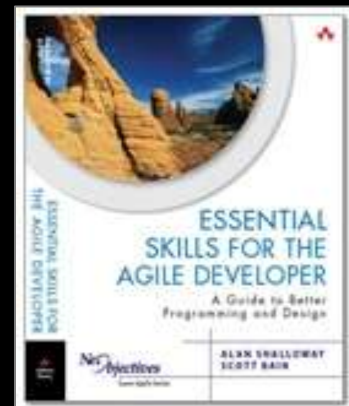
Lean Agile

Essential Skills for the Agile Developer



Net Objectives

© copyright 2010. Net Objectives, Inc.

AlShall
@NetObjectives.com

@alshalloway

WELCOME

Lean for Executives
Product Portfolio Management

Business

Lean Enterprise

Team

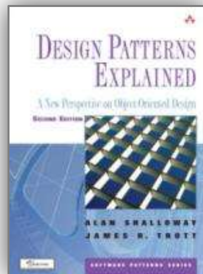
Manage ment

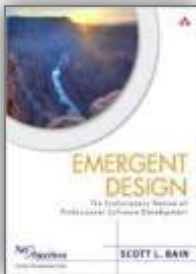
Kanban / Scrum
ATDD / TDD / Design Patterns

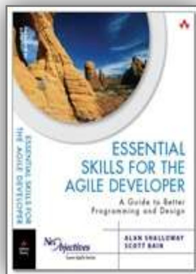
Lean Management
Project Management

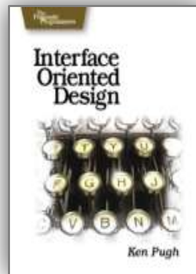
Net Objectives

ASSESSMENTS
CONSULTING
TRAINING
COACHING









The Thought Process of Patterns

1. **What are design patterns?**
2. Programming by Intention
3. Separating Use from Construction
4. Define tests up front
5. Shalloway's Law
6. Encapsulate that!

- How they help us
- The problems they are designed to solve
- Advice from the Gang of Four

thinking points

Which Team?

Team A – Green field

Team B – Extending existing system?

What is the bane of developers?

Design is not about planning

it's about learning

The Essence of Patterns

- Encapsulate what is not known
- What is not known is how things will change
- Make it possible to change things with low risk

Advice from the Gang of Four

Gang of Four Gives Us Guidelines*

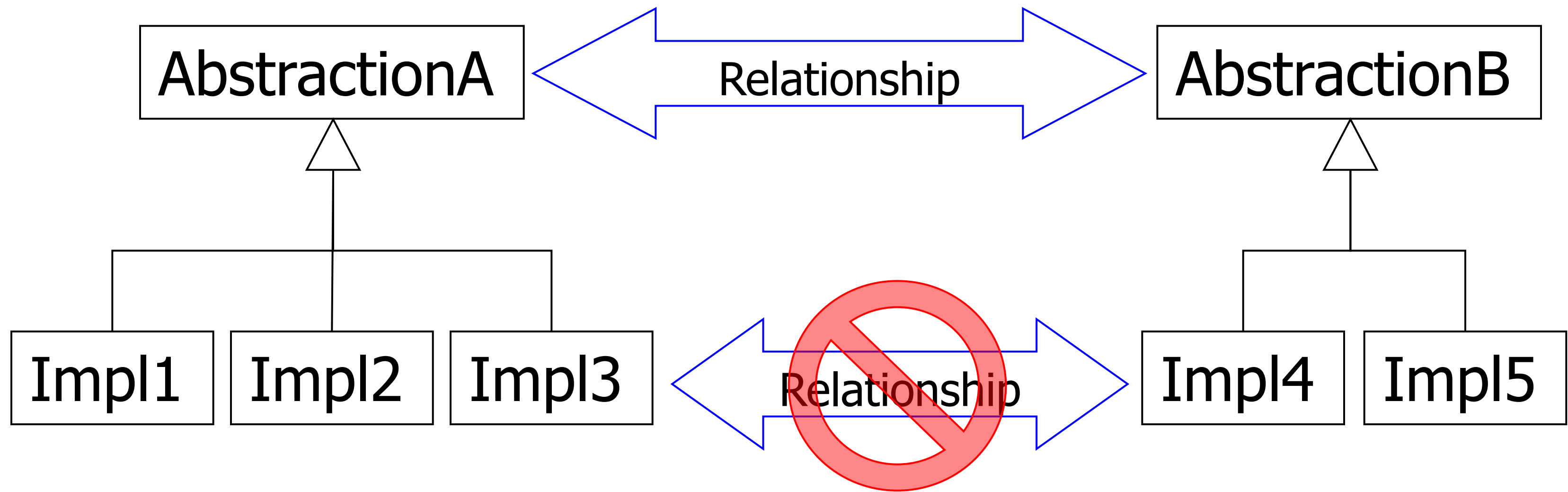
- Design to interfaces
- Favor object delegation over class inheritance
- Consider what should be variable in your design … and "encapsulate the concept that varies"

***** Gamma, Helms, Johnson, Vlissides: The authors of ***Design Patterns: Elements of Reusable Object-Oriented Design***

Design to Interfaces: Methods

- Craft method signatures from the perspective of the consuming entities
- Hides the implementation of your methods
- Programming by intention is a systematized way of designing to interfaces

Design to Interfaces



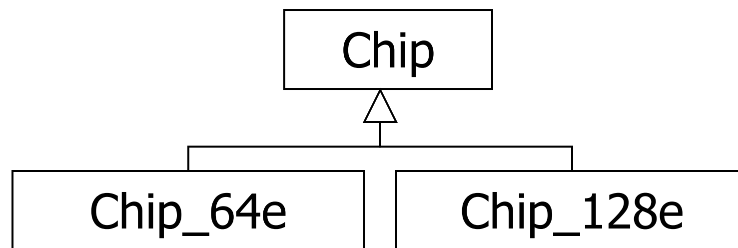
We strive for 1→1 or 1→many relationships between abstractions (abstract classes or interfaces) as opposed to n→m relationships between implementations (subclasses or interface implementations)

Favor Delegation Over Inheritance

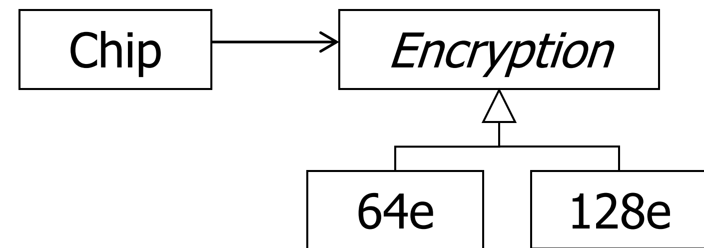
- Specializing function with inheritance is a short path to problems
- As we saw, once we hit variation two, we get multiple problems
- However, even in the patterns, inheritance is used extensively
- So, what does this really mean?

