CSE 331
Software Design & Implementation

Hal Perkins

Winter 2023

Design Patterns, Part 1

Administrivia

- HW9 out now, due next Thursday (+late day if you have it)
  - Lots of opportunity for extras, but get the basics working first!
  - Last day of class we'd like to do a show-n-tell of hw9 projects. Please nominate yours next week (and a mix of basic projects and ones with extra goodies is great!)

- Final exam: Tue. 3/14 12:30-2:20; same rooms as midterm
  - Comprehensive but somewhat weighted towards 2nd half of the quarter.
  - Can bring two 5x8 notecards (reuse midterm for one or two completely new) – get free blank cards after class!
  - Review Q&A Mon. afternoon, 3/13, 4:30 pm, JHN 102

Outline

- Introduction to design patterns
- Creational patterns (constructing objects)

Next lecture:

- Structural patterns (controlling heap layout)
- Behavioral patterns (affecting object semantics)

What is a design pattern?

A standard solution to a common programming problem

- A design or implementation structure that achieves a particular purpose
- A high-level programming idiom

A technique for making code more flexible

- Reduce coupling among program components

Shorthand description of a software design

- Well-known terminology improves communication / documentation
- Makes it easier to "think to use" a known technique

A few simple examples….

Example 1: Encapsulation (data hiding)

Problem: Exposed fields can be directly manipulated

- Violations of the representation invariant
- Dependences prevent changing the implementation

Solution: Hide some components

- Constrain ways to access the object

Disadvantages:

- Interface may not (efficiently) provide all desired operations to all clients
- Indirection may reduce performance

Example 2: Subclassing (inheritance)

Problem: Repetition in implementations

- Similar abstractions have similar components (fields, methods)

Solution: Inherit default members from a superclass

- Select an implementation via run-time dispatching

Disadvantages:

- Code for a class is spread out, and thus less understandable
- Run-time dispatching introduces overhead
- Sometimes hard to design and specify a superclass

Example 3: Iteration

Problem: To access all members of a collection, must perform a specialized traversal for each data structure

- Introduces undesirable dependences
- Does not generalize to other collections

Solution:

- The *implementation* performs traversals, does bookkeeping
- Results are communicated to clients via a standard interface (e.g., **`hasNext()`**, **`next()`**)

Disadvantages:

- Iteration order fixed by the implementation and not under the control of the client

Example 4: Exceptions

Problem:

- Errors in one part of the code should be handled elsewhere
- Code should not be cluttered with error-handling code
- Return values should not be preempted by error codes

Solution: Language structures for throwing and catching exceptions

Disadvantages:

- Code may still be cluttered
- It may be hard to know where an exception will be handled
- Using exceptions for normal control flow may be confusing and inefficient

Example 5: Generics

Problem:

- Well-designed (and used) data structures hold one type of object

Solution:

- Programming language checks for errors in contents
- **`List<Date>`** instead of just **`List`**

Disadvantages:

- More verbose types

Why (more) design patterns?

Advanced programming languages like Java provide many powerful constructs – subtyping, interfaces, rich types and libraries, etc.

- But it's not possible to "put everything in the language"
- Still many common problems not easy to solve

Design patterns are intended to capture common solutions / idioms, name them, make them easy to use to guide design

- For high-level design, not specific "coding tricks"

They increase your vocabulary and your intellectual toolset

Do not overuse them

- Not every program needs the complexity of advanced design patterns
- Instead, consider them to solve reuse/modularity problems that arise as your program evolves

Why should you care?

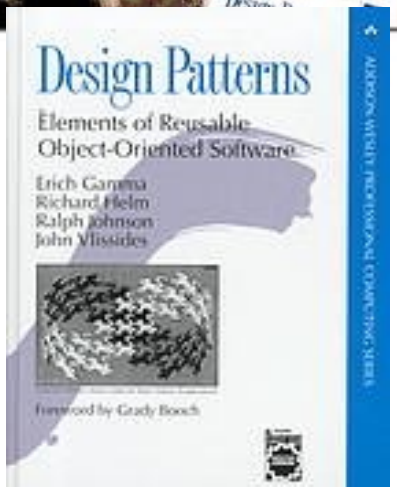
You could come up with these solutions on your own

- You shouldn't have to!

