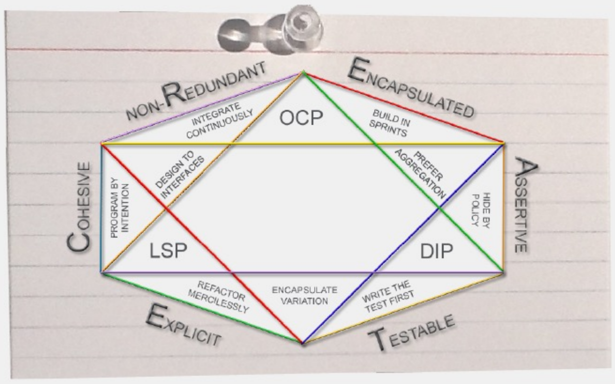
to be agile

Patterns of Instantiation



Object Lifecycle Management

http://ToBeAgile.com
info@ToBeAgile.com

DB20190710

1

David Scott Bernstein

- Software developer since 1980
- Trained 8,000 developers since 1990
- Author of the book *Beyond Legacy Code: Nine Practices to Extend the Life (and Value) of Your Software*

Website: http://ToBeAgile.com

Twitter: @ToBeAgile

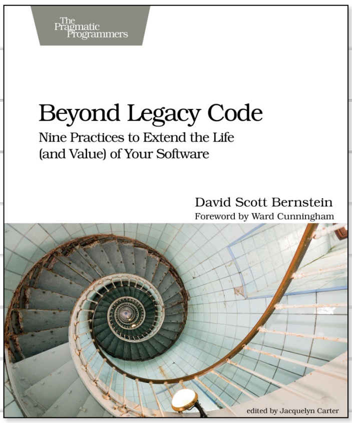
to be agile

2

2

My Book – Beyond Legacy Code

to be agile
lessen the learning curve
http://ToBeAgile.com
info@ToBeAgile.com


The Pragmatic Programmers
Beyond Legacy Code
Nine Practices to Extend the Life (and Value) of Your Software
David Scott Bernstein
Foreword by Ward Cunningham
edited by Jacquelyn Carter

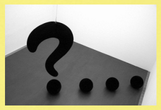
- Nine practices to design and build healthy code, plus some tips on dealing with legacy code.
- Discusses the value and reasoning behind the technical practices, so both managers and the team can get on the same page as to their value.
- It's not a "How To" book, it's a "Why To" book.

http://BeyondLegacyCode.com

to be agile 3

3

Why this Talk?



- Instantiation is at the very core of object-oriented programming but often misunderstood and under-utilized.
- Failing to leverage instantiation in object-oriented programming creates tightly coupled classes that are difficult to extend.
- This is the biggest technical issue I find in virtually all the code I see from my clients, who are the largest companies in the world.
- When we leverage object instantiation we build software that is straightforward to extend and verify, dropping the cost of ownership.
- In the 1990s, I taught nearly 4,000 professional software developers wrong, everyone did, and I want to make up for that now as best I can.

to be agile 4

4

Patterns and Anti-Patterns

- Design patterns is a term coined by Christopher Alexander who used it to describe the forces that make a structure “livable.”
- Design patterns were adopted by software developers to describe common intents or way of encapsulating something that is varying.
- We commonly think of patterns as “best practices.”
- If patterns are “best practices” then anti-patterns are “worst practices.”
- In this session, we’ll look at some common anti-patterns, why they should be avoided, and what good patterns can be used instead.

Anti-Pattern: Creating Objects You Use

- Good Intention: Create an object so you can use its services.
- Flaw: Over-encapsulates services that an object uses.
- Result: From the outside, the created object becomes indistinguishable from the object that creates it, making it impossible to independently verify, extend, or reuse.
- Testability: Objects that create the services they use are inseparable from those services so they must be tested together, which can make tests slow and unreliable.
- Contraindications: This only applies to external dependencies or objects you might want to extend in the future.

Why It’s Bad to Create Objects You Use

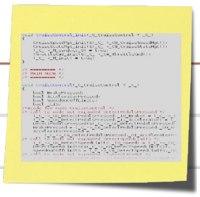
- When one object instantiates another object and then uses it there’s no way to substitute the object it’s using.
- This creates a dependency between the two pieces of code that makes it impossible to test each piece separately.
- It also means that we can’t extend one without changing the other.
- Following this anti-pattern causes a system to become brittle, intertwined, and nearly impossible to work with.

to be agile

7

7

For Example



- A common programming practice is to new up the services you need in an object’s construction. For example:

```
public class MyClass {
      Service myService;
      public MyClass() {
            Service myService = new Service();
       }
      public void doSomething() {
            /* ... */
            myService.process();
            /* ... */
      }
}
```

Instantiates

Uses

to be agile

8

8

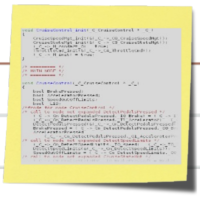
Problems with New

- The “new” keyword is used to create an instance of a class
- It requires that you pass in the class name
- It returns an instance of the class
- Therefore the caller of “new” must know the class it wants to create

to be agile 9

9

Mixing Perspectives

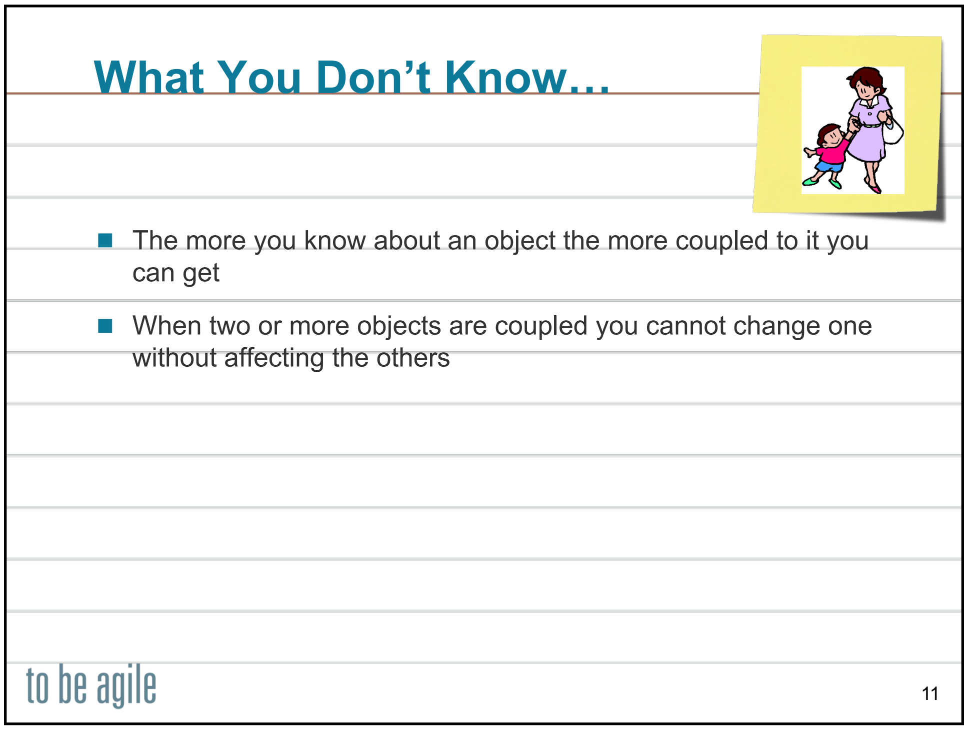


Here’s some code that I would have written 17 years ago

```
public class Document {
  Sort sortStrategy;
  public Document() {
    Sort sortStrategy = new Sort(); ← Creates
    }
   public void prepareDocument() {
   /* ... */
   sortStrategy.sort(); ← Uses
   /* ... */ }
}
```

