abstractions

Abstraction: The Currency of Software

- Dictionary Definition of Abstraction:
  - 1. A *general quality* or characteristic, apart from concrete realities, specific objects, or actual instances
  - 2. The process of formulating generalized ideas or concepts by *extracting common qualities* from specific examples
- "Abstraction is the essence of simple and effective software design" – *Tony Hoare*

Abstraction: Key To Design (and Business) Success

- "The art of abstract thinking lies in choosing qualities that are not only appropriate for current examples, but that will continue to apply in [the] future."

- "In business terms, it is to find a *metaphor or description* for the business that will endure for longer than the particular way the business operates today."

    - *-- CSC Report, "Implementing Business Objects"*

Selective Ignorance

- “It does not mean [totally] ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant.” – *Dijkstra*, 1974

- The *details* of making a concept concrete and usable are *hidden* from users
  - implementation is *encapsulated*

- User are left with the *essence* of the concept
  - the *interface*

- Hence, abstractions allow us to work at a *higher level*

Abstractions Determine Quality

– Daniel Jackson, MIT

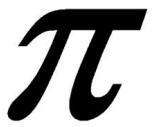
	Abstractions	For User	For Developer
good	robust, flexible	clear model	clean interfaces
typical	weak, broken	complex model	messy interfaces
bad	non-existent	no model	coupling hell

Where Abstractions Live

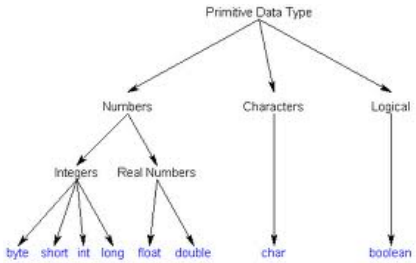
- Everywhere!
- In the user/problem space
  - e.g., a word, paragraph, vehicle, customer, itinerary, train, loan
- In the solution space
  - e.g., a thread, a hash table, a drawing context
- In between
  - e.g., a spreadsheet

Common Software Abstractions

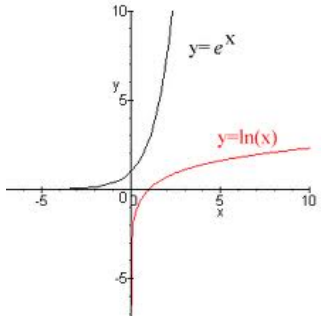
- Symbolic Constants

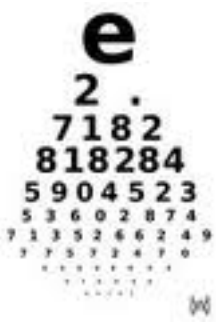


- Types

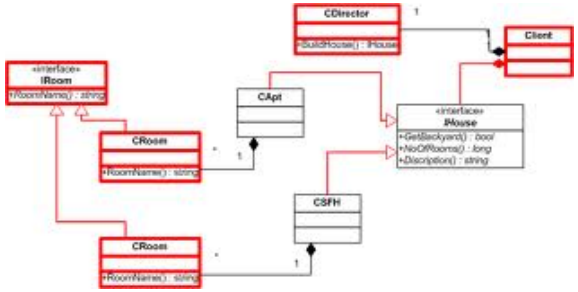


- Functions





- Classes



- Modules

- Patterns …

"An abstraction denotes the *essential characteristics* of an object that distinguish it from all other kinds of objects and this provides crisply defined *conceptual boundaries*, relative to the perspective of the viewer."

"An abstraction focuses on the *outside view* of an object, and so serves to *separate* an object's essential behavior from its implementation."

— Booch

Abstraction Boundaries

- All abstractions *encapsulate* something
- The boundary is the *interface*
- Clients (should) *program to the interface*
- *Where* the boundary is drawn, and what the *interface* is determines the *quality* of the abstraction

Principle of Least Astonishment

- “An abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction.”

Principle: Program to an Interface

- *Not* an Implementation
  - Interfaces should *expose no internals*
- "No part of a complex system should depend on the details of any other part." (Ingalls, POPL 5)

Principle: Program to an Interface

- Dependencies on implementation leads to “Coupling Hell”
  - users should be shielded from implementation detail
  - they are *partly responsible* to remain so
  - i.e., follow the *Principle of Least Knowledge*
- “What you do in the bathroom is no secret, but it *is* private.” – *P.J. Plauger*

Exposed Implementation

- Getters
  - internal types may change

Exposed Implementation

- Setters
  - you have lost control of the abstraction

Measuring the Quality of Abstractions

- How do we know what belongs “inside” and what belongs “outside”

- “Intracomponent linkages are generally stronger than intercomponent linkages. This fact has the effect of *separating* the high-frequency of the components – involving the *internal structure* of the components – from the low-frequency dynamics – involving *interaction among components*.” (Simon, H., *The Sciences of the Artificial*, Cambridge, 1982)

Overall Guiding Principle of Design

CREATE COHESIVE ABSTRACTIONS

Cohesion

- *cohere:* to be naturally or logically connected
- “Conceptual Clustering”
- *Principle: Maximize Cohesion*
  - *Self-containment*: Keep together things that *belong* together
  - *Separate* everything else
- Most sound design practices are variations of or follow from this *guiding principle*

Aiming for Cohesion

– P. J. Plauger

- "When you can describe what a module does in a *simple, active sentence*, then you probably have a highly cohesive module that will stay around."

- "Descriptions such as 'clear update record', or 'compute alternative minimum tax', indicate functional cohesiveness."

Principle: Minimize Coupling

- Coupling ⇔ Cohesion == *Yin* ⇔ *Yang*

Principle: Minimize Coupling

- When things that belong together are not together, then there *is needless communication across abstraction boundaries*

Principle: Minimize Coupling

- High coupling is a sign of poor cohesion
  - *think cohesion*; low coupling will naturally follow
  - *think objects*; data and methods follow from *responsibilities*
  - *think high level*; details will follow

Fundamentals of Function Design

- A function should be as *self-contained* as possible
  - it does its *one job* but no more; no extraneous code
- Black-box Model of a Function:

- *Dependencies* on external objects should be *minimal*
  - Preferably only through *parameters* and *return values*

Functions vs. Procedures

- Side Effects!
  - changes other than in return values
- Procedures change *non-local* state
  - behind the caller's back
- We routinely do this with objects and methods
  - Complex objects are hard to debug
  - Getters and Setters are often Problematic

Suspicious Types of Coupling

- Access to *variables in other scopes*
  - *global* variables
  - other *non-local* variables, e.g. via *reference parameters*
- Access to *functions* inside other scopes
- Access to *types* inside other scopes
- All these dependencies can *complicate* software

Case Study: String Tokenizing

- Consider C's **strtok** function

- `char* strtok(char* search, const char* break);`

- The string, **search**, is traversed and *modified*
  - characters in **break** are skipped
  - a '\0' (NUL) replaces the first break character after the token
  - it *remembers* where it left off for subsequent calls
  - the position of the first non-break character is returned

- Note: **strtok** must be called in *2 different ways*

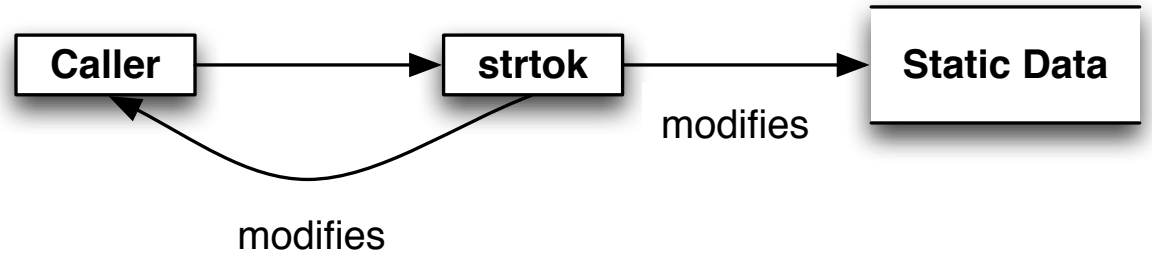
strtok Example

```
int main() {
    char search[BUFSIZ];
    strcpy(search,"This is 1just2a3test#.");
    char brkset[] = " \t\n\r\f\v`~!@#$%^&*()-_=+;:'\",<.>/”
                    "?01234567890";
    char* tokenptr = strtok(search,brkset);
    while (tokenptr != 0) {
        cout << tokenptr << endl;
        tokenptr = strtok(0,brkset);
    }
}
```

```
This\0is 1just2a3test#.
This\0is\01just2a3test#.
This\0is\01just\0a3test#.
This\0is\01just\0a\0test#.
This\0is\01just\0a\0test\0#.
```

```
This
is
just
a
test
```

The Dependencies of **strtok**



- What “wrong” here?

What's "Wrong" with **strtok**?

- It *modifies* the original search string
  - a *side effect!*
- It keeps *static data*
  - *another* side effect!
  - *shared* among *all calls* to **strtok**
  - calls from independent clients can't be interleaved
- Different calls to **strtok** have *different semantics*
  - when you pass 0 (NULL), it picks up where it left off
  - it is essentially *2 different functions*

An Improved Tokenizer

- Will *not* modify the caller's data
  - but how can we keep track of where we are?
- Will *not* use shared (static) data to track its state
  - but where will it put it?
- Will separate *initialization* from *iteration*
  - *different calls* for different actions
  - so we will need more that one function!

Another Try at **strtok**

```
struct Tokenizer {
    const char* search;
    const char* brkset;
    int pos;
};

Tokenizer* init_tok(const char* s, const char* brkset);

string next_token(Tokenizer* tok);

void del_tok(Tokenizer* tok);
```

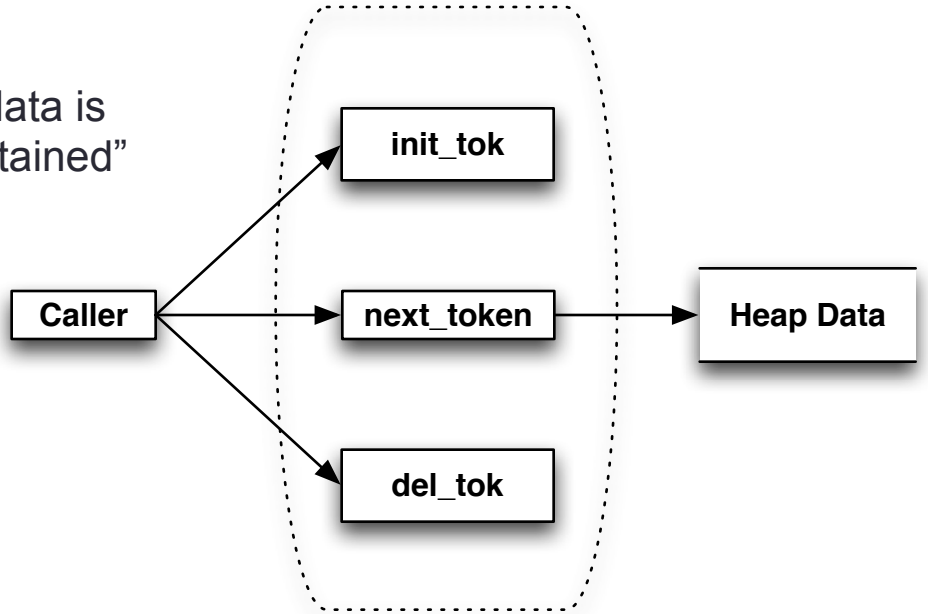
See code in *strtok2.cpp*.