The Proper Use of Inheritance

- We can define a class that encapsulates variation, contain (via delegation) an instance of a concrete class derived from the abstract class defined earlier



Class Inheritance to Specialize



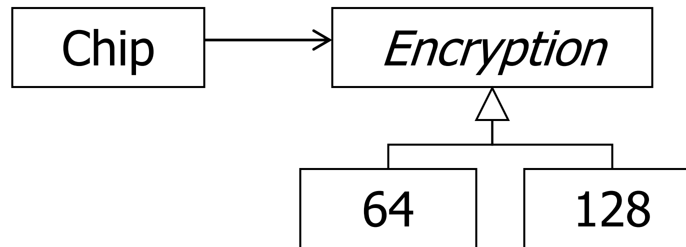
Class Inheritance to Categorize

Who is the variation being hidden from?

1. Allows for decoupling of concepts
2. Allows for deferring decisions until runtime
3. Small performance hit

Find What Varies and Encapsulate It

- Seems like what we just did…



  - Base classes encapsulate their implementing subclasses
  - This encapsulates varying *behavior*

Find What Varies and Encapsulate It

- The GoF meant a varying *anything*:
  - Varying design
  - Varying object creation
  - Varying relationships (1-1, 1-many)
  - Varying sequences and workflows
  - Etc…

Encapsulate Variations

- Encapsulating variation means to make it ***appear as if the varying issue is not varying***

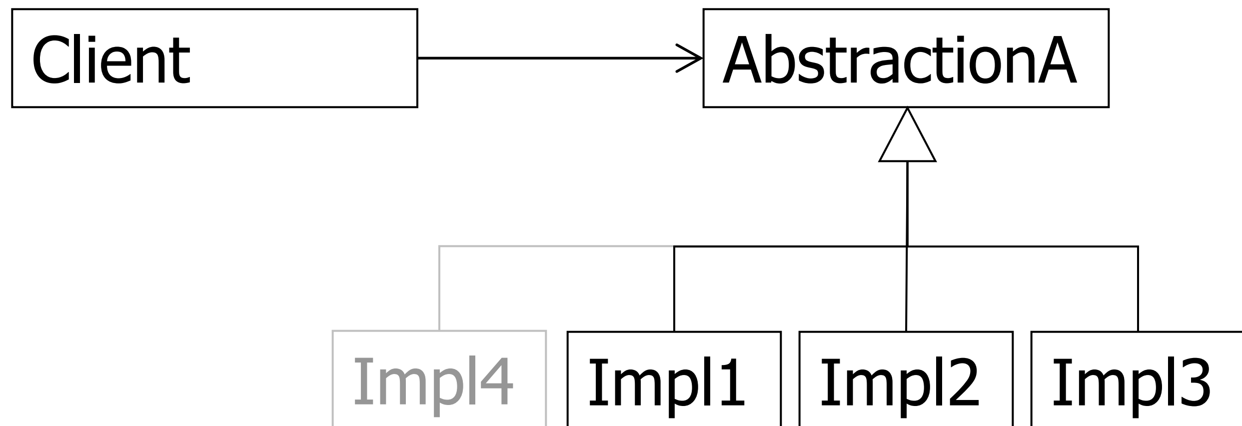
- Each pattern encapsulates a different varying thing

Encapsulate Variations

- Encapsulating variation means to make it ***appear as if the varying issue is not varying***
- Each pattern encapsulates a different varying thing