to be agile 10

10



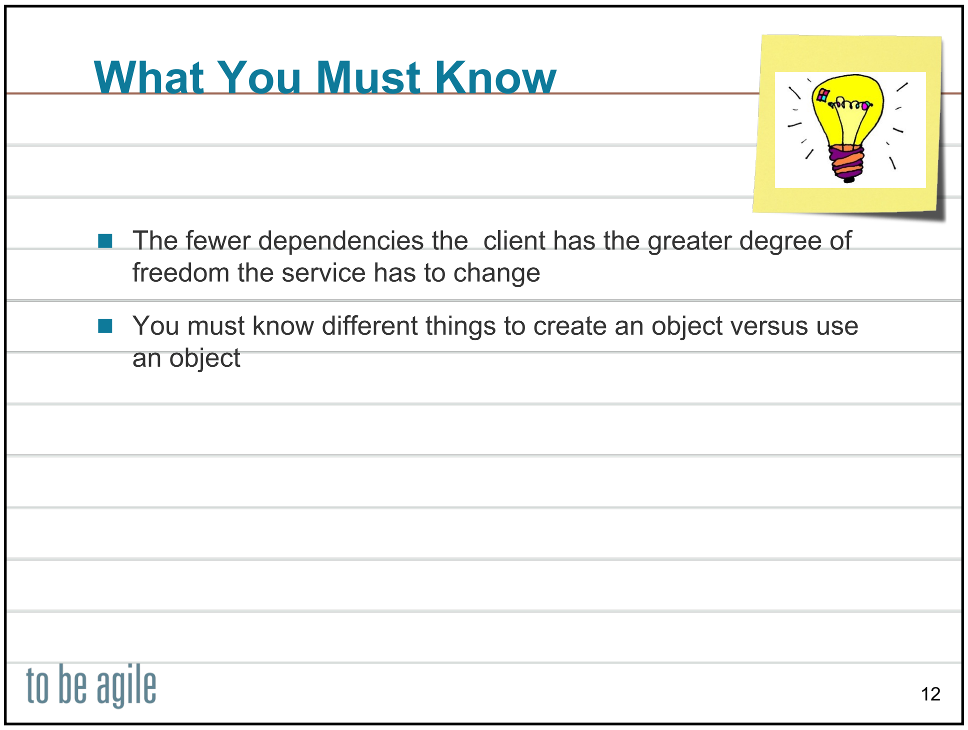
What You Don't Know…

- The more you know about an object the more coupled to it you can get
- When two or more objects are coupled you cannot change one without affecting the others

to be agile

11

11



What You Must Know

- The fewer dependencies the client has the greater degree of freedom the service has to change
- You must know different things to create an object versus use an object

to be agile

12

12

To Create an Object

- To instantiate an object you must know:
  - The object's type
  - Any overloaded constructors

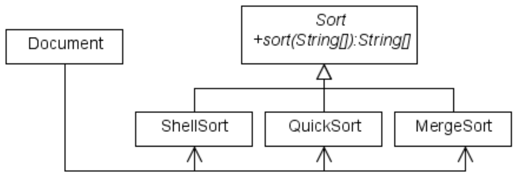
to be agile

13

13

Creating Example

- What can you change without affecting the caller?
  - You can change the method signature
- What can you not change without affecting the caller?
  - You cannot change the specific derivations



to be agile

14

14

To Use an Object

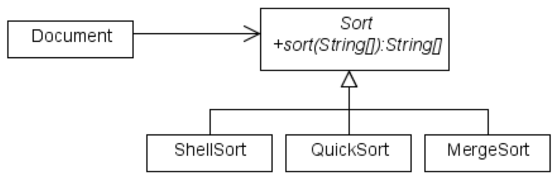
- To call methods on an object you must know:
  - The object’s type, or
  - The type the object is derived from, or
  - An interface the object implements
- When you call a method you are also coupled to its interface

to be agile 15

15

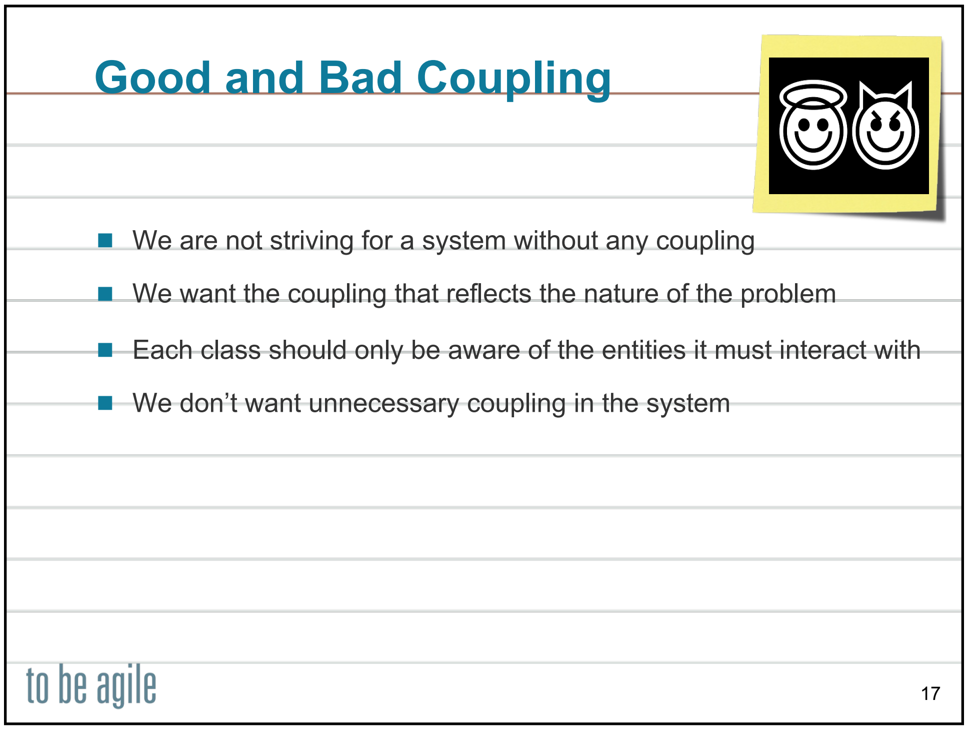
Using Example

- What can you change without affecting the caller?
  - You can add new derivations
- What can you not change without affecting the caller?
  - You cannot change the method signature



to be agile 16

16

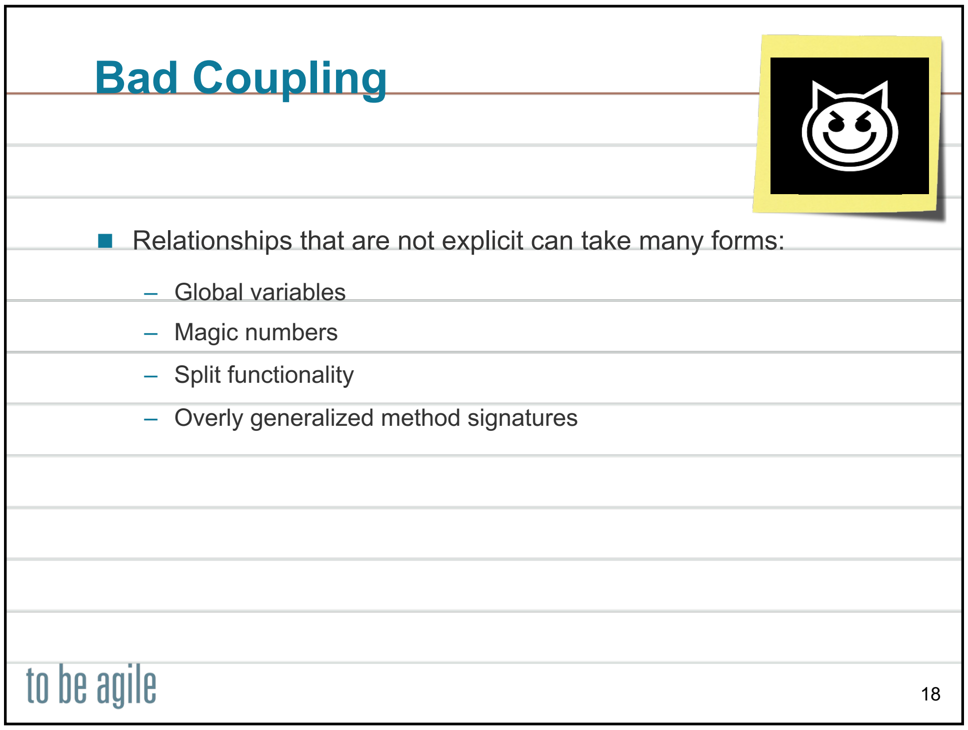


Good and Bad Coupling

- We are not striving for a system without any coupling
- We want the coupling that reflects the nature of the problem
- Each class should only be aware of the entities it must interact with
- We don't want unnecessary coupling in the system

17

17



Bad Coupling

- Relationships that are not explicit can take many forms:
  - Global variables
  - Magic numbers
  - Split functionality
  - Overly generalized method signatures