Using Tokenizer

```
int main() {
    char search[BUFSIZ];
    strcpy(search,"This is 1just2a3test#.");
    char brkset[] = " \t\n\r\f\v`~!@#$%^&*()-_”
                    "=+;:'\",<.>/?01234567890";
    Tokenizer* tok = init_tok(search,brkset);
    string word = next_token(tok);
    while (!word.empty()) {
        cout << word << endl;
        word = next_token(tok);
    }
    del_tok(tok);
}
```

The Dependencies of Tokenizer

But the data is
“self-contained”



Using a *class* would be better
of course (see *strtok3.cpp*)

Reentrant Functions

- The problem with threads: race conditions on *shared data*
  - Using *critical sections* for shared data is a topic for another day
- **strtok** is *not thread-safe*:
  - it uses *static data* which is *shared by its very nature*
  - but each thread needs its *own copy*; FAIL
- Thread-safe functions must be *reentrant*
  - they “start from scratch” on each call;
  - no related static data

Principle: Keep Interfaces Small

- *Fat Interfaces* are often a sign that *too many things* are crammed into a single abstraction ("god objects")
  - *The Goldilocks Principle*: good cohesion requires things to be "just right"
  - Beware of *extremes* (including XP :-)

Principle: Keep Interfaces Small

- An abstraction should have *one key, conceptual reason* to exist
- Separate complex abstractions into smaller, logically independent abstractions

Principle: Keep Interfaces Small

- *Corollary*:

  "Every class should embody only about 3–5 distinct responsibilities." – *Booch*

Principle: Keep Interfaces Small

- *Corollary*:

  Minimize the number of function parameters (3-5)

Keyword Arguments

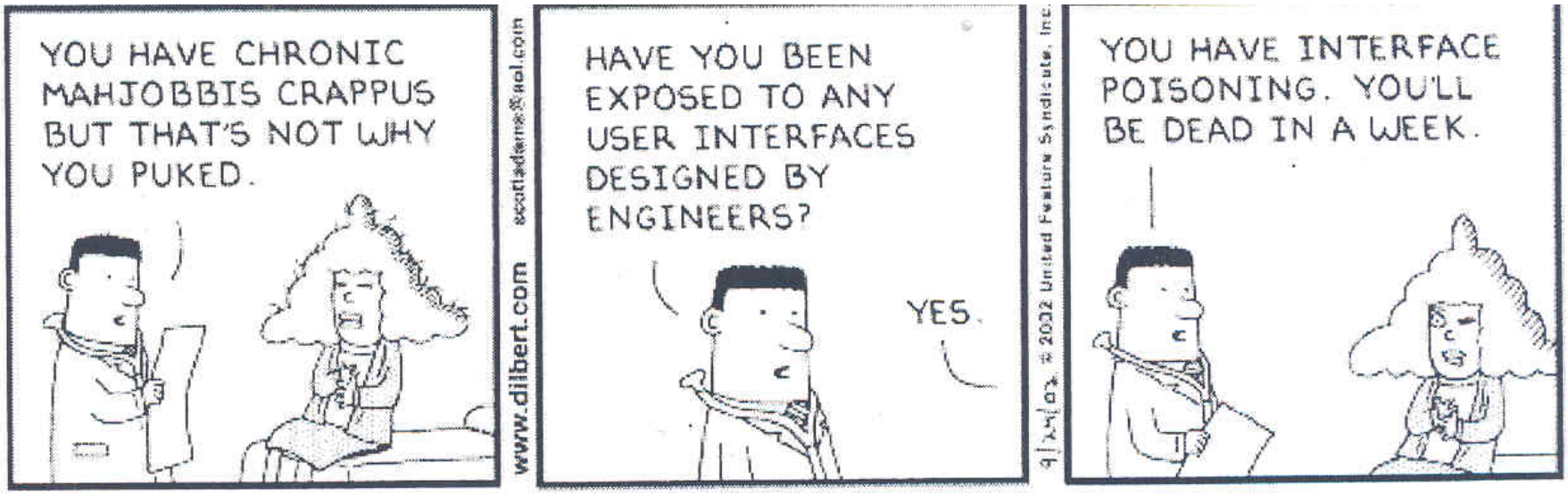
Python Example

- Large parameter lists are made easier with *keyword args*
- Example: ***sorted****(iterable[, cmp[, key[, reverse]]])*

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]

>>> sorted([5, 2, 3, 1, 4],reverse=True)
[5, 4, 3, 2, 1]

>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```


YOU HAVE CHRONIC MAHJOBBIS CRAPPUS BUT THAT'S NOT WHY YOU PUKED.
HAVE YOU BEEN EXPOSED TO ANY USER INTERFACES DESIGNED BY ENGINEERS?
YES.
YOU HAVE INTERFACE POISONING. YOU'LL BE DEAD IN A WEEK.
www.dilbert.com scottadams@aol.com
© 2002 United Feature Syndicate, Inc.

Interface Design

- Any client *interaction*
- Not just GUIs
- Most everything has an interface
  - functions
  - classes
  - modules and packages
  - …

A Date Class

Example from Scott Meyers

```
class Date {
public:
    Date(int month, int day, int year);
    …
};
```

How can a user misuse this class?

Using Types to Encourage Correct Usage

```
// Source: Meyers, S., The Most Important Design Guideline?, IEEE
// Software, July/August 2004
struct Day { int d; };
struct Year { int y; };

class Month {
    static const Month Jan = {1};
    static const Month Feb = {2};
//  …
private:
    explicit Month(int); // User can't create a Month
};

class Date {
public:
    Date(Day d, Month m, Year y) {…}
    Date(Month m, Day d, Year y) {…}
    Date(Year y, Day d, Month m) {…}
//  …
};
```

Interface Responsibility

- "*Responsibility* for interface usage errors belongs to the interface *designer*, not the interface user"

- *Principle*: "Make interfaces easy to use correctly and hard to use incorrectly"

— Scott Meyers

Principle: Don't Repeat Yourself (DRY)

*aka the law of **One Right Place***

- Another example of *poor cohesion*
  - Don't *scatter* the components of an abstraction
  - Repetitions need to be *collapsed into a single abstraction*

- "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." – *The Pragmatic Programmers*

DRY Damage

- The Software is Bigger
- Harder to Comprehend
- Harder to Fix
- Harder Change

A DRY Violation

- Suppose you are a C++ library developer, creating a *String class*
- You begin by encapsulating an array of characters
  - constructor
  - destructor
- After a simple test, you add:
  - copy constructor
  - assignment operator

String class: First Pass

```
class String {
    char* data;
public:
    String(const char* s = "") {
        data = new char[strlen(s)+1];
        strcpy(data,s);
    }
    ~String() {
        delete [] data;
    }
    operator const char*() const {
        return data;
    }
};
```

String Class: Second Pass

```
String(const char* s = "") {
    data = new char[strlen(s)+1];
    strcpy(data,s);
}
~String() {
    delete [] data;
}
String(const String& s) {
    data = new char[strlen(s.data)+1];
    data = strcpy(data,s.data);
}
String& operator=(const String& s) {
    if (&s != this) {
        delete [] data;
        data = new char[strlen(s.data)+1];
        data = strcpy(data,s.data);
    }
    return *this;
}
```

DRY Violations Sneak Up On You!

```
    char* clone(const char* s) {
        return strcpy(new char[strlen(s)+1],s);
    }
public:
    String(const char* s = "") {
        data = clone(s);
    }
    ~String() {
        delete [] data;
    }
    String(const String& s) {
        data = clone(s.data);
    }
    String& operator=(const String& s) {
        if (&s != this) {
            delete [] data;
            data = clone(s.data);
        }
        return *this;
    }
```

Some DRY Solutions

- Move common code into *functions*
- Move common class features into *base classes*
- Make code *generic* when applicable
  - type-independent code
- “If your doing something more than once, you’re probably doing something wrong.”
- [Source: Scott Meyers, *Better Software No Matter What*]

Measuring the Quality of Abstractions

Redux

- Sufficiency
  - capture enough of the idea to be useful
  - *minimal interface*
- Orthogonality
  - provide all needed *primitive operations* (that require access to internals for efficient implementation)
  - think twice about “convenience methods” that can be implemented with public methods

- “You don’t put the dishwasher in the bathroom”

— Grady Booch

Cogent Quote

- “A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads.” – *Dijkstra*, 1974

“Mr. Osborne, may I be excused? My brain is full.”

Separation of Concerns

A Natural Consequence of Cohesion

- "For the separation to be meaningful, we have also an *internal* and an *external* requirement."

Separation of Concerns

A Natural Consequence of Cohesion

- "The internal requirement is one of *coherence*: the knowledge must support the abilities and the abilities must enable us to improve the knowledge.

Separation of Concerns

A Natural Consequence of Cohesion

- “The external requirement is one of what I usually call "a *thin interface*"; the more self-supporting such an intellectual subuniverse, the *less detailed* the knowledge that its practitioners need about other areas of human endeavour, the greater its viability.”

– *Dijkstra*, 1974

Separation Principle #1

- “Separate *interface* and *implementation*”
- That horse is dead :-)

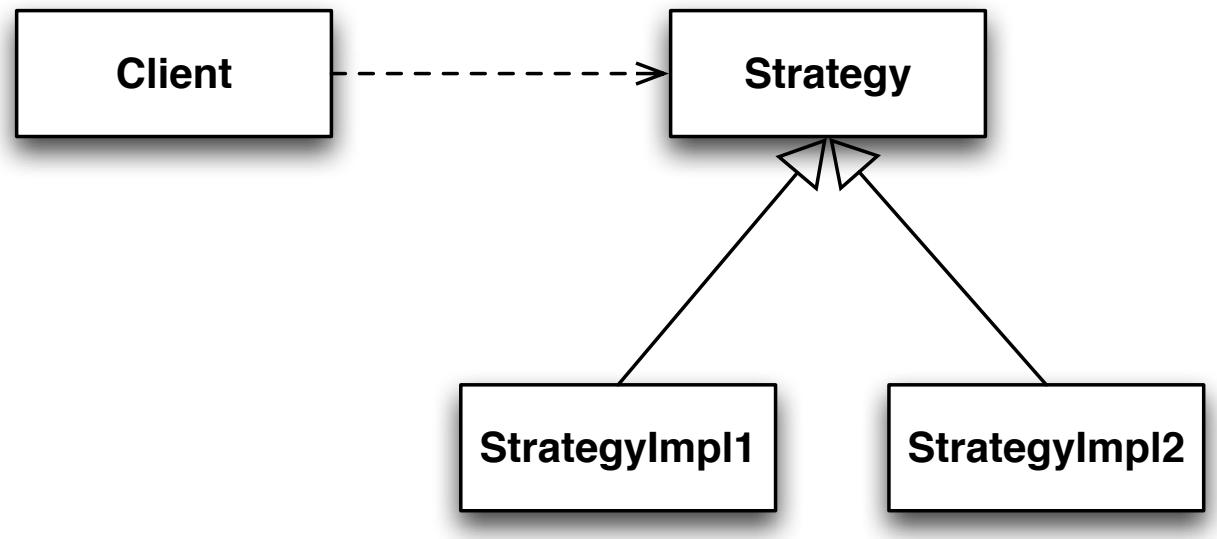
Separation Principle #2

- "Separate things that *change* from things that *stay the same*"
- *Interfaces* (usually) don't change; *implementation* does
  - So Principle #1 is actually a special case of Principle #2