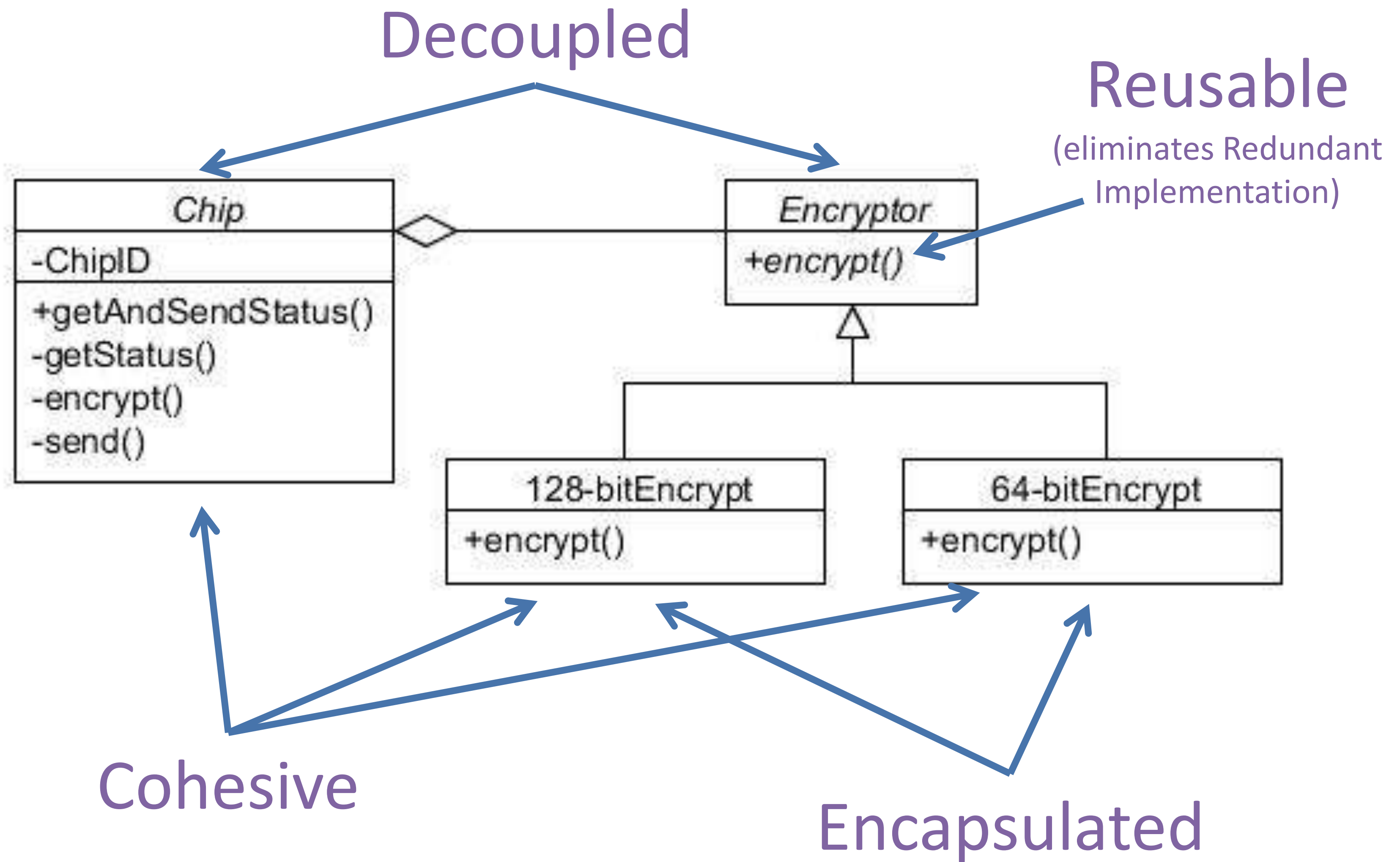
This Advice Relates to Principles

- Deal with things at an abstract level
- Why?



Relates to: The Open-Closed Principle

This Advice Promotes Quality



This Advice Promotes Quality

- Design to Interfaces:
  - Helps eliminate redundant relationships
  - Avoids subclass coupling
- Encapsulate Variation
  - Promotes encapsulation
  - Decouples client objects from the services they use
  - Leads to component-based architectures
- Favor aggregation over inheritance
  - Promotes strong cohesion
  - Helps eliminate redundant implementation

Design Patterns Are *Examples*

- Each pattern is an example of a design that follows this general advice well, *in a given context*

- Design patterns are **discovered**, not **invented**: they are what successful design have done to solve the same problem

- Studying design patterns is a good way to study good design, and how it plays out under various circumstances

The Thought Process of Patterns

1. What are design patterns?
2. **Programming by Intention**
3. Separating Use from Construction
4. Define tests up front
5. Shalloway's Law
6. Encapsulate that!

- Pretend what you need exists
- Use it
- Then build it

thinking points

Programming by Intention

C++

"Sergeant" Method

```
void Report::printReport (string CustomerID) {
  vector<Employee> emps = getEmployees(CustomerID);
  if(needsSorting(emps)) sortEmployees(emps);
  printHeader(CustomerID);
  printFormattedEmployees(emps);
  printFooter(CustomerID);
  paginate();
}
```

Private
Methods

*Note: These methods may not be *literally* private, as we may need to make some of them public or protected for testing. But we *treat them as private* from client objects, to limit coupling.

The Thought Process of Patterns

1. What are design patterns?
2. Programming by Intention
3. **Separating Use from Construction**
4. Define tests up front
5. Shalloway's Law
6. Encapsulate that!

- If you don't know what you are using you can use something else
- If you hide construction you must hide deletion

thinking points

The Use of Factories

- Factories promote the encapsulation of design

- This enables one mandate in design:
  - 1st determine what your entities are and how they work together
  - 2nd decide how to instantiate the right objects

- The only question is, when do you use factories?
  - Always? Seems like overkill
  - But if you don't start with them, putting them in may cause extra maintenance

We Need a Practice

- A Practice is something we can do all the time
- Factories and separate interfaces cannot be practices
  - They are used when they are justified
- We can’t predict whether something is likely to vary
- We need something else

Motivations and concepts

Manage objects separately from their use

Example Code:

C++

```
public class SignalProcessor {
  public:
    SignalProcessor ();
    byte[] process( byte[] signal);
  private:
    ByteFilter *myFilter;
}

SignalProcessor::SignalProcessor () {
    myFilter = new HiPassFilter();
}
byte[] SignalProcessor::process(byte[] signal) {
    // Do preparatory steps
    myFilter->filter(signal);
    // Do other steps
    return signal;
}
```

Mixed Perspectives: *C++*

```
public class SignalProcessor {
  public:
    SignalProcessor ();
    byte[] process( byte[] signal);
  private:
    ByteFilter *myFilter;
}

SignalProcessor::SignalProcessor () {
    myFilter = new HiPassFilter;          <-- Creation
}
byte[] SignalProcessor::process(byte[] signal) {
    // Do preparatory steps
    myFilter->filter(signal);             <-- Use
    // Do other steps
    return signal;
}
```

What you hide you can change

- When entities are coupled, then one cannot be *freely* changed without effecting the other

- The nature of the coupling determines the nature of the freedom

What you hide you can change

- If we want to control coupling, we need to limit perspectives, because
  - The Use Perspective implies one sort of coupling
  - The Creation Perspective implies different coupling

- This would *seem* to imply the use of factories to build all objects, but that’s only one way

Principle:
Separate Use From Construction

The relationship between any entity A and any other entity B in a system should be limited such that
*A **makes** B or*
*A **uses** B,*
*and **never both**.*

This is a "Principle" -

There are Many Ways To Accomplish it

- Using an actual factory pattern
- Object Serialization in one place, de-Serialization in another
- Object-Relational Data-Binding
- Dependency Injection Frameworks
- Etc...