18

18

Kinds of Coupling

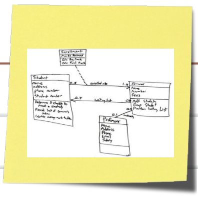
Name	Coupled To
Type coupling	The existence of a class
Interface coupling	The method signatures of another class
Abstract coupling	The abstract type only
Concrete coupling	A subtype in a polymorphic set

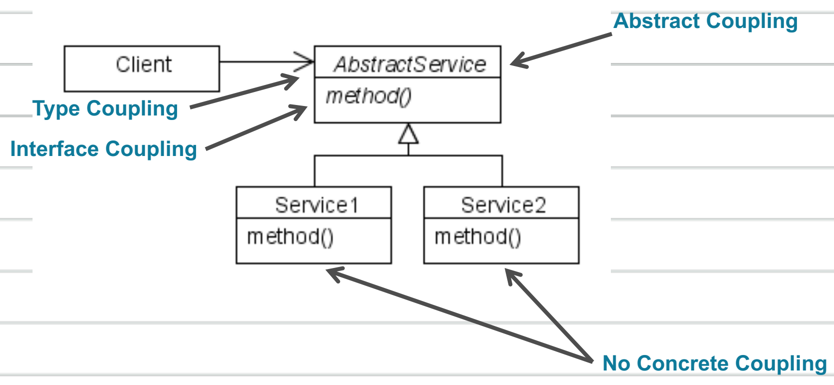
to be agile

19

19

Coupling Example



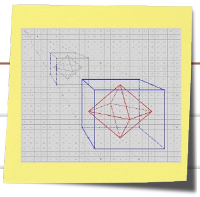


to be agile

20

20

Coupling of Perspectives



Coupling	Creation	Use
Type	Yes	No
Interface	No	Yes
Abstract	Yes	Yes
Concrete	Yes	No

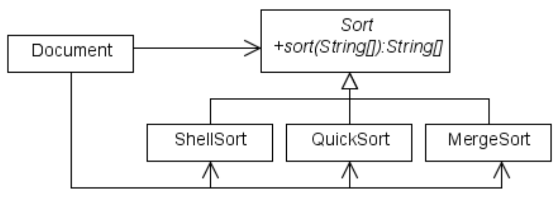
to be agile

21

21

Mixing Perspectives

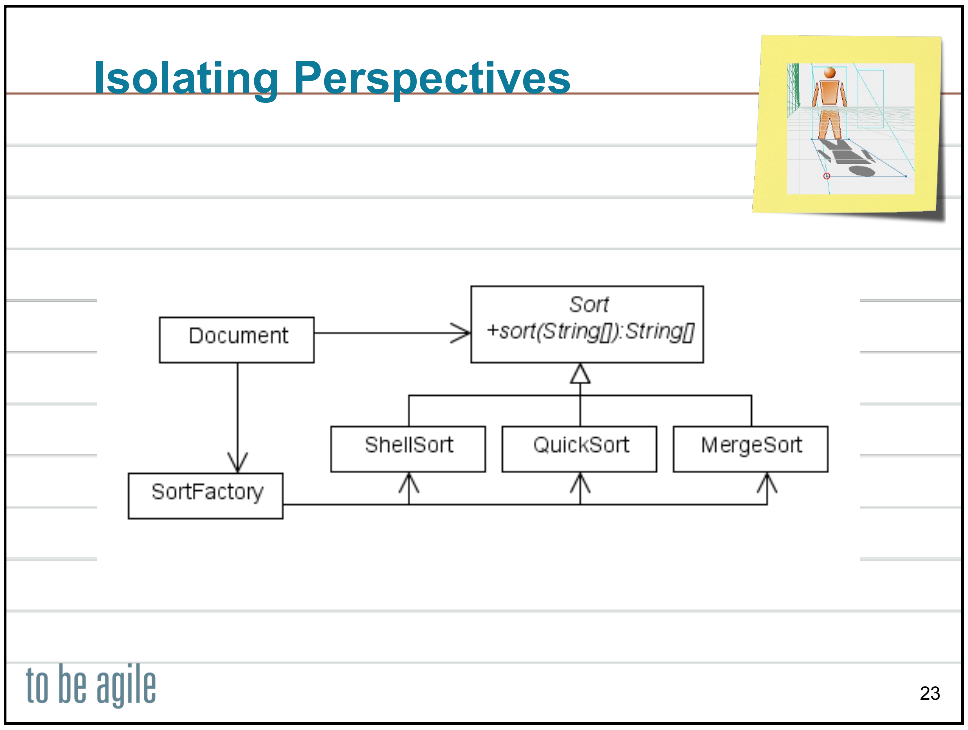
- When you mix the perspective of creation with the perspective of usage
  - What you can change freely is nothing
  - What you cannot change freely is everything