A design pattern is a known solution to a known problem

- A concise description of a successful “pro-tip”

Origin of term



The “Gang of Four” (GoF)

- Gamma, Helm, Johnson, Vlissides

Found they shared a number of “tricks” and decided to codify them

- A key rule was that nothing could become a pattern unless they could identify at least three real [different] examples
- Done for object-oriented programming
  - Some patterns more general; others compensate for OOP shortcomings
  - But any “paradigm” should have design patterns

Patterns vs. Patterns

The phrase *pattern* has been wildly overused since the GoF patterns have been introduced

Misused as a synonym for "[somebody says] **X** is a good way to write programs."

- And "anti-pattern" has become a synonym for "[somebody says] **Y** is a bad way to write programs."

GoF-style patterns have richness, history, language-independence, documentation and thus (most likely) far more staying power

Widely implemented and described for many languages: see *Effective Java* for specific applications to Java

When (not) to use design patterns

- Rule 1: delay
  - Get something basic and concrete working first
  - Improve or generalize once you understand it
- Design patterns can increase or decrease understandability
  - Usually adds (some) indirection, increases code size
  - Improves modularity and flexibility, separates concerns, eases description
- If your design or implementation has a problem, consider design patterns that address that problem

An example GoF pattern

For some class `C`, guarantee that at run-time there is exactly one instance of `C`

- And that the instance is globally visible

First, *why* might you want this?

- What design goals are achieved?

Second, *how* might you achieve this?

- How to leverage language constructs to enforce the design

A pattern has a recognized *name*

- This is the *Singleton Pattern*

Possible reasons for Singleton

- Only one instance of the given type exists; shared resources
- Examples:
  - One cache
  - One **`RandomNumber`** generator
  - One **`KeyboardReader`**, **`PrinterController`**, etc…
  - Single logger for messages; single configuration file
- An object that has fields like "static fields" but a constructor decides their values
  - Example: strings in a particular language for messages
- Make it easier to ensure some key invariants
  - There is only one instance, so never mutate the wrong one
- Make it easier to control when that single instance is created
  - If expensive, delay until needed and then don't do it again

How: multiple approaches

```
public class Foo {
  private static final Foo instance = new Foo();
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static Foo getInstance() {
    return instance;
  }
  … instance methods as usual …
}
```

Eager allocation of instance

```
public class Foo {
  private static Foo instance;
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static synchronized Foo getInstance() {
    if (instance == null) {
      instance = new Foo();
    }
    return instance;
  }
  … instance methods as usual …
}
```

Lazy allocation of instance

GoF patterns: three categories

Creational Patterns are about the object-creation process

Factory Method, Abstract Factory, Singleton, Builder, Prototype, …

Structural Patterns are about how objects/classes can be combined

Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy, …

Behavioral Patterns are about communication among objects

Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method, …

Green = ones we've seen already

Purple = ones we've at-least-sorta-used without knowing/naming it

Creational patterns

Constructors in Java are inflexible

1. Can't return a subtype of the class they belong to
2. Always return a fresh new object, never re-use one

Factories: Patterns for code that you call to get new objects other than constructors

– Factory method, Factory object, Prototype, Dependency injection

Sharing: Patterns for reusing objects (to save space *and* other reasons)

– Singleton, Interning, Flyweight

Factories

Problem: want better control over object creation

Factory method = creator method
- Hides decisions about object creation
- Method can do any computation and return any subtype

Factory object = has creator op, can be passed around
- Bundles factory methods for a family of types
- Code goes into a separate object

Builder = create complex objects incrementally
- Allows step-by-step construction of complex objects
- Move construction logic for an object outside the class constructor

Prototype = knows how to clone itself
- Every object is a factory, can create more objects like itself
- Call `clone` to get a new object of same subtype as receiver

Dependency Injection = external reference to creator operation
- Client controls construction without changing code
- Implementation: read method name from file, call reflectively

Motivation for factories: Changing implementations

Supertypes support multiple implementations

```
interface Matrix { ... }
class SparseMatrix implements Matrix { ... }
class DenseMatrix implements Matrix { ... }
```

Clients use the supertype (**`Matrix`**)

Still need to use a **`SparseMatrix`** or **`DenseMatrix`** constructor

- Must decide concrete implementation *somewhere*
- Don't want to change code to use a different constructor
- Factory methods put this decision behind an abstraction