How Can We Get All the Benefit Without Excessive Cost

- We cannot know when something is going to vary in the future
- This fear can lead to overdesign if we're not careful
- We need a practice, something we can always do, even when we don't have enough motivation to use an actual factory, interface, etc…
- We can ***encapsulate the constructor**** in simple classes

*The original idea for this came from ***Effective Java*** by Joshua Bloch

Encapsulating the Constructor

C++

```
class ByteFilter {
  public:
    static ByteFilter* getInstance();
  protected:
    ByteFilter();
    virtual ~ByteFilter();
}
ByteFilter::ByteFilter () {
  // do any constructor behavior here
}
ByteFilter* ByteFilter::getInstance() {
     return new ByteFilter();
   }
Void ByteFilter::deleteInstance( ByteFilter *bf2Delete) {
    delete bf2Delete;
}
  // the rest of the class follows
}

void main {
  ByteFilter *myByteFilter;
    // . . .
    myByteFilter = ByteFilter::getInstance();
    // . . .
}
```

Accommodating Change: Complexity

C++

```
class ByteFilter { // this is an abstract class.
  public:
    static ByteFilter* getInstance();
  protected:
    ByteFilter();
};
```

```
ByteFilter* ByteFilter::getInstance() {
  if (someDecisionLogic()) {
    return new HiPassFilter();
  } else {
    return new LoPassFilter();
} }
//Note: both HiPassFilter and LoPassFilter derive from ByteFilter
//      but implement the filtering differently
```

```
void main () {
  ByteFilter *myByteFilter;
    // . . .
    myByteFilter = ByteFilter::getInstance();
    // . . .
}
```

No Change!

In C++, Objects on the Stack

C++

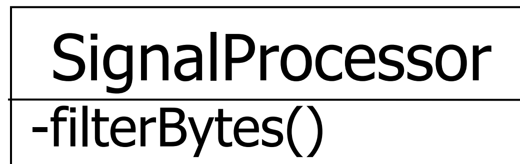
Instead of this…

```
ByteFilter myByteFilter;
//          .
//          . Object gets used
//          .
// Object falls out of scope, memory is cleaned up
```

…we prefer this – “Encapsulating Destruction”

```
ByteFilter* myByteFilter = ByteFilter::getInstance();
//          .
//          . Object gets used
//          .
ByteFilter::returnInstance(myByteFilter);
```

