Creational Design Patterns

Paul Fodor

CSE316: Fundamentals of Software Development

Stony Brook University

http://www.cs.stonybrook.edu/~cse316

Design Patterns

- Design Pattern
  - A description of a problem and its solution that you can apply to many similar programming situations
- Patterns:
  - facilitate reuse of good, tried-and-tested solutions
  - capture the structure and interaction between components

Why is this important?

- Using proven, effective design patterns can make you a better software *designer* & *coder*
- You will recognize commonly used patterns in others' code
  - JS, Java, python APIs
  - Project team members

Different technologies have their own patterns: Web Patterns, Architectural Patterns.

- And you'll learn when to apply them to your own code
  - experience reuse (as opposed to code reuse)
  - you want to think at the pattern level: recognize a problem and apply the appropriate solution to it
- Greatest advantage of patterns: allows easy CHANGE/UPDATE/Maintenance of applications

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

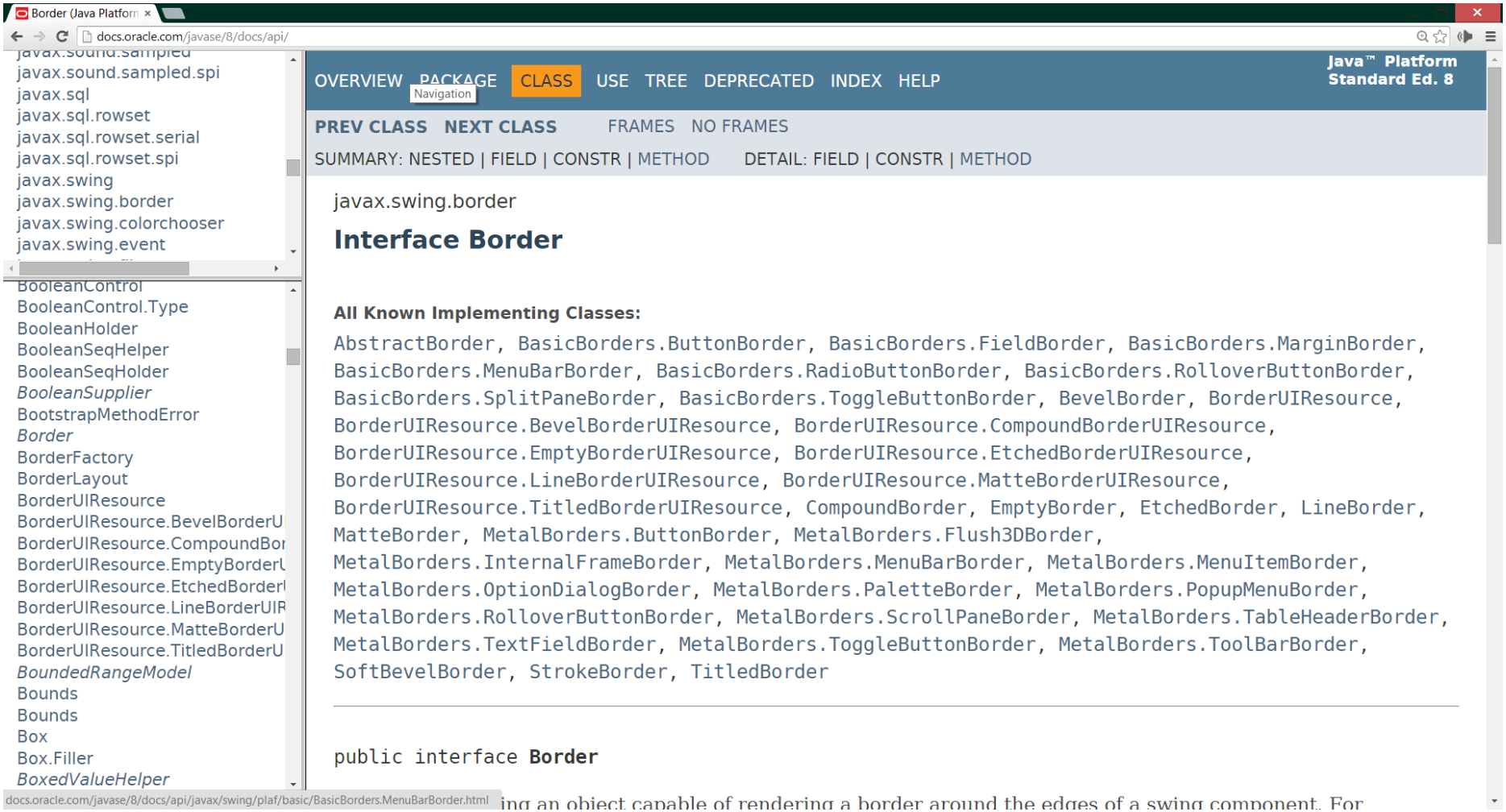
Textbook: Head First Design Patterns

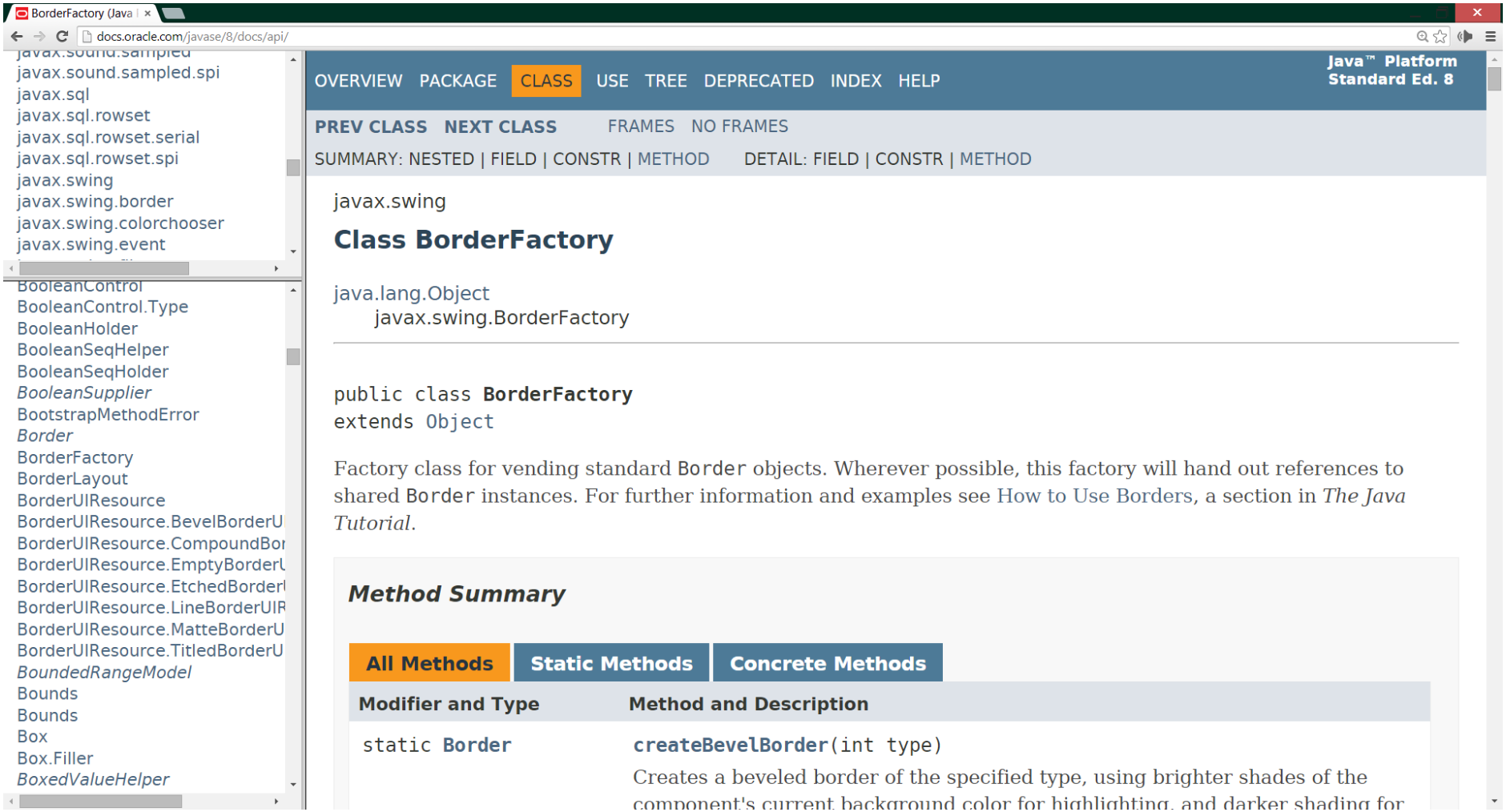
The Factory Pattern

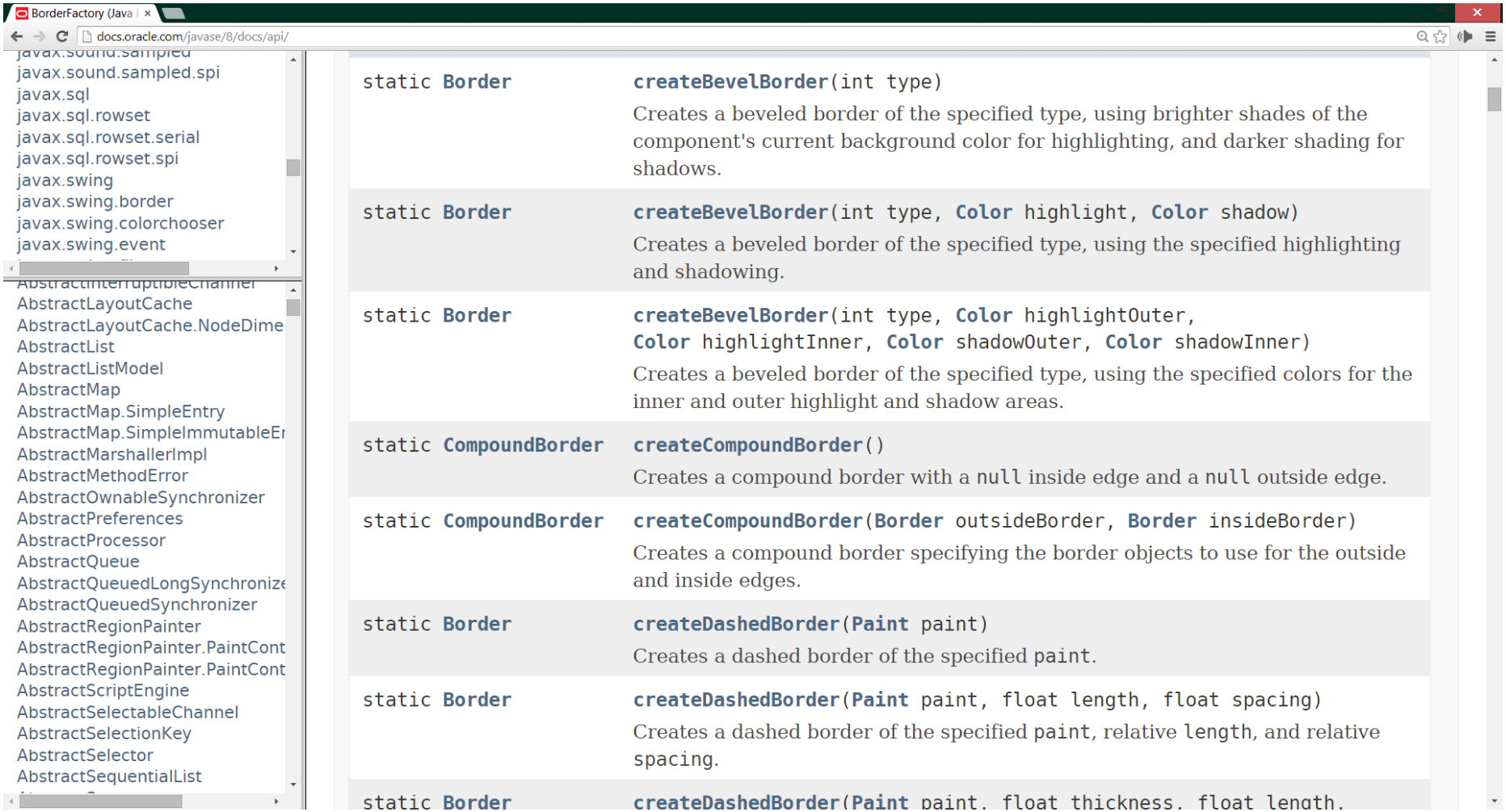
- Factories make stuff
- Factory classes make objects
- Shouldn't constructors do that? Yes, called by the *new* operator.
  - Internally, factory classes employ constructors
- What's the point of the Factory pattern?
  - prevent misuse/improper construction
  - hides construction
  - provide API convenience
  - one stop shop for getting an object of a family type