Use of factories

Factory

```
class MatrixFactory {
  public static Matrix createMatrix() {
    return new SparseMatrix();
  }
}
```

Clients call **`createMatrix`** instead of a particular constructor

Advantages:

- To switch the implementation, change only *one* place
- **`createMatrix`** could do arbitrary computations to decide what kind of matrix to make (unlike what's shown above)

Examples in the JDK

```
class Calendar {
  static Calendar getInstance(Locale);
}
```

- Might return a **`BuddhistCalendar`**, **`JapaneseImperialCalendar`**, **`GregorianCalendar`**, …

`DateFormat` factory methods

`DateFormat` class encapsulates knowledge about how to format dates and times as text

- Options: just date? just time? date+time? where in the world?
- Instead of passing all options to constructor, use factories
- The subtype created by factory call need not be specified

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance
                    (DateFormat.FULL, Locale.FRANCE);

Date today = new Date();

df1.format(today)  // "Jul 4, 1776"
df2.format(today)) // "10:15:00 AM"
df3.format(today));// "jeudi 4 juillet 1776"
```

Example: Bicycle race (no factories)

```
class Race {
  public Race() {
    Bicycle bike1 = new Bicycle();
    Bicycle bike2 = new Bicycle();
    …
  }
  …
}
```

New example:

- No factories yet
- Coming: factories for the *bicycles* to get flexibility and code reuse
- Could also use factories for the *races*, but that complicates the example, so will stick with constructors

Specialization: Tour de France

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    Bicycle bike1 = new RoadBicycle();
    Bicycle bike2 = new RoadBicycle();
    …
  }
  …
}
```

The problem: We are reimplementing the constructor in every **`Race`** subclass just to use a different subclass of **`Bicycle`**

Specialization:

```
class Cyclocross extends Race {
  public Cyclocross() {
    Bicycle bike1 = new MountainBicycle();
    Bicycle bike2 = new MountainBicycle();
    …
  }
  …
}
```

The problem: We are reimplementing the constructor in every **`Race`** subclass just to use a different subclass of **`Bicycle`**

Factory *method* for Bicycle

```
class Race {
  Bicycle createBicycle() { return new Bicycle(); }
  public Race() {
    Bicycle bike1 = createBicycle();
    Bicycle bike2 = createBicycle();
    ...
  }
}
```

Use a factory method to avoid dependence on specific new kind of bicycle in the constructor

- Call the factory method instead

Subclasses override factory method

```
class TourDeFrance extends Race {
  Bicycle createBicycle() {
    return new RoadBicycle();
  }
  public TourDeFrance() { super(); }
}
class Cyclocross extends Race {
  Bicycle createBicycle() {
    return new MountainBicycle();
  }
  public Cyclocross() { super(); }
}
```

- "Foresight" to use factory method in superclass constructor
- Then dynamic dispatch will call overridden method
- Subtyping in the overriding methods (covariant returns type also ok)

Next step

- **`createBicycle`** was just a factory method

- Now let's move the method into a separate class
  - So it's part of a *factory object*

- Advantages:
  1. Can group related factory methods together
     - Not shown: **`repairBicycle`**, **`createSpareWheel`**, …
  2. Can pass factories around as objects for flexibility
     - Choose a factory at runtime
     - Use different factories in different objects (e.g., races)
     - Example…

Factory ***objects*/classes** encapsulate factory method(s)

```
class BicycleFactory {
  Bicycle createBicycle() {
   return new Bicycle();
  }
  Frame createFrame() { ... }
}
class RoadBicycleFactory extends BicycleFactory {
  Bicycle createBicycle() {
    return new RoadBicycle();
  }
}
class MountainBicycleFactory extends BicycleFactory {
  Bicycle createBicycle() {
    return new MountainBicycle();
  }
}
```

Using a factory object

```
class Race {
  BicycleFactory bfactory;
  public Race(BicycleFactory f) {
    bfactory = f;
    Bicycle bike1 = bfactory.createBicycle();
    Bicycle bike2 = bfactory.createBicycle();
    …
  }
  public Race() { this(new BicycleFactory()); }
  …
}
```

Setting up the flexibility here:

- Factory object stored in a field, set by constructor
- Can take the factory as a constructor-argument