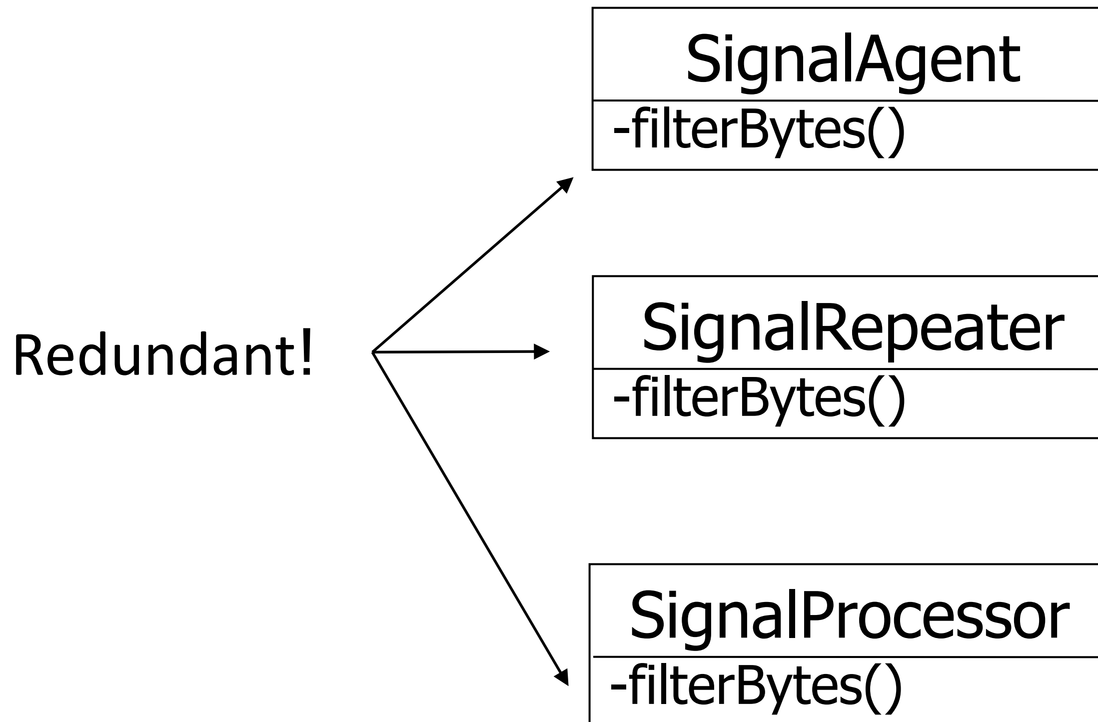
Evolution in Systems: 1



Single Client object
Non-Varying Service algorithm

Programming by Intention at least puts it in it's own method: method cohesion

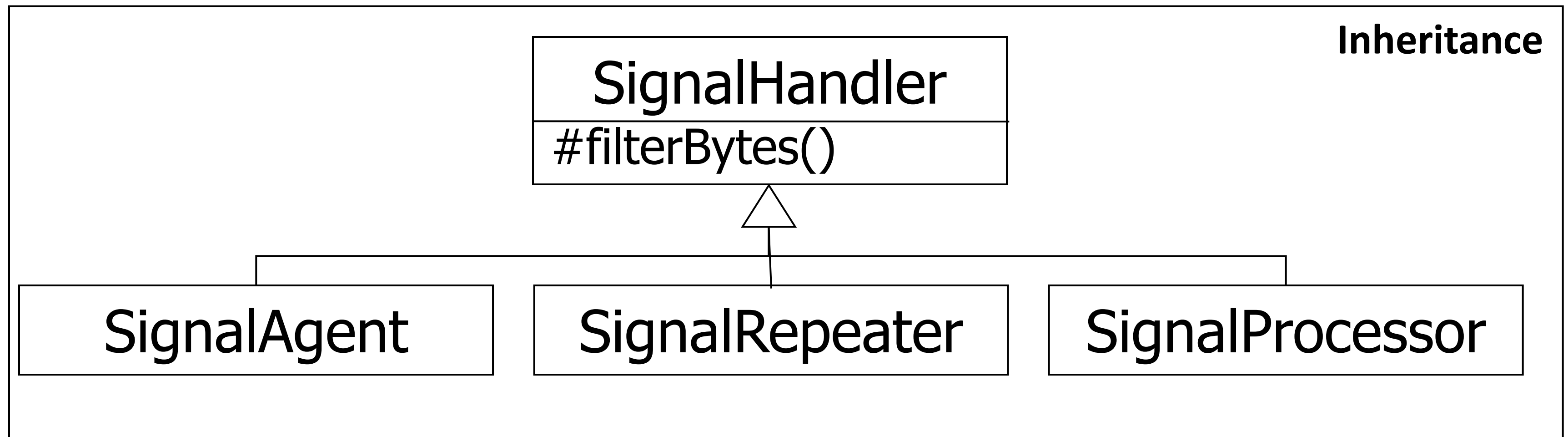
Evolution in Systems: 2

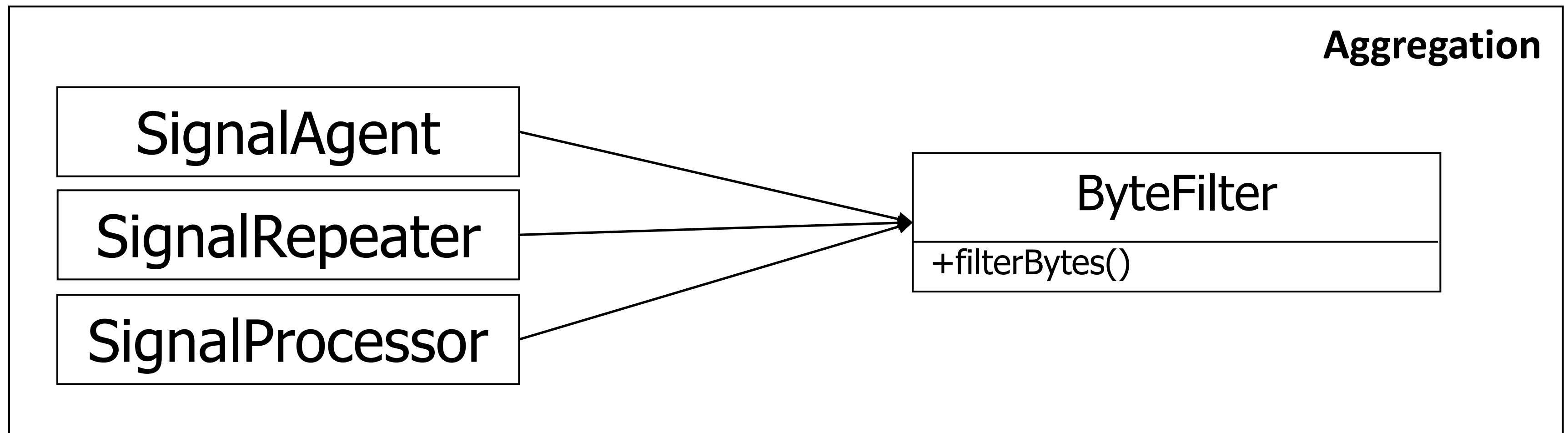


Eliminate Redundancy…

Evolution in Systems: 2