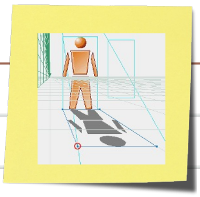
to be agile

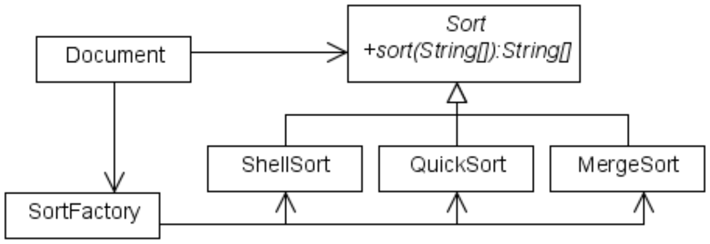
22

22



Isolating Perspectives

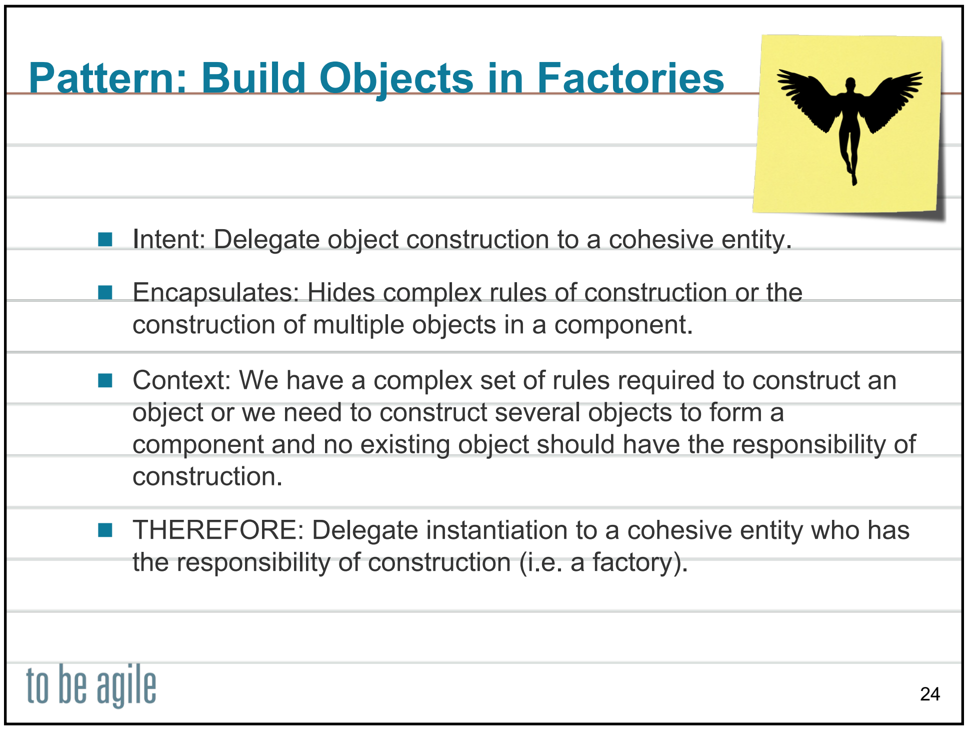




to be agile

23

23



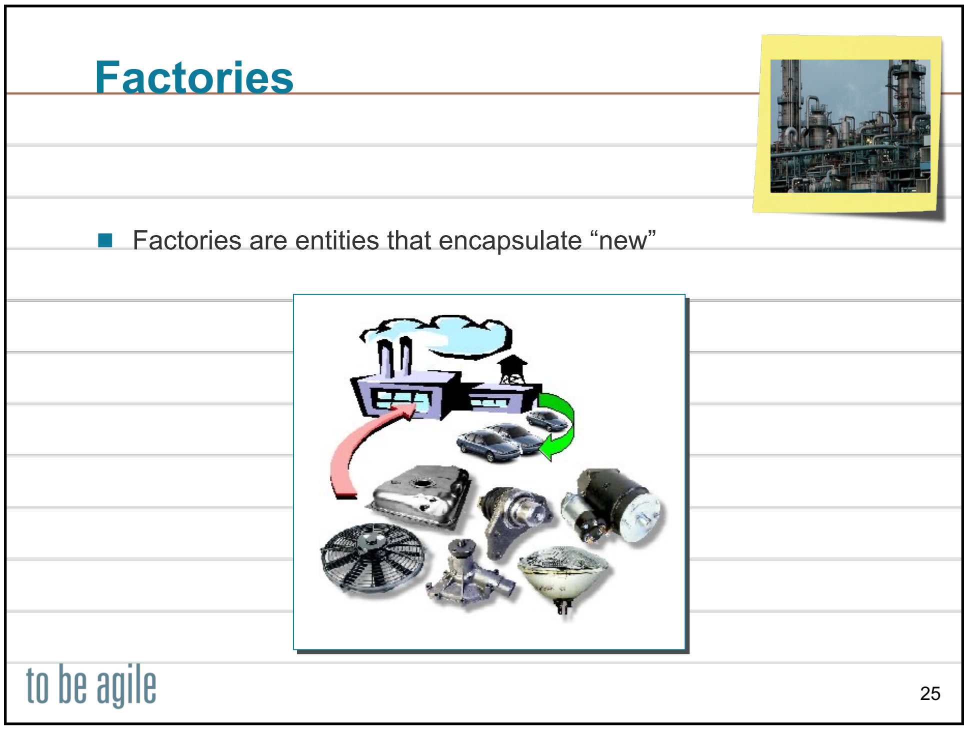
Pattern: Build Objects in Factories

- Intent: Delegate object construction to a cohesive entity.
- Encapsulates: Hides complex rules of construction or the construction of multiple objects in a component.
- Context: We have a complex set of rules required to construct an object or we need to construct several objects to form a component and no existing object should have the responsibility of construction.
- THEREFORE: Delegate instantiation to a cohesive entity who has the responsibility of construction (i.e. a factory).

to be agile

24

24



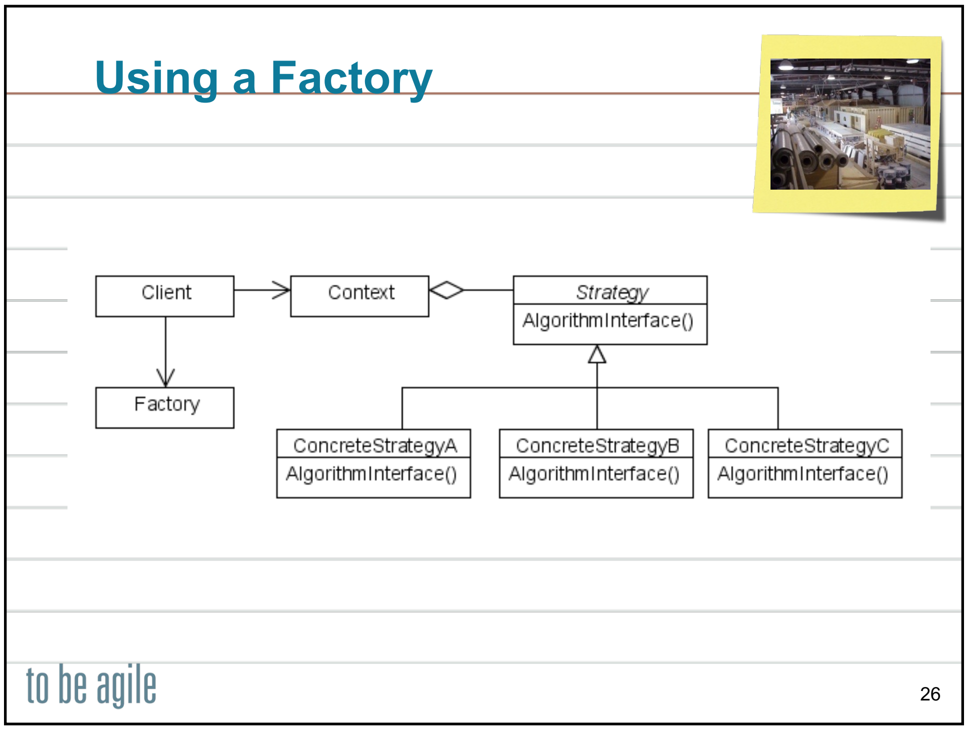
Factories

- Factories are entities that encapsulate "new"