The subclasses

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    super(new RoadBicycleFactory());
  }
}

class Cyclocross extends Race {
  public Cyclocross() {
    super(new MountainBicycleFactory());
  }
}
```

Voila!

- Just call the superclass constructor with a different factory
- **`Race`** class had foresight to delegate "what to do to create a bicycle" to the factory object, making it more reusable

Separate control over bicycles and races

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    super(new RoadBicycleFactory()); // or this(…)
  }
  public TourDeFrance(BicycleFactory f) {
    super(f);
  }
  …
}
```

By having factory-as-argument option, we can allow arbitrary mixing by client: **`new TourDeFrance(new TricycleFactory())`**

Less useful in this example (?): Swapping in different factory object whenever you want

Builder pattern

Motivations

- Sometimes we don't have all the information needed to completely build an object when we first create it, so the logic to build the object needs to be outside the class constructor
  - Sometimes we want to construct an object step-by-step as we accumulate more information to be incorporated
  - Side benefit: can avoid constructors with excessive numbers of parameters or excessively complex logic

- Idea: Construct basic object, then call methods that update the object state and return a reference to the object
  - These builder method calls are often chained to build the final object

Builder example

- Build OpenCSV parser/iterator (another way to parse csv data)
  - After creating the basic object, each method call modifies the object and returns a reference to it for the next call

```
Reader reader =
        Files.newBufferedReader(Paths.get(”..."));

Iterator<UserModel> csvUserIterator =
  new CsvToBeanBuilder<UserModel>(reader)  // input
        .withType(UserModel.class)      // entry type
        .withSeparator(',')
        .withIgnoreLeadingWhiteSpace(true)
        .build()       // return CsvToBean<UserModel>
        .iterator();
```

Prototype pattern

- Every object is itself a factory
- Each class contains a **`clone`** method that creates a copy of the receiver object

```
class Bicycle {
  Bicycle clone() { ... }
}
```

Often, **`Object`** is the return type of **`clone`**

- **`clone`** is declared in **`Object`**
- Design flaw in Java 1.4 and earlier: the return type could not change covariantly in an overridden method

  i.e., return type could not be made more restrictive

Using prototypes

```
class Race {
  Bicycle bproto;

  public Race(Bicycle bproto) {
    this.bproto = bproto;
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
  }
}
```

Again, we can specify the race and the bicycle separately:

```
new Race(new Tricycle())
```

Dependency injection

- Change the factory without changing the code
- With a regular in-code factory:

```
BicycleFactory f = new TricycleFactory();
Race r = new TourDeFrance(f)
```

- With external dependency injection:

```
BicycleFactory f = ((BicycleFactory)
      DependencyManager.get("BicycleFactory"));
Race r = new TourDeFrance(f);
```

- *Plus* an external file:

```
<service-point id="BicycleFactory">
  <invoke-factory>
    <construct class="Bicycle">
      <service>Tricycle</service>
    </construct>
  </invoke-factory>
</service-point>
```

+ Change the factory without recompiling
- External file is essential part of program
- Errors in file not caught until runtime

Sharing

Recall the second weakness of Java constructors

Java constructors always return a *new object*

Singleton: only one object exists at runtime

- Factory method returns the same object every time
- (we've seen this already)