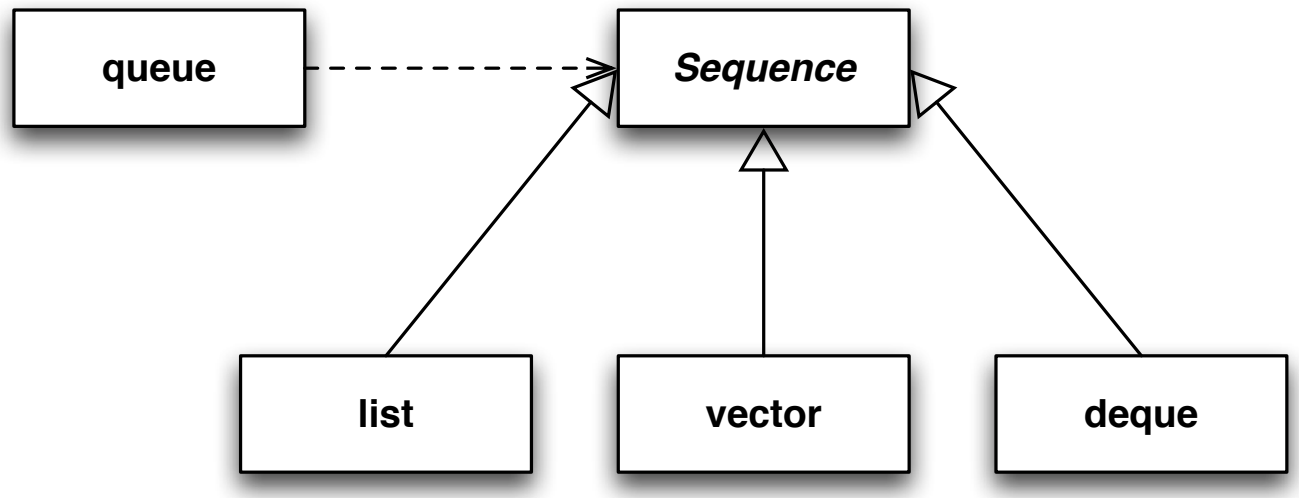
The Strategy Design Pattern

- “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets each algorithm vary independently from clients that use it.”
- Also Known As: *Policy*

Strategy Sketch



Implicit Strategy in C++




```
queue<int,list> q;    // Use a list as a storage policy
```

Separation Principle #3

- Separate *client* from *server*
- They're often *physically* separated anyway
  - different process
  - different computer
  - different network
- Example:
  - Database servers
  - CORBA, COM
  - The Web (duh!)

A Numeric Separation Example

- Classic algorithm for **sqrt(x)**:
  - start with an initial guess, g_1
  - compute next guess as:
    - $g_2 = ½ (g_1 + x/g_1)$
  - continue until the difference between guesses is “small”

```
def mysqrt(x,g1,tol):
    g2 = (g1 + x/g1)/2.0
    while abs(g2 - g1) > tol:
        g1 = g2
        g2 = (g1 + x/g1)/2.0
    return g2
```

Rate this design…

Is There Room for Improvement?

- Depends…

- There is no coupling to non-local data ✔
- There is no shared data ✔

- Hmmm. Maybe it's "perfect"

- How is its *cohesion*?

There are 2 Things At Play in **mysqrt**

- 1) Generating the *next guess*
- 2) Checking the *stopping criterion*
- Maximal cohesion says do only “one thing”
- How can these be separated?

Loosening the Coupling

- Iterating until a stopping condition is obtained is a very common operation
- Let's feed the sequence of guesses to a *separate*, generic iteration procedure
- Thus we will *loosen* the coupling between the two actions
  - by making the sequence of guesses a *parameter* to the iteration

The Sequence Generator

- An *unbounded* sequence

```
def sqrt_seq(x,g):
    yield g
    while True:
        g = (g + x/g) / 2.0
        yield g
```

The Iteration Procedure

- It decides when to quit

```
def iterate(seq, tol):
    last = seq.next()
    current = seq.next()
    while (abs(current-last) > tol):
        last = current
        current = seq.next()
    return current
```

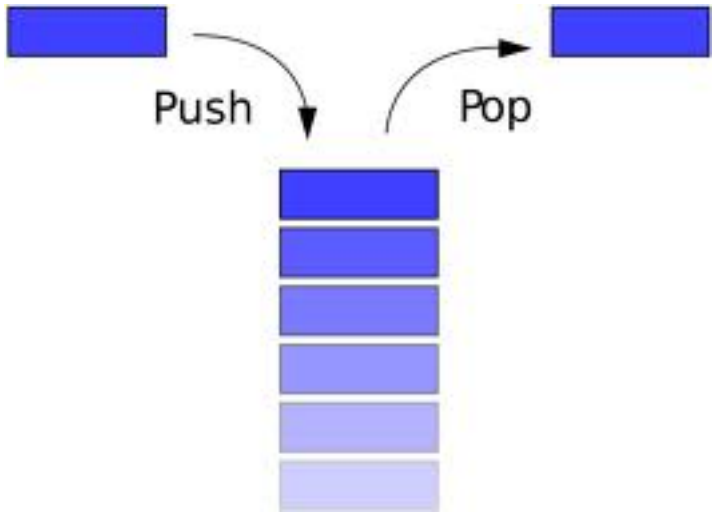
Using the New Arrangement

```
def mysqrt(x,g1,tol):
    return iterate(sqrt_seq(x,g1),tol)
```

- AND, we can now *reuse* **iterate** on any sequence!
- See *sqrt.cpp* for a C++ version…

The Behavior of C++'s **stack::pop()**

- It doesn't return anything!
  - it only removes the top element
- Why?



Separation Principle #4: Ownership

- *Principle*: Only one abstraction should "own" an object
  - The same layer of abstraction that allocates a resource is responsible to deallocate it

Separation Principle #4: Ownership

- *Corollary*: Separate the *creation/destruction* of an object from client *use* of the object
- Example: *Factory Design Pattern*

Separation Principle #4

- *Corollary*: Minimize the *ownership* of objects

Object Management

– Tom Cargill

1. Creator as Sole Owner
2. Sequence of Owners
   - 1 at a time
3. Shared Ownership
   - simultaneous access

C++ 2011's **unique_ptr**

```
void function()
{
    unique_ptr<Resource> owner(res);
    owner->do_resource_stuff();
    unique_ptr<Resource> new_owner = owner;
    new_owner->do_resource_stuff();
}

// At exit, res is cleaned up when
// new_owner is destroyed
```

Shared Ownership

*C++ 2011's **shared_ptr***

```
class Connection {
public:
    Connection() {cout << "connecting...\n";}
    ~Connection() {cout << "disconnecting...\n";}
};

class Client {
    shared_ptr<Connection> ptr;
public:
    Client(shared_ptr<Connection>& p) : ptr(p) {}
};
```

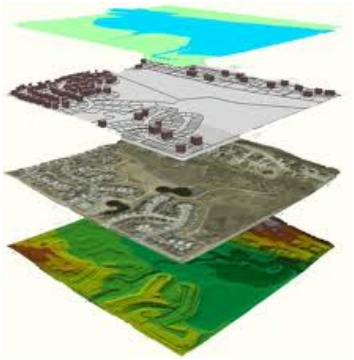
```
int main() {
    shared_ptr<Connection> conn(new Connection);
    cout << conn.use_count() << endl;
    {
        Client c1(conn);
        Client c2(conn);
        cout << conn.use_count() << endl;
    }
    cout << conn.use_count() << endl;
}
```

```
connecting...
1
3
1
disconnecting...
```

Layered Abstractions

- "All well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface." – *Booch*

- "The design principle … of separating the interface from the implementation, should be applied rigorously at each layer's boundaries." – *CSC Report*



Examples of Layering

- A *layer* is a *group of components* that are reusable in similar circumstances
- 3-tier Architecture
  - Presentation ⟹ Business Rules ⟹ Persistence
- OSI Model

The Law of Demeter

aka Principle of Least Knowledge

- Layers are themselves *abstractions*
- Clients should not “cross” abstraction boundaries
  - but should use *interfaces*
- Anything beyond more than 1 boundary is a “stranger”
- Law of Demeter: *Don’t Talk to Strangers*

The Law of Demeter

Guidelines

- A method should only access:
  - Fields in its object
  - Methods in its class
  - Parameters it receives
  - Anything it creates dynamically
- Beware multiple “dots”:
  - `o1.o2.o3.m()`
  - `o.m1().m2().m3()`

Separation Summary

- “Go ahead and chop your [code] into modules, but do it along the seams.” – *PJP*
- In other words, “maximize cohesion”

What We've Heard

"Solutions should be as simple as possible, but no simpler"

- Who said this?

What Einstein Really Said

“It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.”

- *Philosophy of Science*, Vol. 1, No. 2 (April 1934), p. 165.

Simplicity

- “The discovery of fundamental principles leads to a simpler conceptual model.”

— Booch

Focus On Simplicity

- “There are two ways of constructing a software design: one way is to make it so simple that there are *obviously no deficiencies* and the other is to make it so complicated that there are *no obvious deficiencies*.” – *Tony Hoare*

- *Principle*: Prefer the former :-)

Focus on Simplicity *Early*

- "Since the essence of programming is controlling complexity, nothing lowers the cost of debugging and maintaining code so much as eliminating unnecessary logic *as early as possible*." – *P. J. Plauger*

- YAGNI

- "The Simplest Thing That Could Possibly Work"

The Importance of Elegance

- Elegance = Simplicity
  - no *cruft*
  - not *cryptic*
- Elegance/Simplicity accommodates the *humans* that must read and maintain code.

The Importance of Elegance

- "Users … are indifferent to the need for elegance – until they later get bitten by its lack. Programs silt up over time. The cleaner you make them up front, the longer they last." – *P. J. Plauger*

The Degradation of Software Quality

"Architecture degradation begins simply enough. When market pressures for key features are high and the needed capabilities to implement them are missing, an otherwise sensible engineering manager may be tempted to coerce the development team into implementing the requested features without the requisite architectural capabilities."