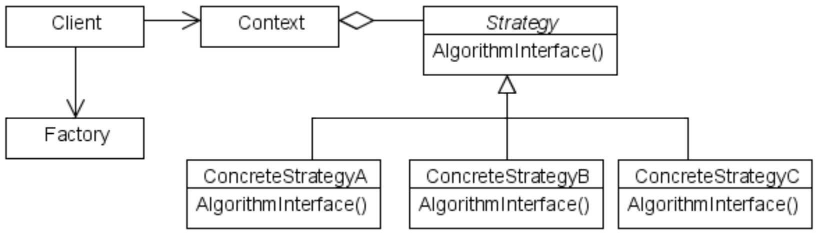
to be agile

25

25



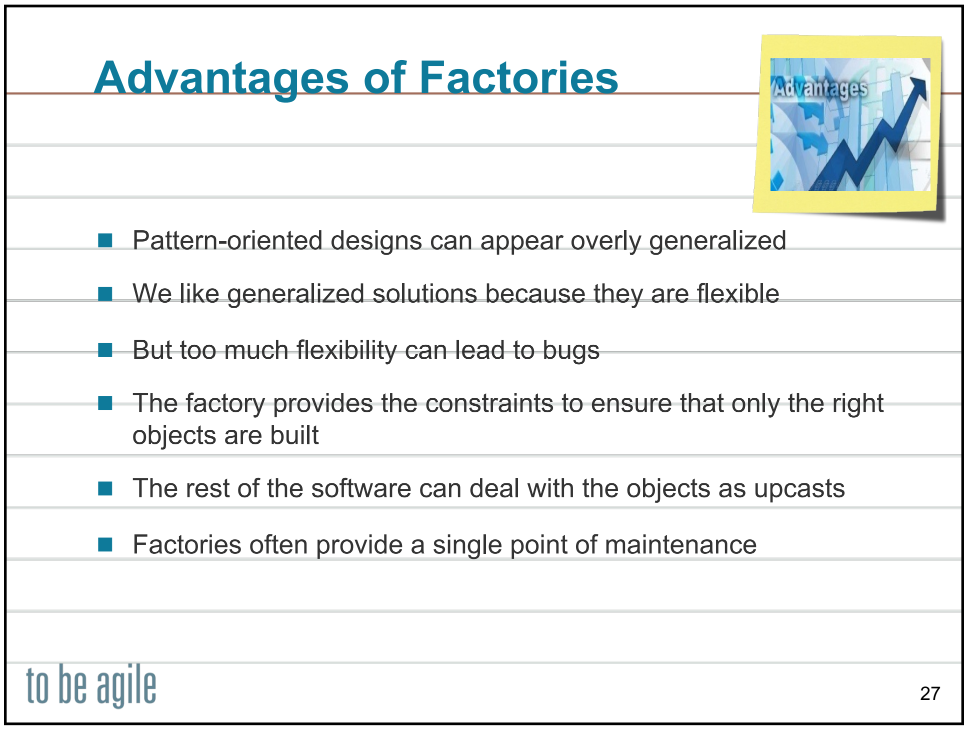
Using a Factory



to be agile

26

26

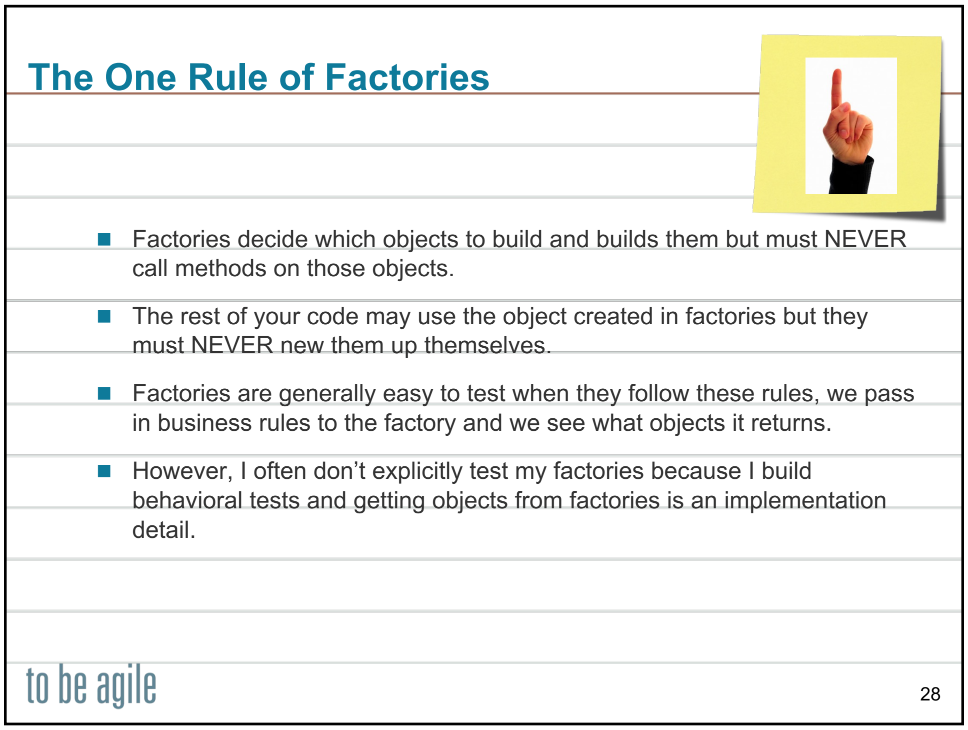


Advantages of Factories

- Pattern-oriented designs can appear overly generalized
- We like generalized solutions because they are flexible
- But too much flexibility can lead to bugs
- The factory provides the constraints to ensure that only the right objects are built
- The rest of the software can deal with the objects as upcasts
- Factories often provide a single point of maintenance

to be agile 27

27

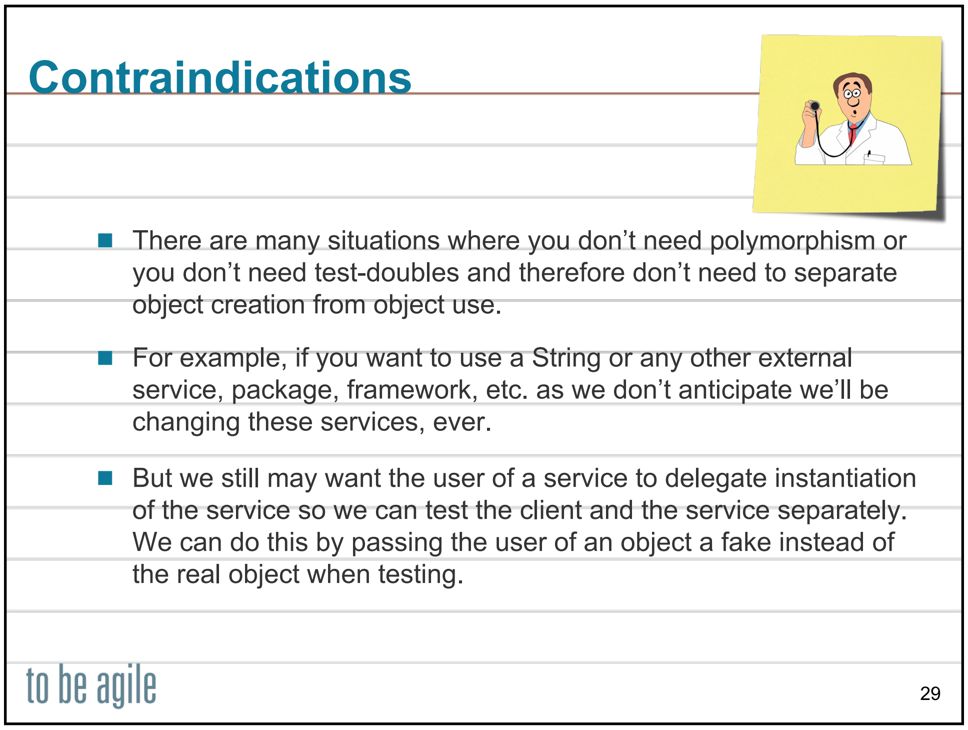


The One Rule of Factories

- Factories decide which objects to build and builds them but must NEVER call methods on those objects.
- The rest of your code may use the object created in factories but they must NEVER new them up themselves.
- Factories are generally easy to test when they follow these rules, we pass in business rules to the factory and we see what objects it returns.
- However, I often don’t explicitly test my factories because I build behavioral tests and getting objects from factories is an implementation detail.

to be agile 28

28



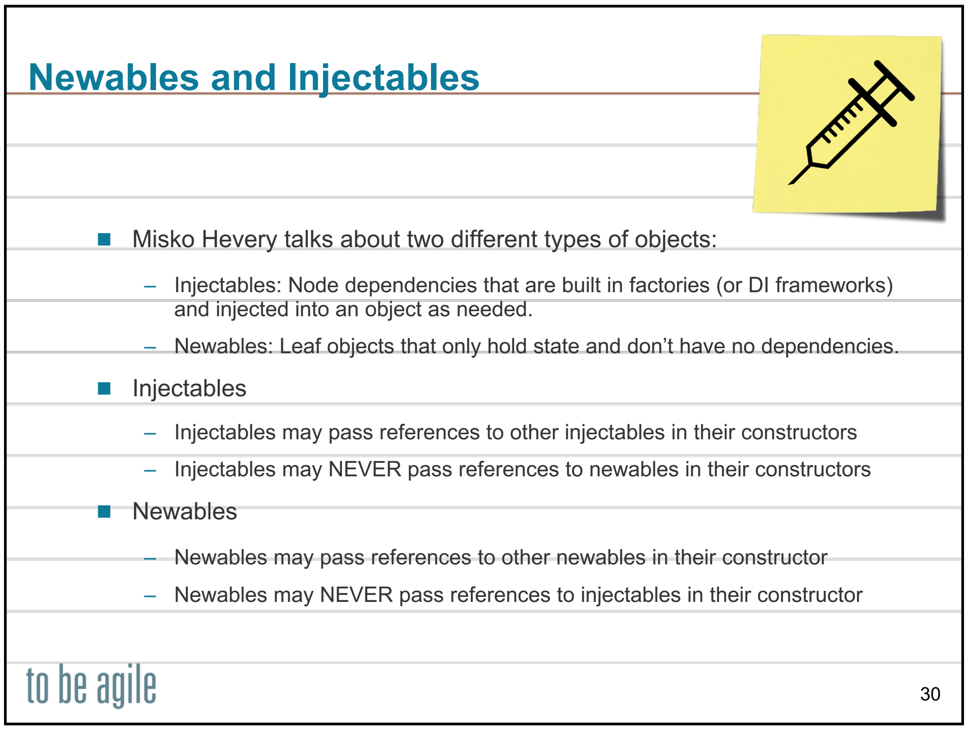
Contraindications

- There are many situations where you don’t need polymorphism or you don’t need test-doubles and therefore don’t need to separate object creation from object use.
- For example, if you want to use a String or any other external service, package, framework, etc. as we don’t anticipate we’ll be changing these services, ever.
- But we still may want the user of a service to delegate instantiation of the service so we can test the client and the service separately. We can do this by passing the user of an object a fake instead of the real object when testing.