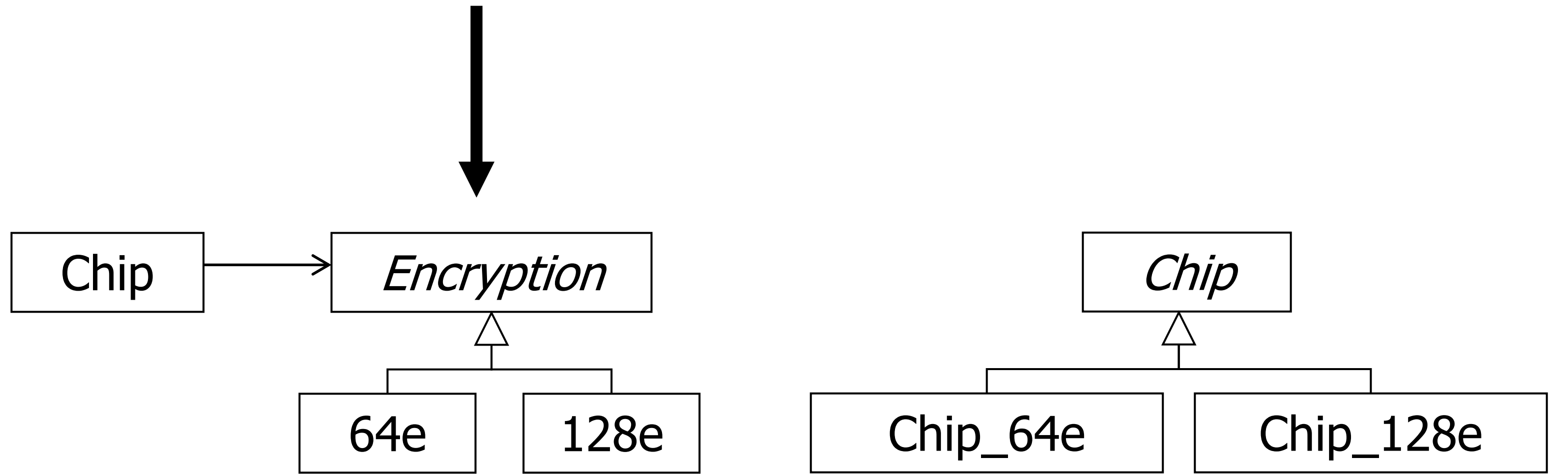
Choices…





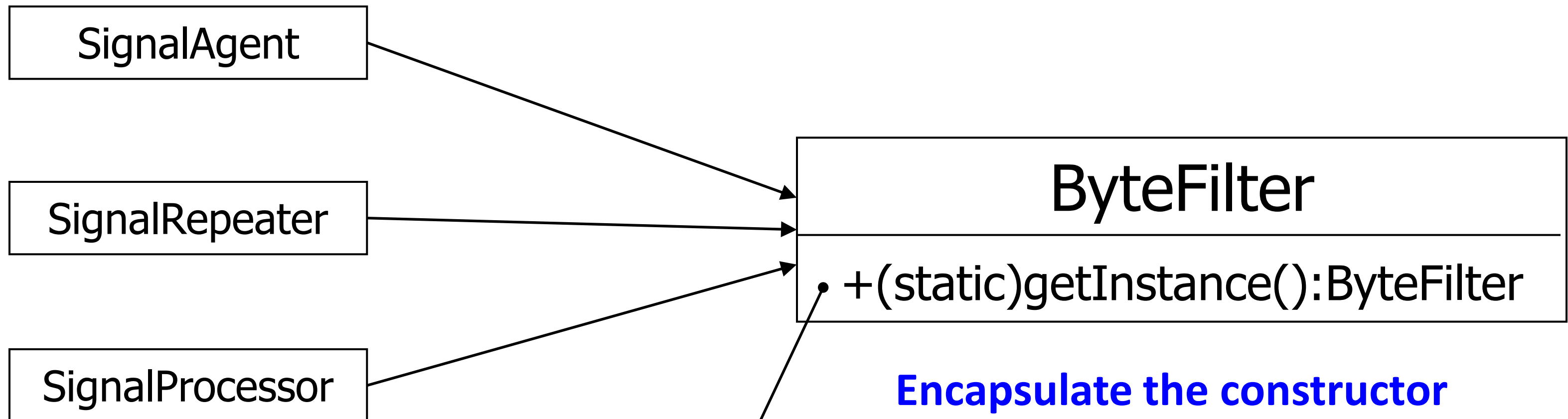
The Gang of Four said:

"Favor Aggregation Over Inheritance"



Evolution in Systems: 2

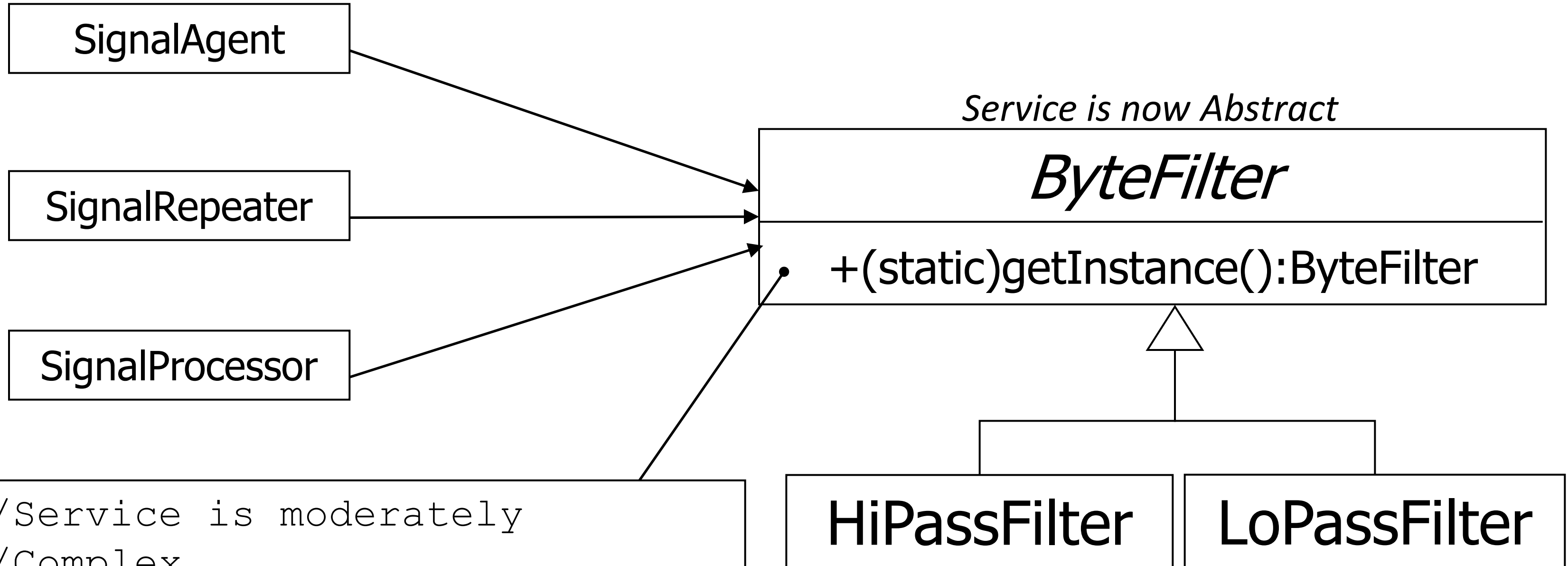
Move Method **Refactoring** to eliminate redundancy when multiple Clients require the same service…




```
//Service is simple
return new Bytefilter();
```