What objects do factories make?

- Typically objects of the same family
  - common ancestor
  - same apparent type
  - **different actual type**
- Example of Factory Patterns in the Java SWING API:
  - BorderFactory.createXXXBorder methods
    - return apparent type of interface Border
    - return actual types of BevelBorder, EtchedBorder, etc …
  - factory classes in security packages:
    - java.security.KeyFactory
    - java.security.cert.CertificateFactory



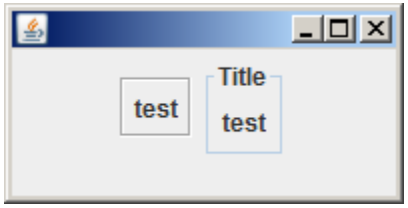




Border Example

```
…
JPanel panel = new JPanel();
Border border =
  BorderFactory.createEtchedBorder();
panel.setBorder(border);

JPanel panel2 = new JPanel();
Border border2 =
  BorderFactory.createTitledBorder("Title");
panel2.setBorder(border2);
…
```

```
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.Border;
public class FactoryExample {
  public static void main(String[] args) {
    JFrame f = new JFrame();
    f.setSize(200, 100);
    JPanel panel = new JPanel();
    f.add(panel);
    JPanel panel1 = new JPanel();
    panel.add(panel1);
    panel1.add(new JLabel("test"));
    Border border = BorderFactory.createEtchedBorder();
    panel1.setBorder(border);
    JPanel panel2 = new JPanel();
    panel.add(panel2);
    panel2.add(new JLabel("test"));
    Border border2 = BorderFactory.createTitledBorder("Title");
    panel2.setBorder(border2);
    f.setVisible(true);
  }
}
```



How to implement a Factory Pattern?

```
interface Border{
}
class EtchedBorder implements Border{
  // Border Methods
}
class TitledBorder implements Border{
  // Border Methods
}
public class BorderFactory{
  public static Border createEtchedBorder(){
       return new EtchedBorder();
  }
  public static Border createTitledBorder(String title){
       return new TitledBorder();
  }
}
```