to be agile

29

29



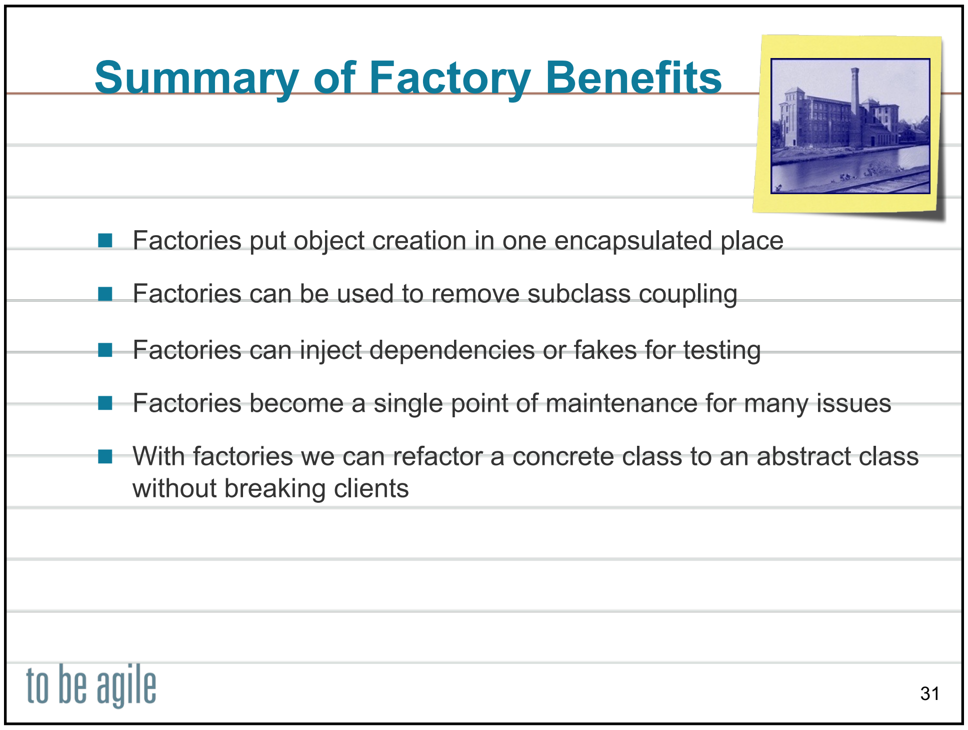
Newables and Injectables

- Misko Hevery talks about two different types of objects:
  - Injectables: Node dependencies that are built in factories (or DI frameworks) and injected into an object as needed.
  - Newables: Leaf objects that only hold state and don’t have no dependencies.
- Injectables
  - Injectables may pass references to other injectables in their constructors
  - Injectables may NEVER pass references to newables in their constructors
- Newables
  - Newables may pass references to other newables in their constructor
  - Newables may NEVER pass references to injectables in their constructor

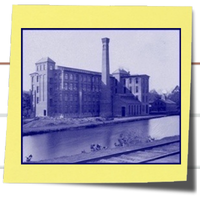
to be agile

30

30



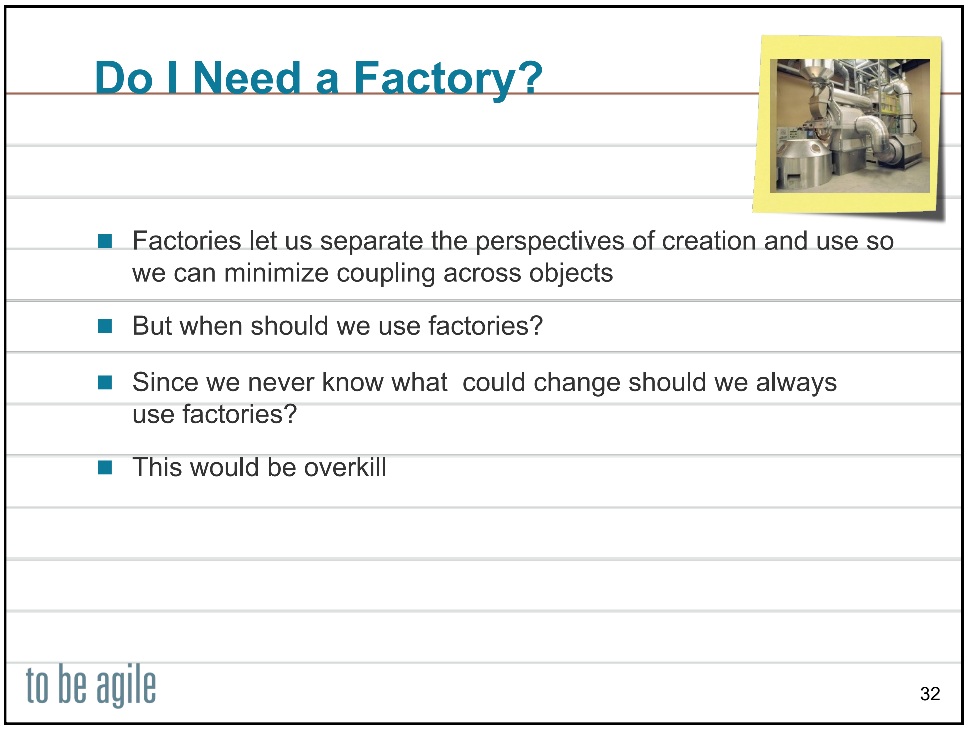
Summary of Factory Benefits



- Factories put object creation in one encapsulated place
- Factories can be used to remove subclass coupling
- Factories can inject dependencies or fakes for testing
- Factories become a single point of maintenance for many issues
- With factories we can refactor a concrete class to an abstract class without breaking clients

to be agile 31

31



Do I Need a Factory?

- Factories let us separate the perspectives of creation and use so we can minimize coupling across objects
- But when should we use factories?
- Since we never know what could change should we always use factories?
- This would be overkill

to be agile 32

32

The Question of Construction

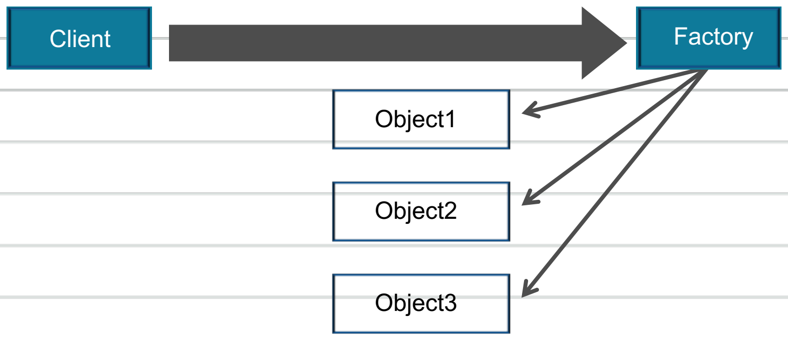
- When should we focus on constructing our objects?
- It is often easier to focus on object construction after you have come up with your basic design
- Building objects apart from where they are used will lead to higher code quality

to be agile

33

33

And Then a Miracle Happens



to be agile

34

34

Problems with Factories

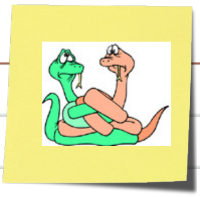
- But when should we use factories? Always?
- That seems like an awfully big burden

to be agile

35

35

Separate Construction from Use

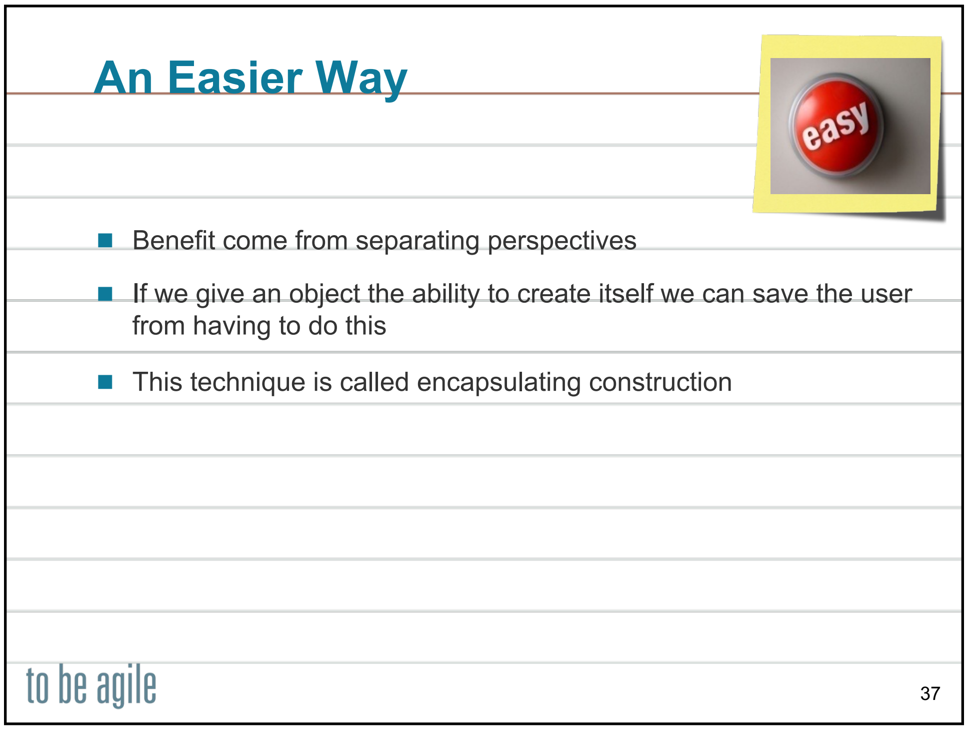


- Many of the benefits of using factories come from the separation of construction from use
- You must know different things to create an object versus to use it
- Separating out these perspectives means less unintentional coupling for the classes involved