-- Luke Hohmann, *Beyond Software Architecture*

Technical Debt

- Market and budget pressures may “require’ cutting corners
- Changing existing code to meet changing requirements can introduce “design smells”
- These constitute a “technical debt”
  - The software is not in a state suitable for long-term maintenance
  - Leaving these debts “unpaid” results in *Software Entropy*
  - Such systems are *abandoned* before their time

The Need for Refactoring

- Restores Simplicity
  - “clean code”
- “Post Release Entropy Reduction”
  - Will see later in “Applying Design Principles”

Measuring Simplicity

- “The fastest way to discover whether or not you have invented a simple program structure is to try to describe it in completely readable terms… if you discover that there is no simple way of describing what you intend to do, then you should probably look for some other way of doing it.”

  – *Per Brinch Hansen*

- “If we can’t explain a concept to college freshmen, we really don’t understand it.”

  – *Richard Feynman*

Summary of Fundamentals

- "*Modularity* is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules"

- Strive for Modularity

THE S.O.L.I.D. PRINCIPLES OF OO DESIGN

"Object-orientation is fundamentally a set of *design principles*, such as the separation of layers and modelling the abstraction of the business. It is possible to make use of these principles at a variety of levels: from the high-level business capabilities of the system to the low-level technological implementation. The greatest benefit comes when the principles are consistently applied at each of these levels."

– *CSC Report*

Design Smells

– Robert C. Martin (aka Uncle Bob)

- You usually can tell when a design is poor…

- Rigidity – one change leads to (too) many others (ripple effect)
- Fragility – changing one component breaks unrelated components
- Immobility – can't separate components for reuse (tangled mess)
- Viscosity – hard to do things right (code inertia)
- Needless Complexity – infrastructure that adds no benefit
- Needless Repetition – DRY violations
- Opacity – hard to understand; poor expression of purpose/intent

S.O.L.I.D. Principles

– Uncle Bob

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

- Intuitively, this is just a cool name for *cohesion*
- Robert Martin gives it a different slant:
  - "A class should have only one reason to *change*"
- Or more explicitly:
  - "Gather together those things that change for the same reason, and separate those things that change for different reasons."
- If you find that you are changing a class for seemingly unrelated reasons, your class may not be cohesive
  - i.e., your code is *Fragile*

SRP Example

From "97 Things Every Programmer Should Know"

```
public class Employee {
    public Money calculatePay() …
    public String reportHours() …
    public void save() …
}
```

- A very common design
- What would cause changes to this code?

SRP: Redesigning Employee

```
public class Employee {
    public Money calculatePay() …
}
public class EmployeeReporter {
    public String reportHours() …
}
public class EmployeeRepository {
    public void save() …
}
```

- Separate components:
  - business rules
  - reporting
  - persistence

Open-Closed Principle (OCP)

– Bertrand Meyer

- "Software entities should be open for extension but closed for modification"

- Extend software with *new abstractions*
  - not by changing existing code
  - Otherwise your code smells of *Rigidity*

- Following the OCP:
  - program to an interface
  - the Decorator Pattern

OCP Violation

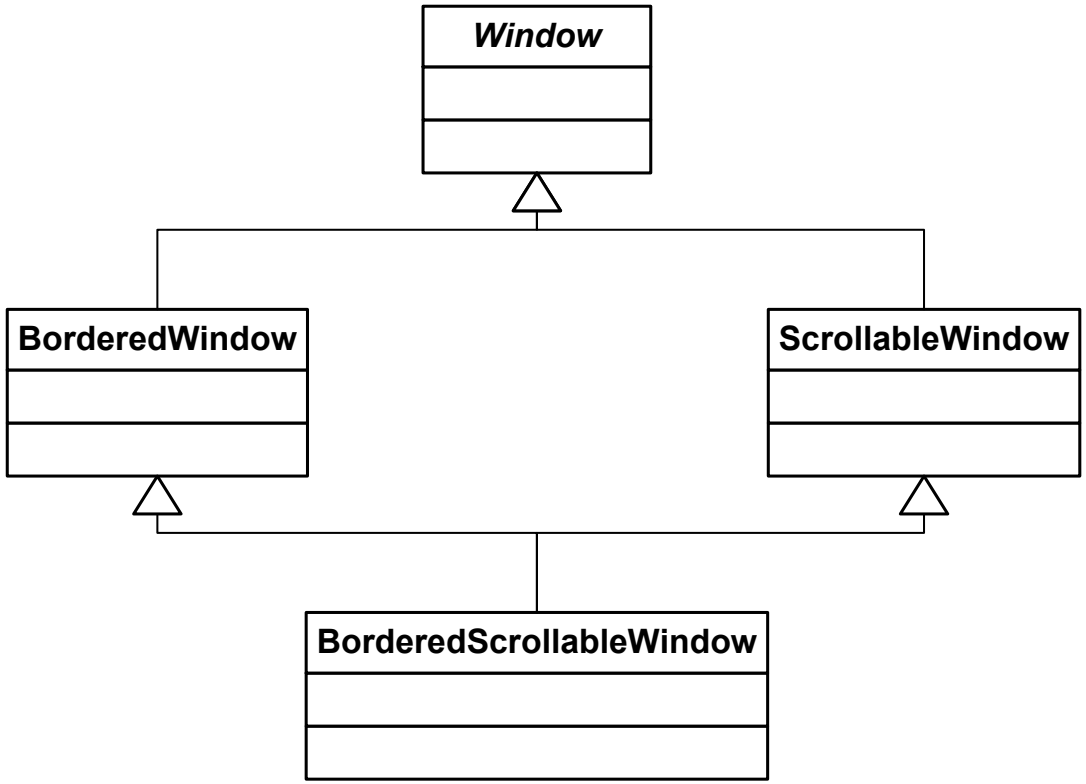
From "Head First Design Patterns"

- Consider a GUI type named **Window**
  - Unadorned, but functional
- Now suppose we want some more full-featured windows
  - Bordered, scrollable, etc.
- How do we design this?

OO Design 101

- A **BorderedWindow** is most assuredly a **Window**
  - Sounds like an “is-a” to me!
- Ditto **ScrollableWindow**
  - Sort of obvious, no?

A Simple Hierarchy



Evaluating Our Design

- Ignore details of multiple inheritance
  - We can always work around that
- Any other problems?

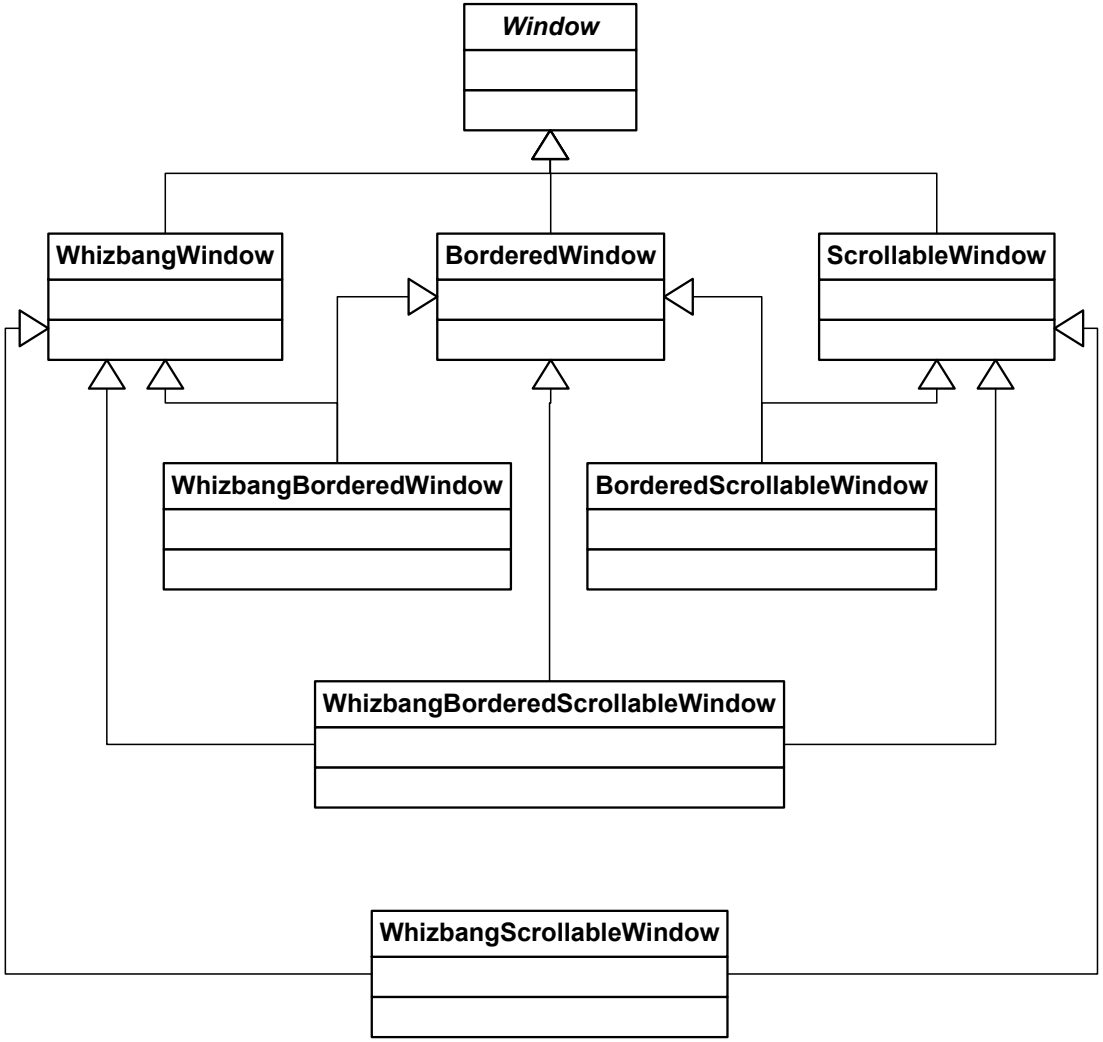
Problem #1

- The subclasses have operations that the **Window** superclass doesn't
  - **scroll**, for example
  - Not completely an "is-a"
    - But it isn't unusual for a subclass to add operations; no biggie
- We could put these methods in **Window**
  - But they'd be no-ops in the subclasses that don't use them
  - Someone isn't *encapsulating variation*!

Problem #2

- What if we need to add another important, independent windowing feature?
  - **WhizbangWindow**
- What impact does this have on the hierarchy?

Hierarchical “Progress”



Definitely Counting Classes

- 1 (= C(3,0)) for the root
- 3 (= C(3,1)) for the first row
  - Single-featured
- 3 (= C(3,2)) for the second “row”
  - Double-featured
- 1 (= C(3,3)) for the leaf
  - All three

- Total of 8

Looking Ahead

- C(n,0) + C(n,n-1) + … + C(n,1) + C(n,0)
- Equals 2^n
- Can anyone say “combinatorial explosion”?