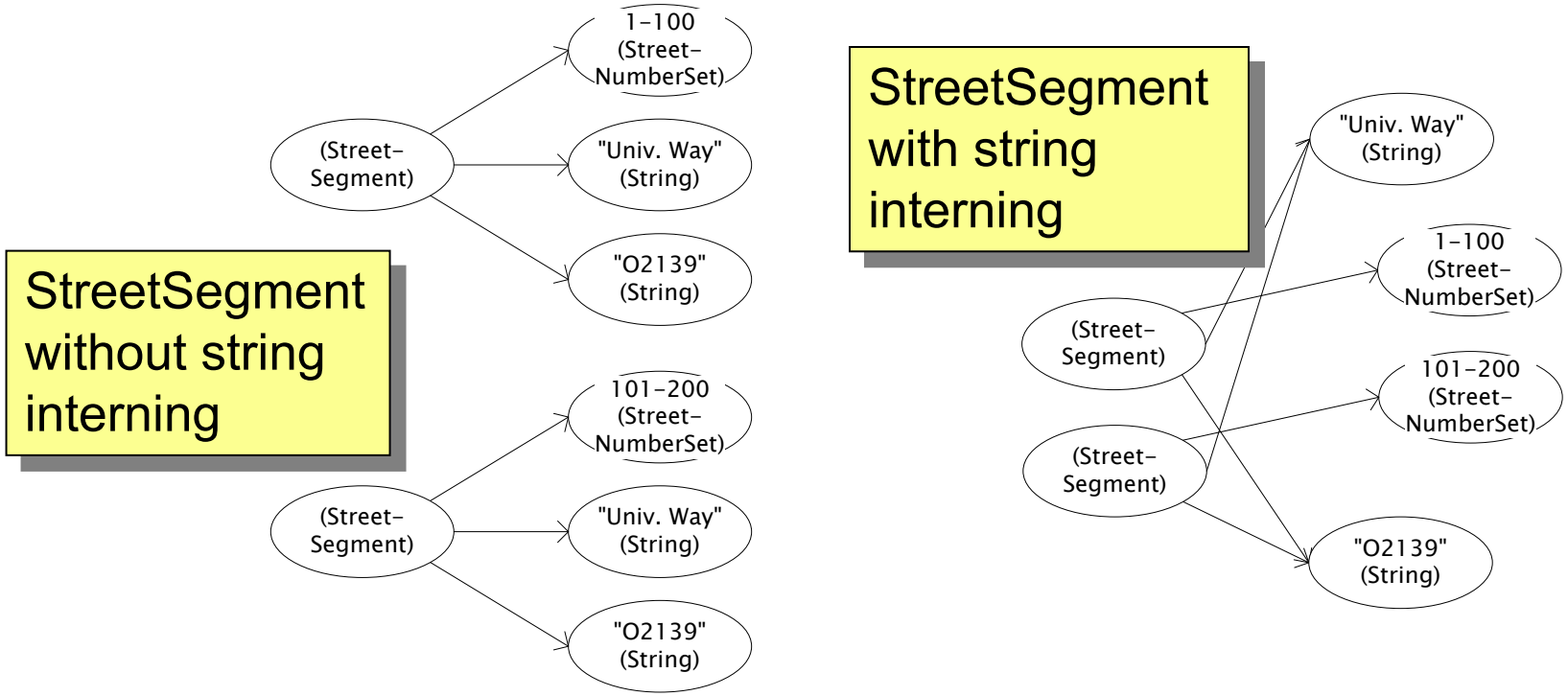
Interning: only one object with a particular (abstract) value exists at runtime

- Factory method returns an existing object, not a new one

Flyweight: separate intrinsic and extrinsic state, represent them separately, and intern the intrinsic state

Interning pattern

- Reuse existing objects instead of creating new ones
  - Less space
  - Then can compare with `==` instead of **`equals()`**
- Sensible only for immutable objects



Implementing interning

- Maintain a collection of all objects
- If an object already appears, return that instead

```
HashMap<String, String> segnames;
String canonicalName(String n) {
  if (segnames.containsKey(n)) {
    return segnames.get(n);
  } else {
    segnames.put(n, n);
    return n;
  }
}
```

Why not `Set<String>` ?

`Set` supports `contains` but not `get`

- Two approaches:
  - Create the object, but perhaps discard it and return another
  - Check against the arguments before creating the new object
- Java builds this in for strings: `String.intern()`

Space leaks

- Interning can waste space if your collection:
  - Grows too big
  - With objects that will never be used again

- Not discussed here: The solution is to use *weak references*
  - This is their canonical purpose

- Do not reinvent your own way of keeping track of whether an object in the collection is being used
  - Too error-prone
  - Gives up key benefits of garbage-collection

`java.lang.Boolean`
does not use the Interning pattern

```
public class Boolean {
  private final boolean value;
  // construct a new Boolean value
  public Boolean(boolean value) {
    this.value = value;
  }

  public static Boolean FALSE = new Boolean(false);
  public static Boolean TRUE = new Boolean(true);

  // factory method that uses interning
  public static Boolean valueOf(boolean value) {
    if (value) {
      return TRUE;
    } else {
      return FALSE;
    }
  }
}
```

Recognition of the problem

Javadoc for **`Boolean`** constructor:

> Allocates a **`Boolean`** object representing the value argument.
>
> **Note: It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.**

Josh Bloch (JavaWorld, January 4, 2004):

> The **`Boolean`** type should not have had public constructors. There's really no great advantage to allow multiple **`true`**s or multiple **`false`**s, and I've seen programs that produce millions of **`true`**s and millions of **`false`**s, creating needless work for the garbage collector.
>
> So, in the case of immutables, I think factory methods are great.