to be agile

36

36



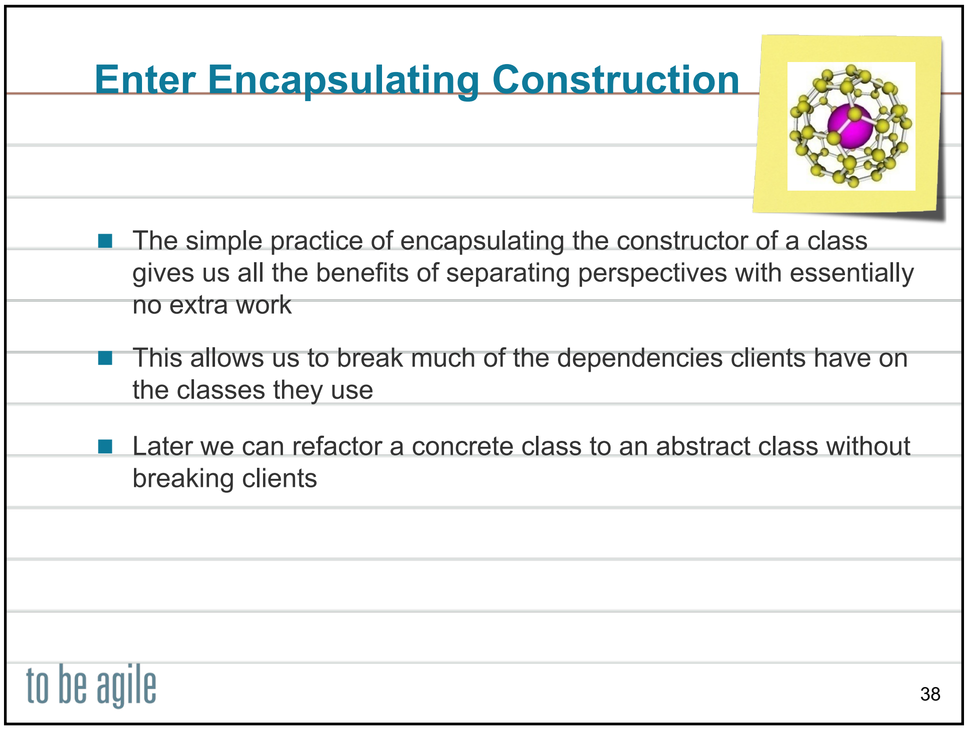
An Easier Way

- Benefit come from separating perspectives
- If we give an object the ability to create itself we can save the user from having to do this
- This technique is called encapsulating construction

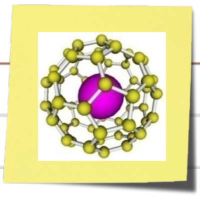
to be agile

37

37



Enter Encapsulating Construction



- The simple practice of encapsulating the constructor of a class gives us all the benefits of separating perspectives with essentially no extra work
- This allows us to break much of the dependencies clients have on the classes they use
- Later we can refactor a concrete class to an abstract class without breaking clients

to be agile

38

38

Pattern: Encapsulate Construction

- Intent: Give objects the responsibility of creating themselves
- Encapsulates: Hides the object's type from its users.
- Context: We would like users of objects to not have to create those objects themselves.
- THEREFORE: Objects can expose a public static method users can call so the object creates itself.

to be agile 39

39

Encapsulating Construction

```
public class Sort {
    private Sort() {
        // construction goes here
    }
    public static Sort getInstance() {
        return new Sort();
    }
    // ...
}
public class Document {
    private Sort mySort;
    public void processDocument() {
        // ...
        mySort = Sort.getInstance();
        mySort.sort();
        // ...
    }
}
```

to be agile 40

40

Refactoring to a Strategy

```
public abstract class Sort {
    private Sort() {
        // construction goes here
    }
    public static Sort getInstance() {
        if (someDecision() == true)) {
            return new ShellSort();
        } else {
            return new QuickSort();
        }
    }
    // ...
}
```

```
public class ShellSort extends Sort {
    // …
}
public class QuickSort extends Sort {
    // …
}
public class Document {
    private Sort mySort;
    public void processDocument() {
        // ...
        mySort = Sort.getInstance();
        mySort.sort();
        // ...
    }
}
```

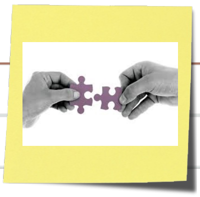
No change to client!

to be agile

41

41

Now Objects are Extensible



- Notice how when we encapsulate construction we can change a concrete class into an abstract class and introduce polymorphism without breaking our callers.
- Encapsulating construction allows us to inject design patterns, which are often based on abstract classes, virtually anywhere in code without breaking callers, allowing us a great deal of freedom to emerge designs.
- This one simple technique enables code to have maximum extensibility as well as independently verifiability.

to be agile

42

42

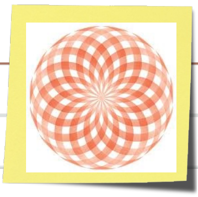
Why Encapsulate Construction

- When encapsulating construction we get many of the benefits of using a factory without the extra effort.
- The benefits of encapsulating construction include
  - Takes no extra time to provide
  - Lets us refactor a concrete class into an abstract class without affecting the caller
  - Promotes the Open-Closed Principle
  - Promotes a cohesion of perspectives by separating object creation from use

to be agile 43

43

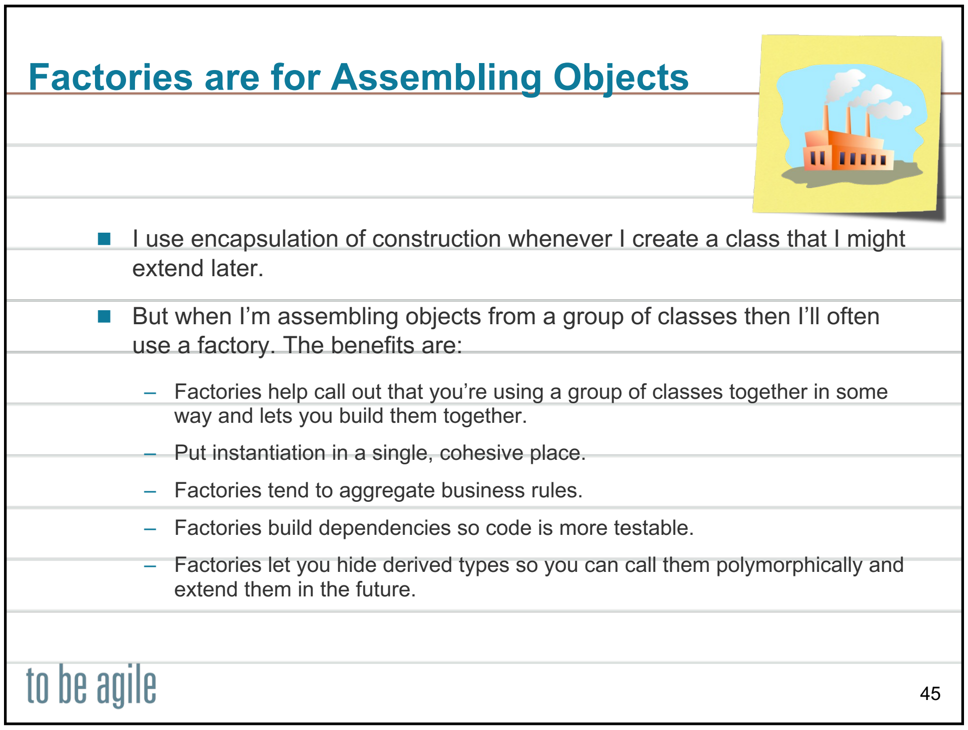
An Object's Responsibility



- The object-oriented programming model is based on created autonomous, assertive objects who are responsible for themselves.
- One of an object's most important responsibilities is to instantiate itself.
- This is true for biological organisms like bacteria and humans as well as solar systems and galaxies.
- If fact, we see many similar patterns in nature for instantiating biological processes that we see good coding practices, including abstract factory and builder patterns.