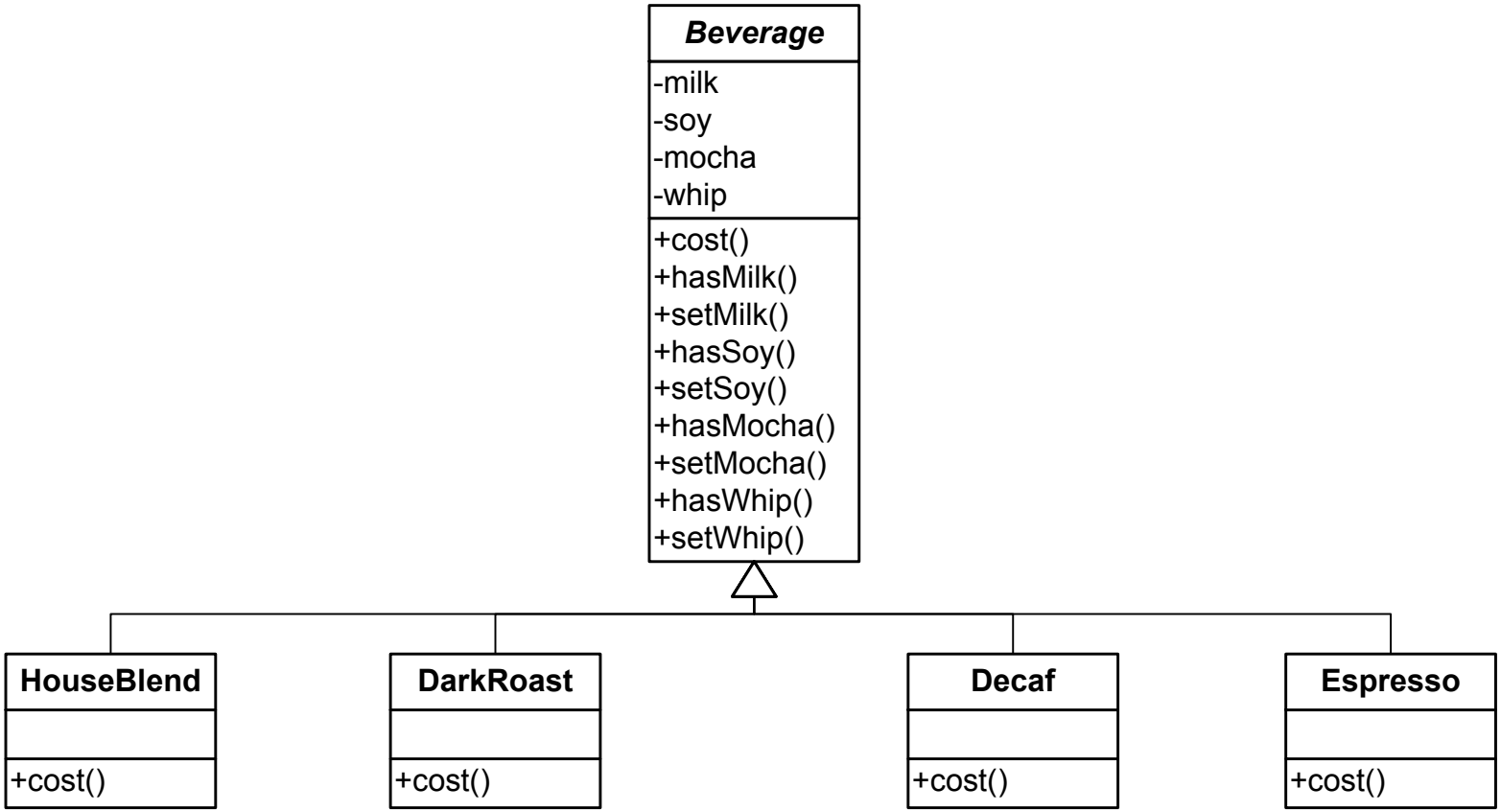
Starbuzz Coffee in HFDP

- Has 4 concrete specializations of the **Beverage** type
  - **HouseBlend**, **DarkRoast**, **Decaf**, **Espresso**
- Also, there are four “features” that can adorn these drinks
  - **SteamedMilk**, **Soy**, **Mocha**, **WhippedMilk**
- $4 * 2^4 = 64$ clunky classes to look forward to!

Another Attempt

- These “features” (condiments) really feel like *attributes*, don’t they?
- So let’s just move them to **Beverage**
- The concrete classes can compute the correct cost
  - See next slide

A Better Hierarchy?



Evaluating Take #2

- **Beverage.cost( )** must check for the presence of each condiment
- As new condiments arrive, **Beverage**'s code must change *noticeably*
  - New fields and methods, additional processing
- We would rather be able to add new condiments without changing **Beverage** at all!
  - "Separate what varies"
  - At *runtime*!

Evaluating Take #2

continued

- A new beverage may arrive that does not accommodate all condiments
  - Who ever heard of *mocha tea*?
- So we may find ourselves *removing* functionality in a derived class
  - Violates “is-a”
  - Can even result in repeated code
    - As many “no-ops” will pop up

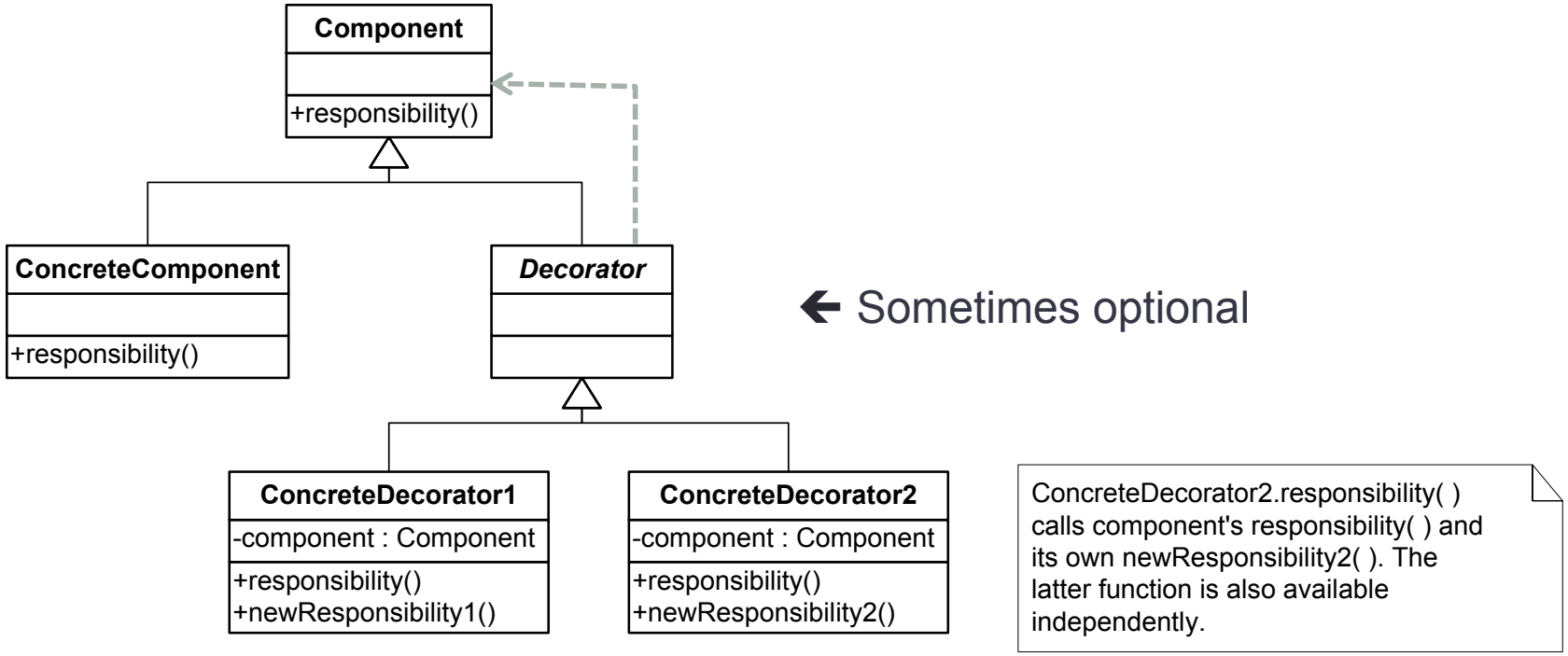
Using the Open-Closed Principle

- *Extend* the class; don't change its code
- The Decorator Design Pattern fits here

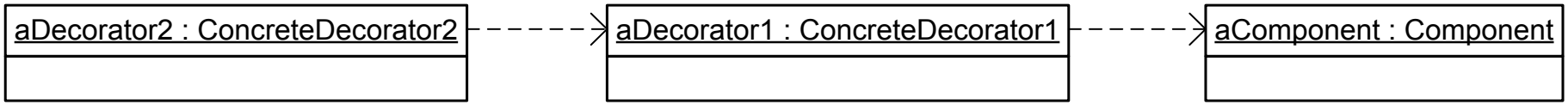
The Decorator Design Pattern

- Intent:
  - Add additional responsibilities to an object *dynamically*.
- Context:
  - Applies when clients program to an abstraction. A decorator can be used in the same way as its subject. The set of additional responsibilities can be *open-ended*. Existing subject code does not change. Multiple decorators can be *combined sequentially.*
- Solution:
  - The Decorator implements the *same interface* as the abstraction. It manages a concrete instance of the abstraction via *composition*, and adds additional functionality. It calls the managed object's methods and combines the result with the additional functionality.

Class Sketch

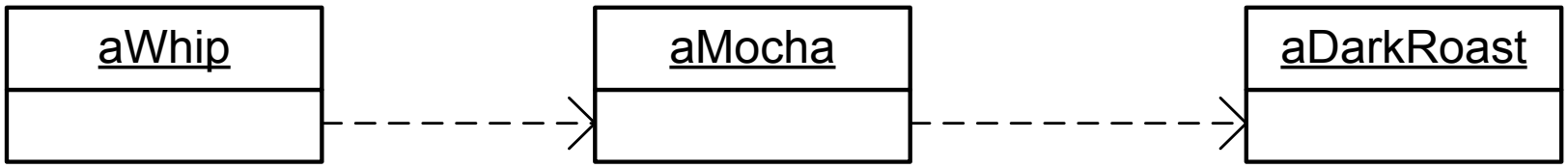


Object Sketch



Decorators and StarBuzz

Object Sketch View



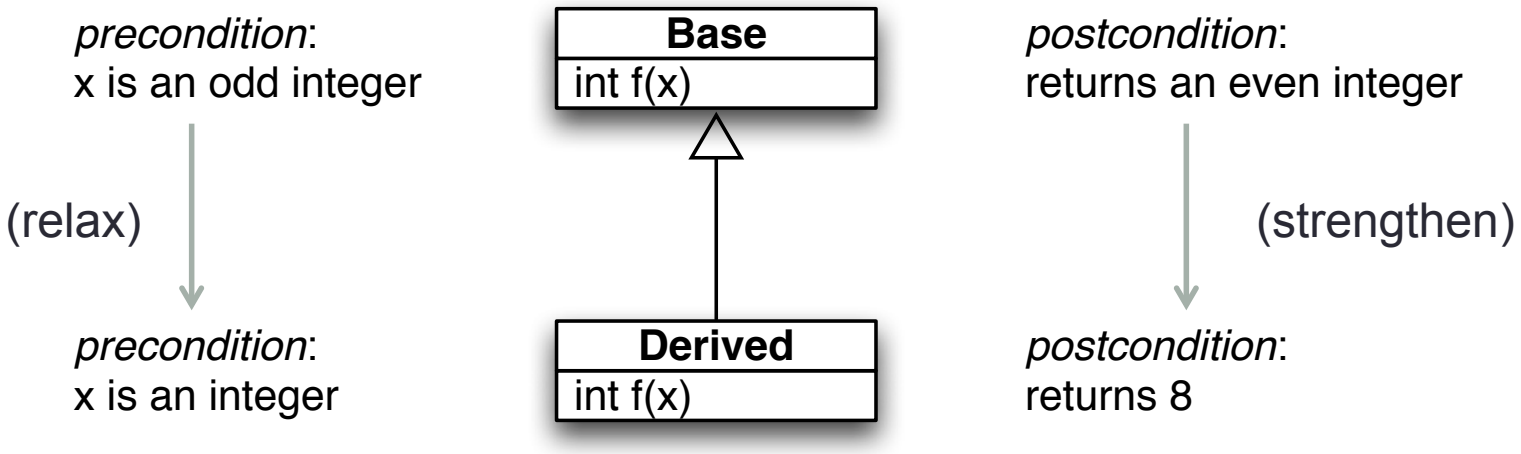
The Liskov Substitution Principle

- “Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.”
- *Derived objects should be substitutable base objects*
  - because of the “is-a” relationship
- This principle must be followed when *overriding functions*
  - Derived classes must not change the “rules of the game”