Evolution in Systems: 3

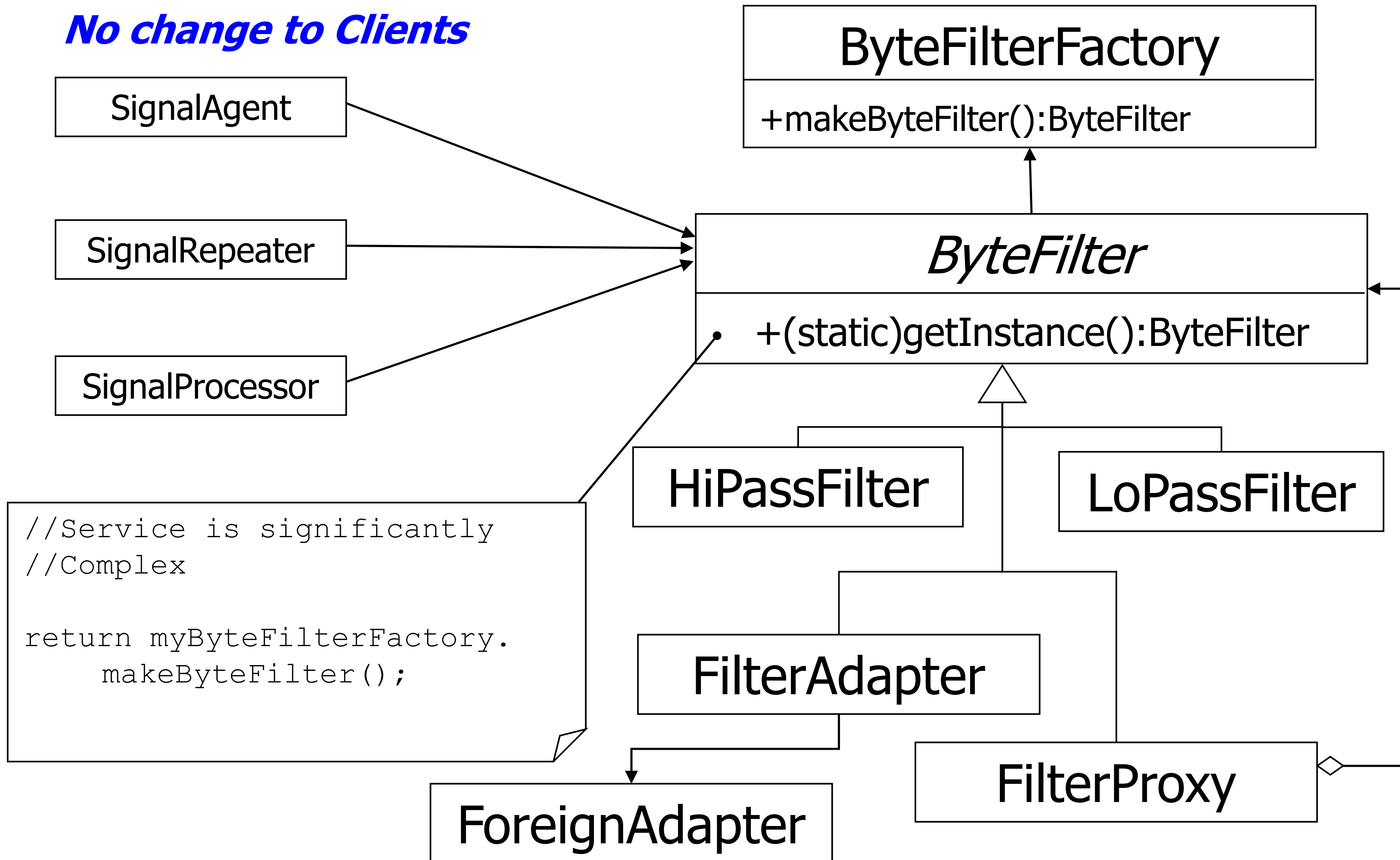
No change to Clients



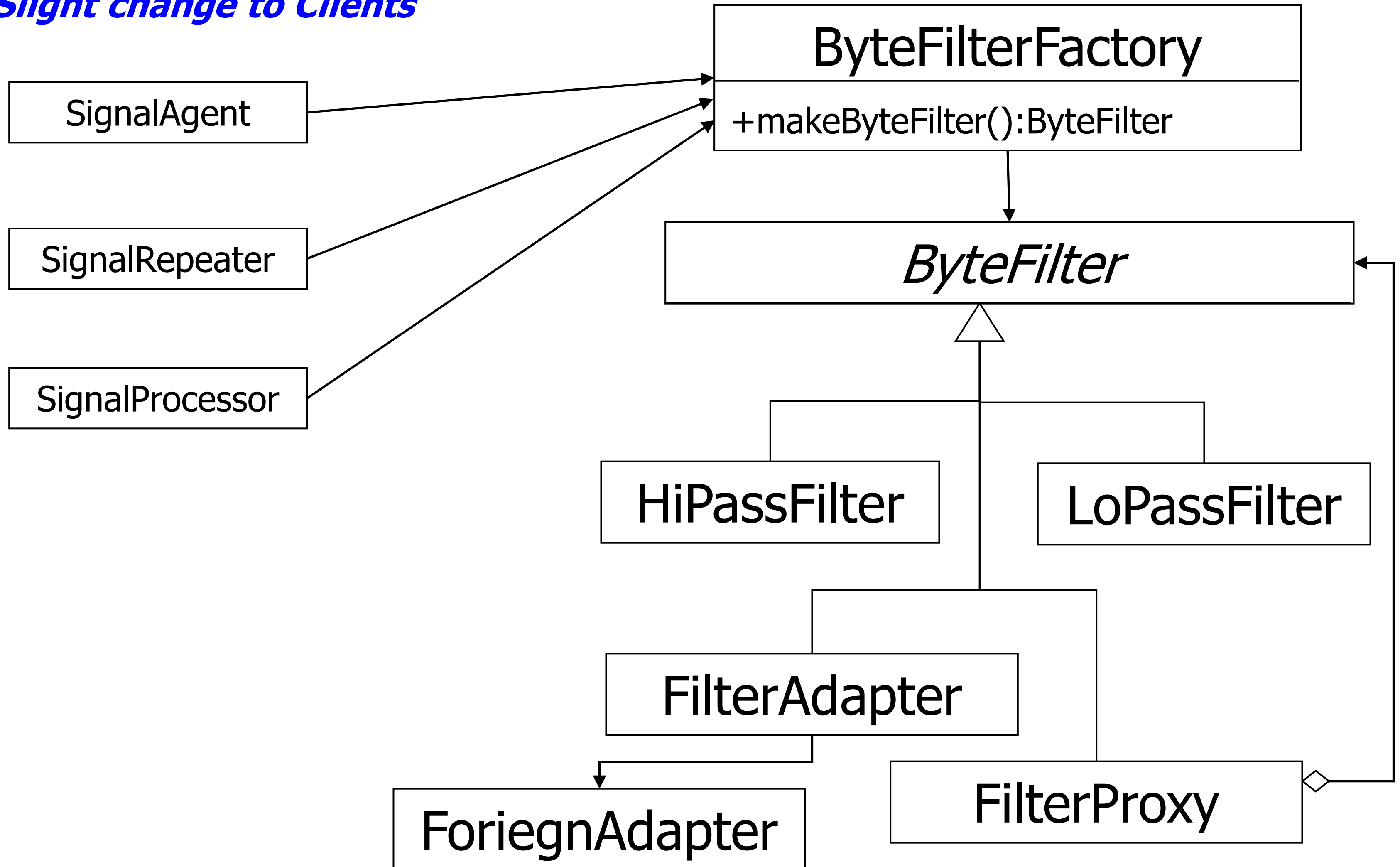

```
//Service is moderately
//Complex
if (decisionLogic()) {
  return new HighPassFilter();
} else {
  return new LoPassFilter();
}
```

The ***Strategy*** Pattern

Evolution in Systems: 4



An Optional Step



How do Factories Make Decisions?