Flyweight pattern – save space: don't store same data more than once

Good when many objects are *mostly* the same

- Interning works only if objects are *entirely* the same (and immutable)

Intrinsic state: Independent of object's "context"

- Often same across many objects and immutable
- Technique: intern it

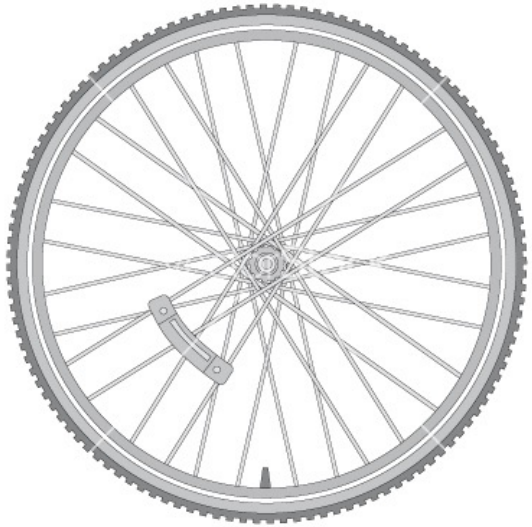
Extrinsic state: different for different objects; depends on "context"

- Have clients store it separately, or better:
- Advanced technique:
  - Make it implicit (clients *compute* it instead of represent it)
  - Saves space

Details omitted 23wi due to time…

Example without flyweight: bicycle spoke

```
class Wheel {
  FullSpoke[] spokes;
  ...
}
class FullSpoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
  int location;  // position on the rim
}
```



- Typically 32 or 36 spokes per wheel but only 3 varieties per bicycle
- In a bike race, hundreds of spoke varieties, millions of instances

Alternatives to `FullSpoke`

```
class IntrinsicSpoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
}
```

This does *not* save space compared to `FullSpoke`

```
class InstalledSpokeFull extends IntrinsicSpoke {
  int location;
}
```

This *does* saves space

```
class InstalledSpokeWrapper {
  IntrinsicSpoke s; // refer to interned object
  int location;
}
```

But flyweight version [still coming up] uses even less space…

Original code to true (align) a wheel

```
class FullSpoke {
  // Tension the spoke by turning the nipple the
  // specified number of turns.
  void tighten(int turns) {
    ... location ...    // location is a field
  }
}

class Wheel {
  FullSpoke[] spokes;
  void align() {
    while (wheel is misaligned) {
      // tension the ith spoke
      ... spokes[i].tighten(numturns) ...
    }
  }
}
```

What is the value of the **`location`** field in **`spokes[i]`**?

Flyweight code to true (align) a wheel

```
class IntrinsicSpoke {
  void tighten(int turns, int location) {
    ... location ...   // location is a parameter
  }
}

class Wheel {
  IntrinsicSpoke[] spokes;

  void align() {
    while (wheel is misaligned) {
      // tension the ith spoke
      ... spokes[i].tighten(numturns, i) ...
    }
  }
}
```

What happened

- *Logically*, each spoke is a different object
  - A spoke "has" all the intrinsic state and a location
- But if that would be a lot of objects, i.e., space usage, we can instead…
- Create *one actual* flyweight object that is used "in place of" all logical objects that have that intrinsic state
  - Use interning to get the sharing
  - Clients store or compute the extrinsic state and pass it to methods to get the right behavior
  - Only do this when logical approach is cost-prohibitive and it's not too complicated to manage the extrinsic state
    - Here spoke location was particularly easy and cheap because it was implicit in array location of reference

Flyweight discussion

What if **`FullSpoke`** contains a **`wheel`** field pointing at the **`Wheel`** containing it?

> **`Wheel`** methods pass this to the methods that use the **`wheel`** field.

What if **`FullSpoke`** contains a **`boolean`** field **`broken`**?

> Add an array of **`boolean`**s in **`Wheel`**, parallel to the array of **`Spoke`**s.

Flyweight: rarely used – resist it

- Flyweight is manageable only if there are very few mutable (extrinsic) fields
- Flyweight complicates the code
- Use flyweight only when profiling has determined that space is a *serious* problem

Next up…

- Structural patterns
- Behavioral patterns