Abstractions and Inheritance

- "Contracts" are set by the *base class interface*
  - Derived classes must *obey* the base contract
- Using a contractor-customer analogy:
  - Subcontractors must *not charge more* than originally agreed upon
  - Subcontractors must *deliver at least* what was agreed upon
- Clients program to the *contract*
  - By using and understanding the *base class interface* and its *conditions*

Sample Contract Specification



Contract Conditions and Inheritance

- Preconditions are *contravariant*
  - e.g., assumptions about *method arguments*
  - they can be *relaxed* in derived classes
  - but *not* made more strict
- Postconditions are *covariant*
  - e.g., assumptions about *return values*, *exceptions* thrown
  - they can be *strengthened* in derived classes
  - but *not* weakened
- *"Require no more; Promise no less"*

The Problem with Fat Interfaces

- They try to be “all things to all people”
  - “Interface Pollution”
- Design Smells
  - *Rigidity* – clients dependent on only *part* of the functionality are affected when other parts change
  - *Immobility* – hard to disentangle functionality into independent, reusable components
  - *Needless Complexity* – Unused functionality

The Interface Segregation Principle (ISP)

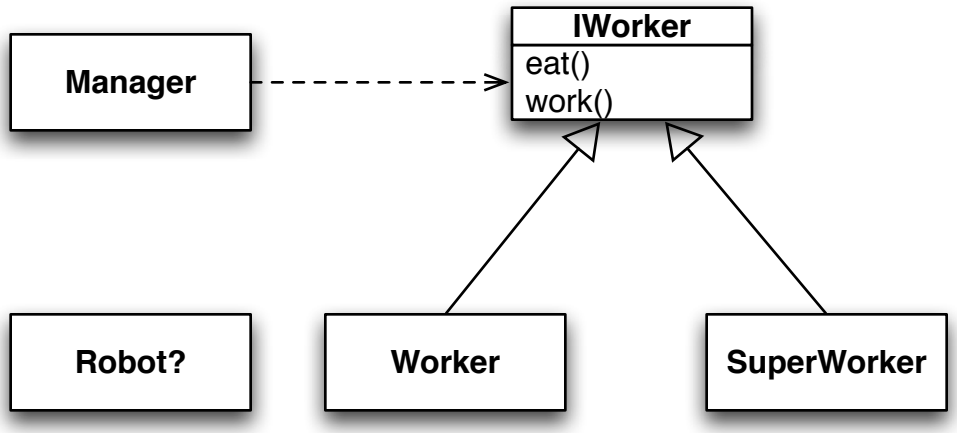
– Robert C. Martin

- "Many client specific interfaces are better than one general purpose interface"
  - *Fat interfaces are bad* ("Interface Pollution")
  - "Separate clients means separate interfaces"

- "Clients should not be forced to depend on methods that they do not use"

ISP Violation

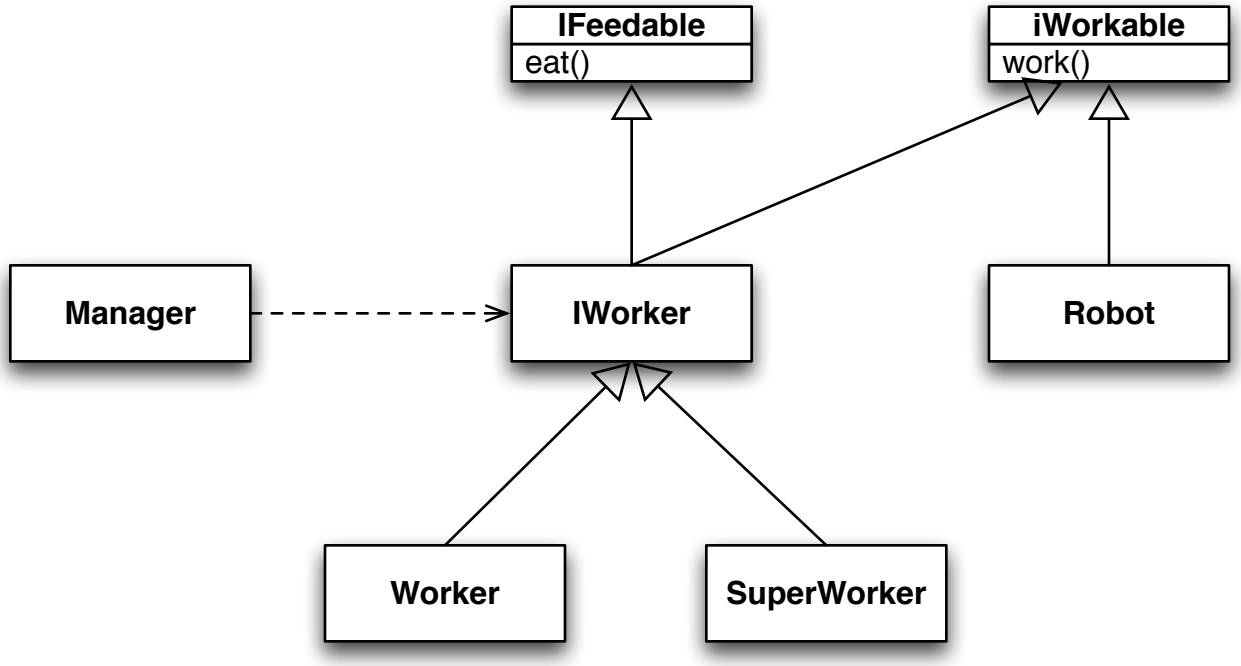
Adapted from oodesign.com



Where does the Robot fit?
IWorker is not cohesive in this context.

A Better Design

Separate Interfaces for Separate Clients

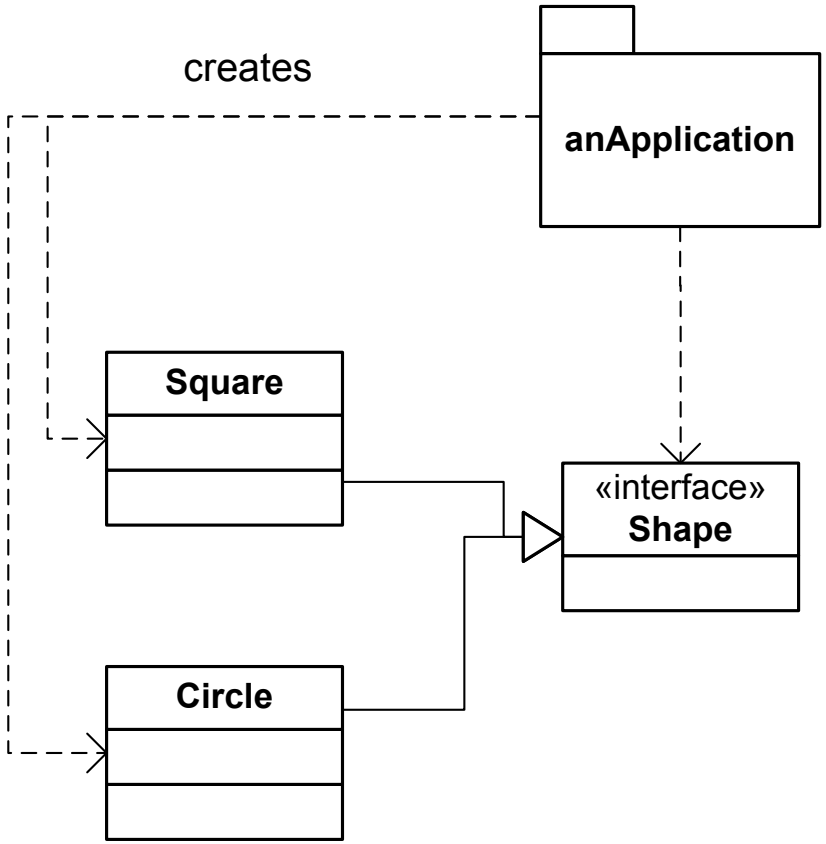


Dependency Inversion Principle

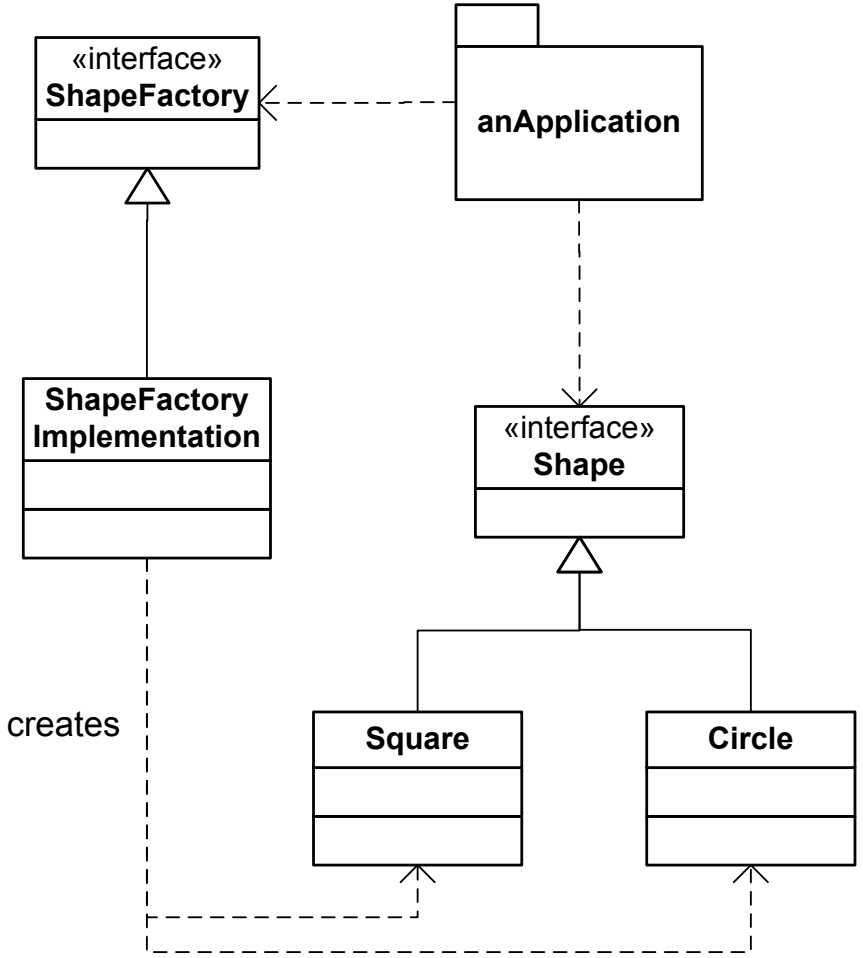
-- Robert C. Martin

- High-level modules should *not* depend on low-level modules
  - Both should depend on *abstractions*
- Abstractions should *not* depend on details
  - Details should depend on abstractions
  - Program to the “metaphor”
- Why?
  - Clients of *abstractions* are insulated from changes
  - *Reuse* happens at higher levels, so keep things clean