to be agile 44

44

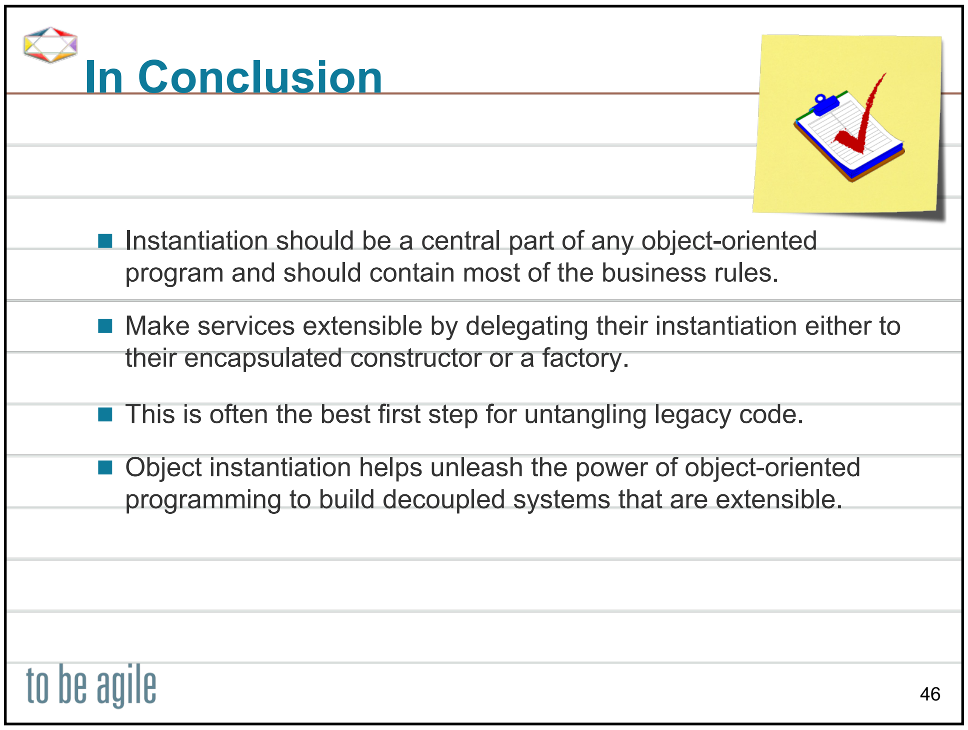


Factories are for Assembling Objects

- I use encapsulation of construction whenever I create a class that I might extend later.
- But when I'm assembling objects from a group of classes then I'll often use a factory. The benefits are:
  - Factories help call out that you're using a group of classes together in some way and lets you build them together.
  - Put instantiation in a single, cohesive place.
  - Factories tend to aggregate business rules.
  - Factories build dependencies so code is more testable.
  - Factories let you hide derived types so you can call them polymorphically and extend them in the future.

to be agile 45

45

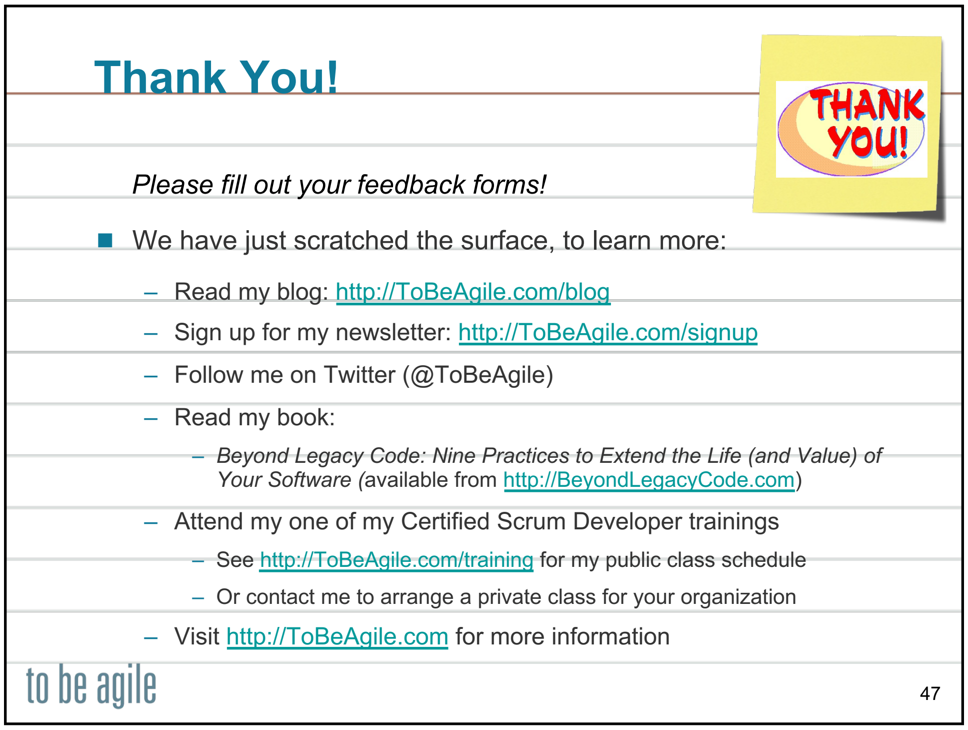


In Conclusion

- Instantiation should be a central part of any object-oriented program and should contain most of the business rules.
- Make services extensible by delegating their instantiation either to their encapsulated constructor or a factory.
- This is often the best first step for untangling legacy code.
- Object instantiation helps unleash the power of object-oriented programming to build decoupled systems that are extensible.

to be agile 46

46



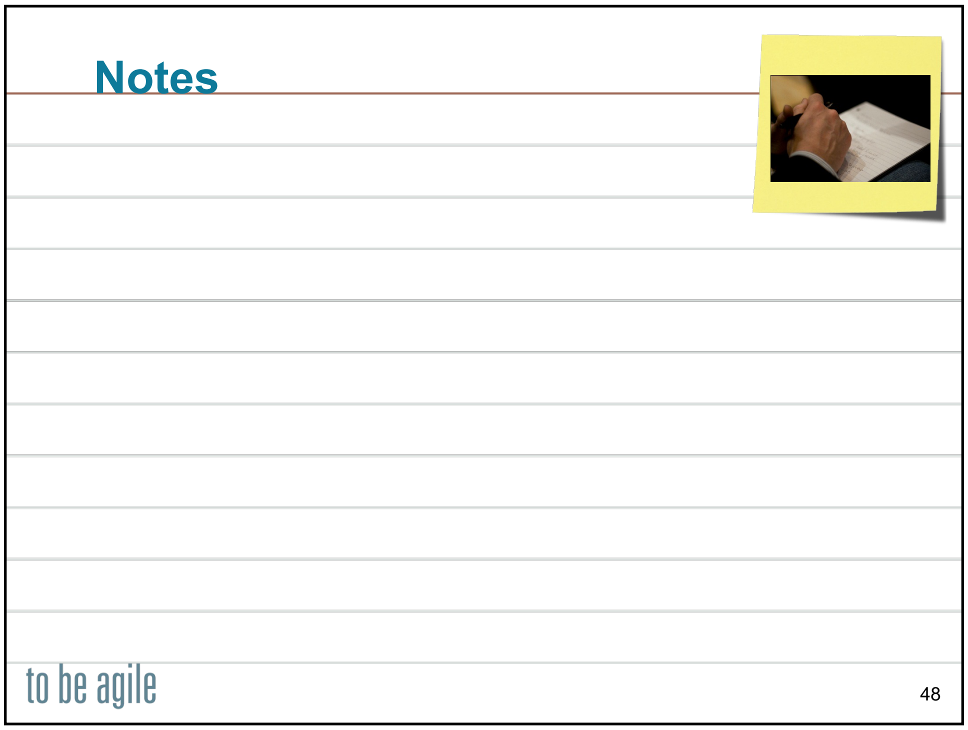
Thank You!

Please fill out your feedback forms!

- We have just scratched the surface, to learn more:
  - Read my blog: http://ToBeAgile.com/blog
  - Sign up for my newsletter: http://ToBeAgile.com/signup
  - Follow me on Twitter (@ToBeAgile)
  - Read my book:
    - *Beyond Legacy Code: Nine Practices to Extend the Life (and Value) of Your Software* (available from http://BeyondLegacyCode.com)
  - Attend my one of my Certified Scrum Developer trainings
    - See http://ToBeAgile.com/training for my public class schedule
    - Or contact me to arrange a private class for your organization
  - Visit http://ToBeAgile.com for more information

to be agile 47

47



Notes

to be agile 48

48