- Based on a rule:
  - We filter one way during the day, another after hours
  - We call this "Rule Based“
- Based on state external to this subsystem:
  - There is a GUI or configuration file or environment variable that holds the information needed to make the decision
  - We Call this "Extrinsic”
- The Client has the information needed to make the decision
  - We call this "Intrinsic”

In Each Case

- **Rule Based**: The factory or encapsulating method keeps the rule in one place. This is a good thing
- **Extrinsic**: Only the factory on encapsulating method couples to the GUI, config file, or whatever. This limits coupling, and is a good thing
- **Intrinsic**: The client will have to pass in a parameter, which does mean a bit of maintenance, however…

Encapsulation of Construction

- Essentially no programming cost
- Allows for accommodating future, unforeseen variation without excessive anticipation
- Promotes the Open-Closed Principle
- Promotes cohesion of Perspective:
  - Fowler's Conceptual, Specification, Implementation
  - Our Use, Construction

The Thought Process of Patterns

1. What are design patterns?
2. Programming by Intention
3. Separating Use from Construction
4. **Define tests up front**
5. Shalloway's Law
6. Encapsulate that!

- Test specifications are about behavior
- Defining tests up front is a kind of analysis
- Defining tests up front is a kind of design

thinking points

Testability

- Code that is difficult to unit test is often:
  - Tightly Coupled: "I cannot test this without instantiating half the system"
  - Weakly Cohesive: "This class does so many things, the test will be enormous and complex!"
  - Redundant: "I'll have to test this in multiple places to ensure it works everywhere"

The Role of Testability

- Unit testing is a good thing. You should do it
- Whether you agree or not
  - You should always ask yourself "if I were to test this, how would I do it?"
  - If you find the design would be very hard to test, or if you cannot see a way to test it, ask yourself "why isn't this more testable?"
- This is a significant Trim Tab

The Thought Process of Patterns

1. What are design patterns?
2. Programming by Intention
3. Separating Use from Construction
4. Define tests up front
5. **Shalloway's Law**
6. Encapsulate that!

- No redundancy is a good idea
- No redundancy is also impossible
- How can we manage redundancy?

thinking points

Shalloway's Law

If 'N' things need to change and 'N>1', Shalloway will find at most 'N-1' of these things

Shalloway's Principle

Avoid situations where
Shalloway's Law Applies

The Thought Process of Patterns

1. What are design patterns?
2. Programming by Intention
3. Separating Use from Construction
4. Define tests up front
5. Shalloway's Law
6. **Encapsulate that!**

- What do we do when we don't know what to do?
- Scott Bain's Magic Card

thinking points

Our Situation

- We need a method of real-time messaging
- We know we'll have a performance problem
- We don't know where it will be

- What can we do?

Our Options

1. Figure out, up-front, what'll work

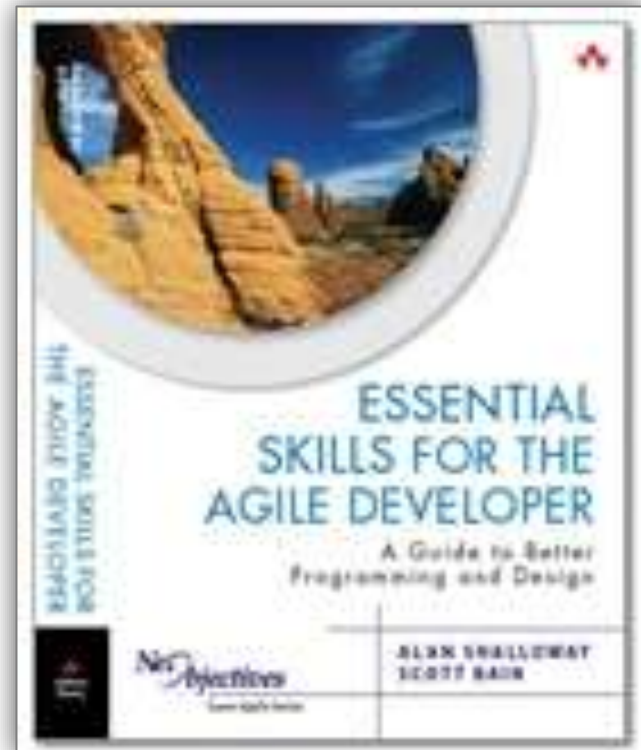
2. Just use what appears best at first and fix it later if it becomes clear later that we need something else

Downsides to First Two Options

- Figuring out up-front is fraught with risk
- Just use an easy one and fix it later has different risk
- How would I design this if I knew that no matter how I designed it I would later discover I did it the wrong way?

Learn Trim Tabs

1. Programming by Intention
2. Separating Use from Construction
3. Define tests up front
4. Shalloway's Law
5. Encapsulate that!



Thank You!

NetObjectives

ASSESSMENTS
CONSULTING
TRAINING
COACHING

Register at www.netobjectives.com/register to access many webinars

Contact me at alshall@netobjectives.com @alshalloway

Webinar: Lean-Agile: The Next Generation May 22

Webinar: Product Portfolio Management: Why It Is Critical for Agile at Scale June 18

Design Patterns for Agile Developers, Seattle, June 19-21

Lean-Agile Project Management, Seattle, June 19-21

Sustainable Test-Driven Development, Seattle, July 10-12