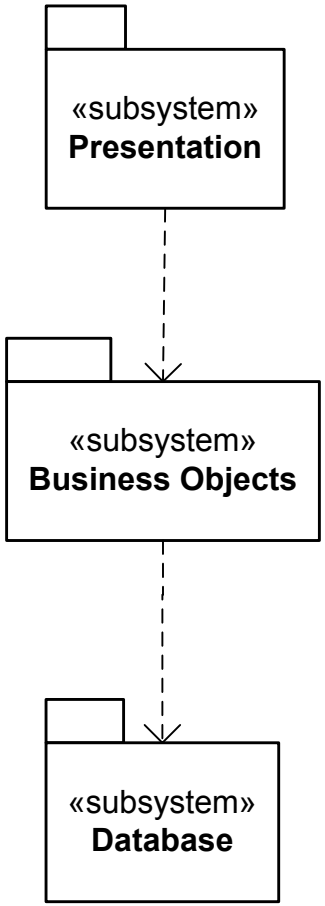
Another DIP Violation



A Better **Shape** Scenario



3-tier Architecture

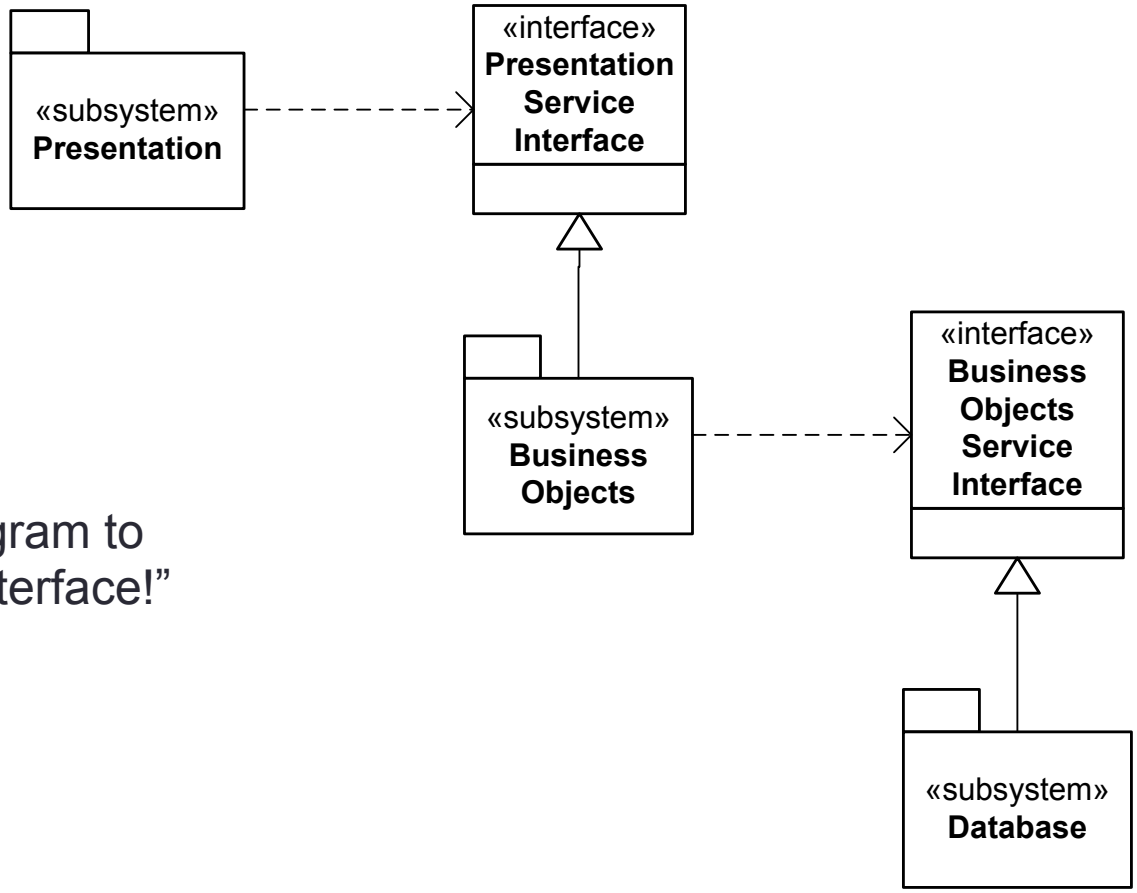


N-tier Architecture

Critique

- Appears to violate the DIP
- This is solved by having high-level modules write to an *abstraction* of its lower-level service module
  - So implementation can vary and not disturb the high-level client
- (See next slide)

Following the DIP



"Program to an interface!"

APPLYING DESIGN PRINCIPLES

It's All About Balance

- You don't use a principle just because you *can*
  - use it only when it applies
  - to maximize cohesion, maintain simplicity
- "Overconformance" to principle is not a virtue
  - and creates needless complexity
- Applying a principle is often a natural consequence of *incremental development* or *evolving requirements*
  - "Design Smells" appear with code modification

Removing Design Smells

- 1) Detect the Smell
- 2) Discover the violated principle(s)
- 3) Redesign according to the principle(s)

Practicing Simplicity

- "Once you appreciate the value of description as an early warning signal of unnecessary complexity it becomes self-evident that program structures should be described (without detail) *before* they are built and should be described by the *designer* (and not by anybody else). *Programming is the art of writing essays in crystal clear prose and making them executable*."

  – *Per Brinch Hansen*

An Application Principle

- “A complex system that works is invariably found to have *evolved from a simple system* that worked… A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.”

— Gall, J. *How Systems Really Work and How They Fail*, 1986.

Stream of Prototypes

- Design and Implement a *small subset first*
- Repeat…
  - Iterative Development
  - Evolving Design Decisions
- “The purpose of the Evolutionary Phase is to grow and change the implementation through successive refinement, ultimately leading to the production system” — Booch


ROAD
WORK
AHEAD


END
ROAD WORK
BIG BINDER
EXPRESS,
72192

Things to Minimize

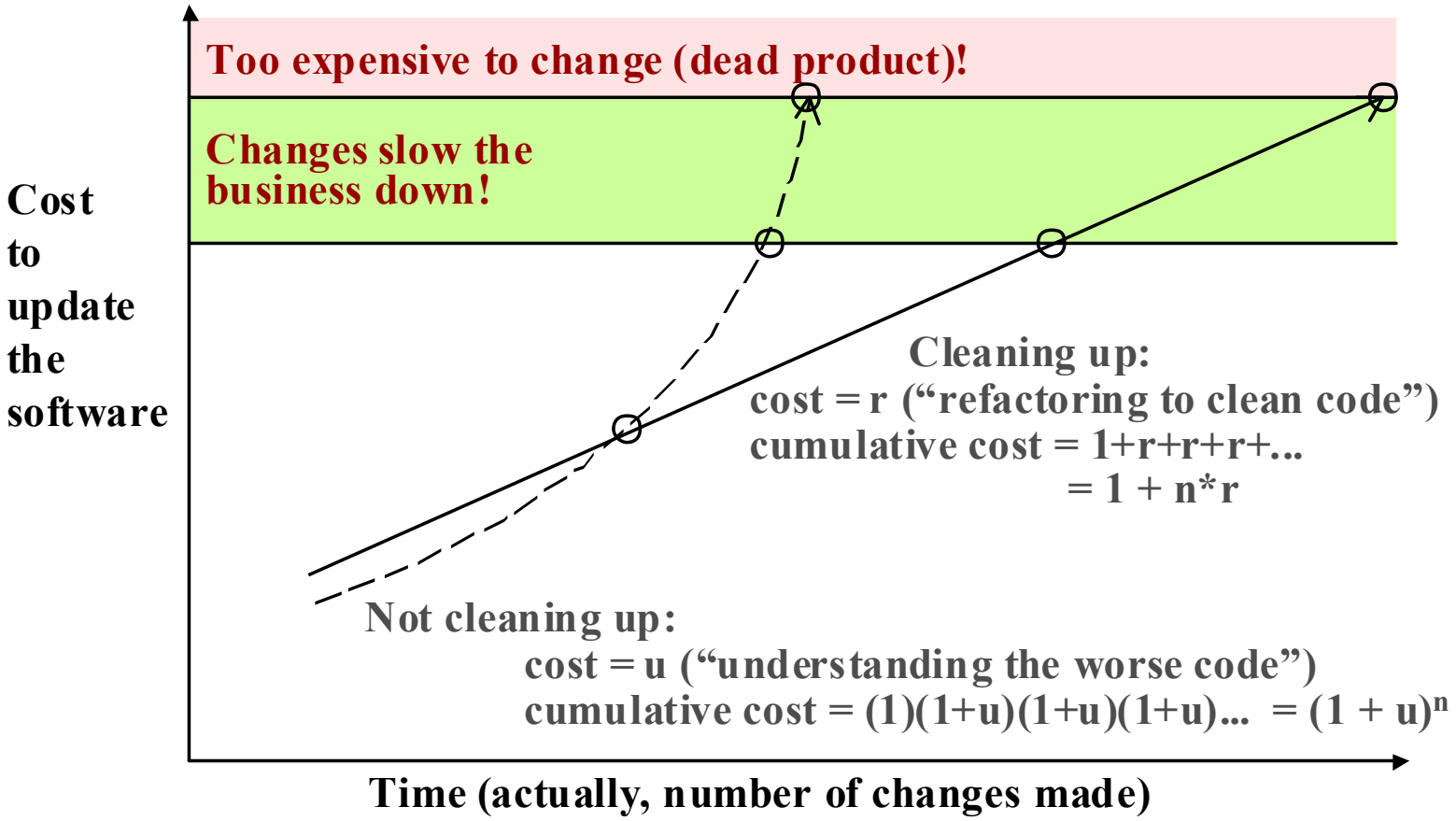
- Coupling
- Non-local data
- Scope
- Lifetime
  - initialize late; destroy early
- Number of Parameters
- Number of Methods

Technical Debts Interact

– Alistair Cockburn

- First maintenance visit: $1 + u$
- Second visit: $1 + u + v + uv$ (interaction)
  - approximates $1 + 2u + u^2 = (1 + u)^2$
- Third visit: $1 + u + v + w + uv + uw + vw + uvw$
  - approximates $1 + 3u + 3u^2 + u^3 = (1 + u)^3$

Poor code quality penalizes exponentially;
Cleaning up penalizes linearly.



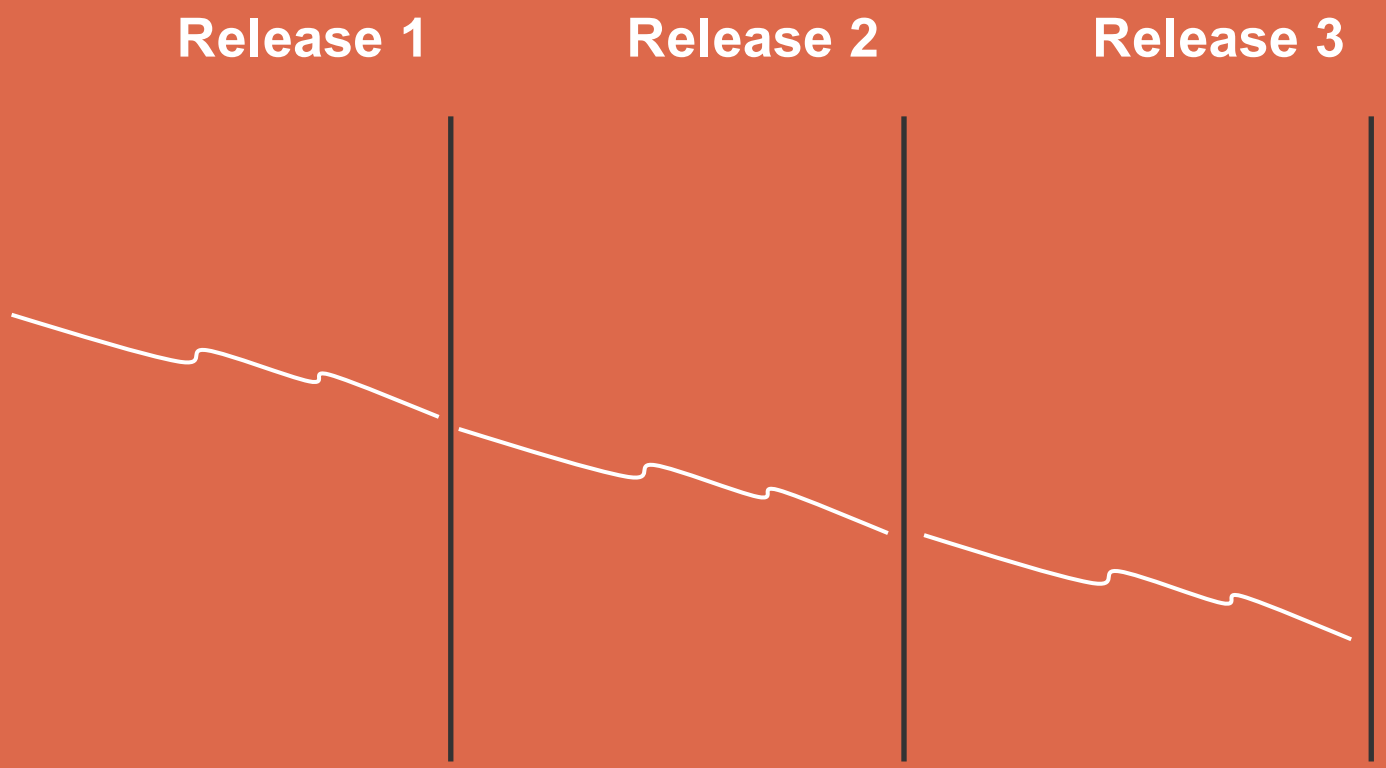
Post-Release Entropy Reduction

– Luke Hohmann

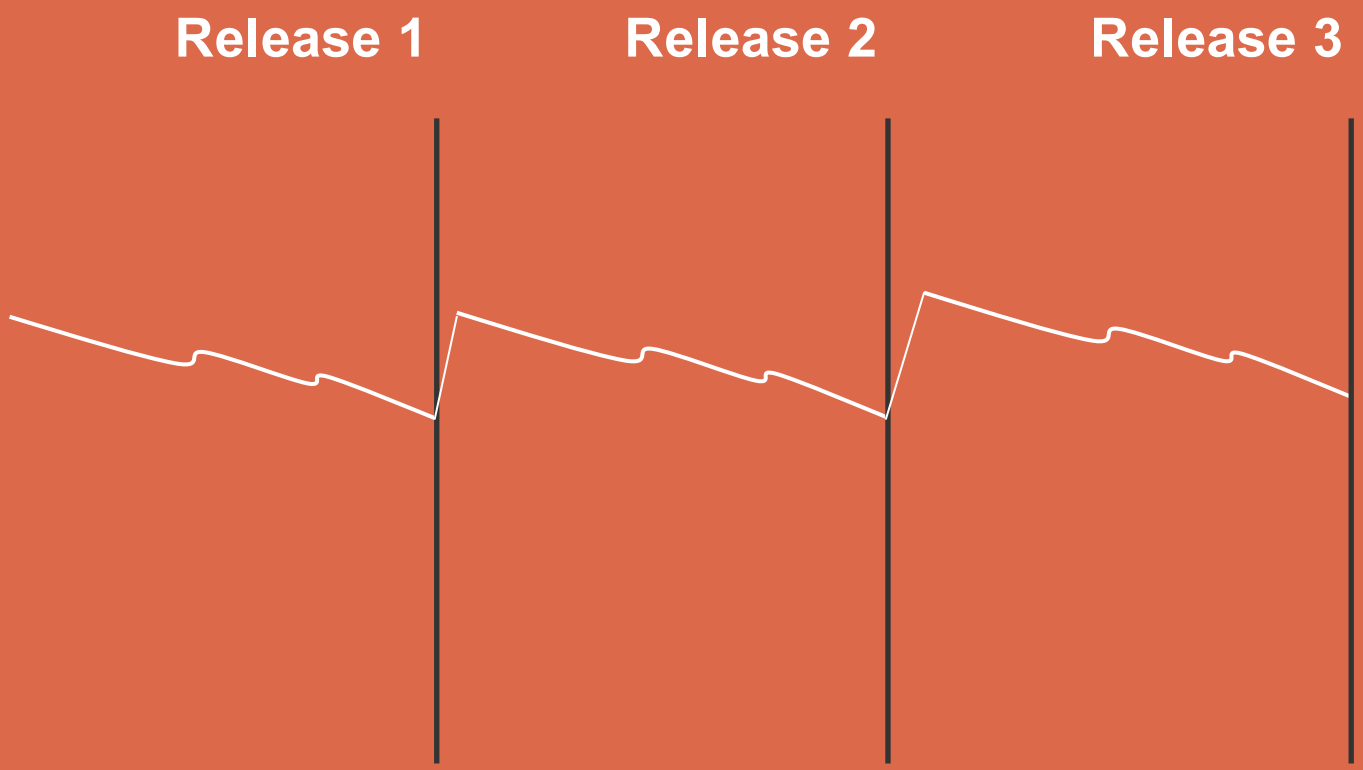
- No time for refactoring near release
  - Quick hacks pay off near release time
  - Market windows *are* important
- There must be time to *refactor* after release
  - Otherwise entropy kills you prematurely
  - Yes, there is a short-term cost
  - The long-term reduction of technical debt is worth it!

Code Quality Without PRER

– Randy Stafford



Code Quality With PRER



First Things First

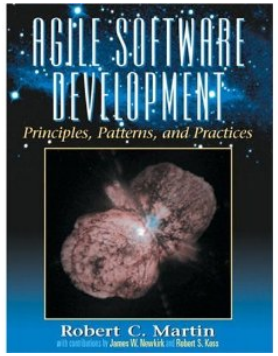
- *Principle*: Design for *normal* usage first; design exception handling *later*

The Need for a Chief Architect

- "An architect determines where the walls go; an engineer determines how strong to make the walls; a contractor builds the walls." -- *Allen Holub*

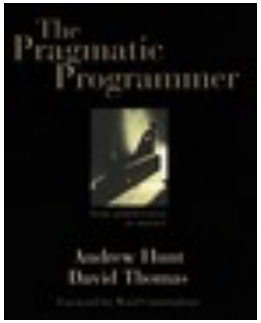
- "The architect owns the vision of what the new kind of system represents… is the owner of the *overall concept* of the system … and of the *set of design principles* needed to implement that concept."
- "A good architect knows the trades and 'walks the halls' during construction and finishing to ensure that the same design principles are being enforced at all levels."
  - -- *CSC Report*

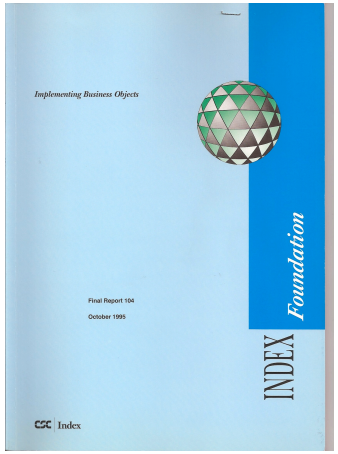
References


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin

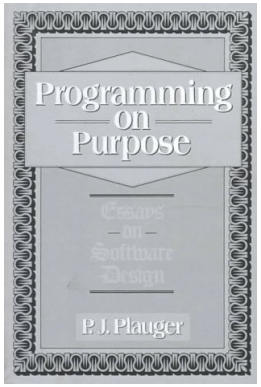

A Brain-Friendly Guide
Head First
Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman


OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS
THIRD EDITION


The Pragmatic Programmer
Andrew Hunt
David Thomas


Implementing Business Objects
Final Report 104
October 1995
INDEX
Foundation
CSC Index


BJECT-ORIENTED
SECOND EDITION


Programming on Purpose
Essays on Software Design
P. J. Plauger

More References

- Esther Schindler, “5 Things Grady **Booch** Has Learned About Complex Software Systems”, *CIO.com*, May 29, 2008
- Per Brinch **Hansen**, *Architecture of Concurrent Programs*, Prentice-Hall, 1977.
- Tom **Cargill**, “Localized Ownership: Managing Dynamic Objects in C++”, in J. Vlissides, J. Coplien, and N. Kerth, eds., *Patterns Languages of Program Design*, vol 2, Addison-Wesley, 1996.
- Edsger W. **Dijkstra**, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0–387–90652–5.
- Christopher **Strachey**, *Fundamental Concepts in Programming Languages*, Higher-Order and Symbolic Computation, 13, 11–49, 2000