Factory Pattern Advantages

- The programmer using the Factory class never needs to know about the actual class/type:
  - simplifies use for programmer
  - fewer classes to learn
  - For Example: Using BorderFactory, one only needs to know Border & BorderFactory
    - NOT TitledBorder, BeveledBorder, EtchedBorder, AbstractBorder, BasicBorders.ButtonBorder, BasicBorders.FieldBorder, BasicBorders.MarginBorder, BasicBorders.MenuBarBorder, BasicBorders.RadioButtonBorder, BasicBorders.RolloverButtonBorder, BasicBorders.SplitPaneBorder, BasicBorders.ToggleButtonBorder, CompoundBorder, EmptyBorder, EtchedBorder, LineBorder, MatteBorder, MetalBorders.ButtonBorder, MetalBorders.Flush3DBorder, MetalBorders.InternalFrameBorder, MetalBorders.MenuBarBorder, MetalBorders.MenuItemBorder, MetalBorders.OptionDialogBorder, MetalBorders.PaletteBorder, MetalBorders.PopupMenuBorder, etc.

```
// Cars example: Dealer.java
abstract class Car {
}
class Bmw extends Car {
}
class Bmw320 extends Bmw {
}
abstract class CarFactory {
        public abstract Car createCar(String type);
}
class BmwFactory extends CarFactory {  // the factory can use inheritance
        @Override          // employ the Command design pattern (you pass as a
                           //  String the type of class that you want to create)
        public Car createCar(String type) {
                if("Bmw320".equals(type)) {
                        return new Bmw320();
                }
                else return new Bmw();
        }
}
public class Dealer {
        public static void main(String[] args) {
                // this also employes Command design pattern to pass a String
                Car bmw1 = new BmwFactory().createCar("Bmw320");
                Car bmw2 = new BmwFactory().createCar("Bmw");
                //Car camry1 = new ToyotaFactory().createCar("Camry");
                System.out.println(bmw1);
                System.out.println(bmw2);
        }
}
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

The Singleton Pattern

- What makes a good singleton candidate?
  - central app organizer class
    - Example: a basic/simple Web/FTP server can be a singleton class
  - something everybody needs
    - Example: a class that stores global properties for the application, a logging service class.
- Advantage: we don't need to pass it as a parameter in methods
- Define a type where only one object of that type may be constructed:
  - make the constructor private! AND
  - implement a method that returns the same static object all the time

Example: The PropertiesManager Singleton

```
public class PropertiesManager {
  private static PropertiesManager singleton;
  private PropertiesManager() {}
  public static PropertiesManager
          getPropertiesManager() {
      if (singleton == null) {
          singleton = new PropertiesManager();
      }
      return singleton;
  }
}
```

- Singleton object favorable to fully static class, why?
  - can be used as a method argument
  - class can be extended

Example: The PropertiesManager Singleton

```
public static void main(String[] args) {
    PropertiesManager p1 =
     PropertiesManager.getPropertiesManager();
    PropertiesManager p2 =
     PropertiesManager.getPropertiesManager();
    System.out.println(p1);
    System.out.println(p2);
}
```

```
cse316.PropertiesManager@15db9742
cse316.PropertiesManager@15db9742
```

What's so great about a singleton?

- Other classes may now easily USE the PropertiesManager

```
PropertiesManager singleton =
   PropertiesManager.getPropertiesManager();
```

- Don't have to worry about passing objects around
- Don't have to worry about object consistency

Singleton Pattern Examples in public APIs

- Java API:
  - java.lang.Runtime.getRuntime():

    `public static Runtime getRuntime()`

    - Returns the runtime object associated with the current Java application. Most of the methods of class Runtime are instance methods and must be invoked with respect to the current runtime object.
  - java.awt.Desktop.getDesktop()

    `public static Desktop getDesktop()`

    - Returns the Desktop instance of the current browser context. On some platforms the Desktop API may not be supported; use the isDesktopSupported() method to determine if the current desktop is supported.

Singleton Pattern Examples in public APIs

```
public class APIsExamples {
        public static void main(String[] args) {
                Runtime r1 = java.lang.Runtime.getRuntime();
                Runtime r2 = java.lang.Runtime.getRuntime();
                System.out.println(r1);
                System.out.println(r2);

                java.awt.Desktop d1 = java.awt.Desktop.getDesktop();
                java.awt.Desktop d2 = java.awt.Desktop.getDesktop();
                System.out.println(d1);
                System.out.println(d2);
        }
}
```

- java.lang.Runtime@15db9742
- java.lang.Runtime@15db9742
- java.awt.Desktop@3d075dc0
- java.awt.Desktop@3d075dc0

java.lang

Class Runtime

java.lang.Object
  └ java.lang.Runtime

```
public class Runtime
extends Object
```

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in wh

An application cannot create its own instance of this class.

Since:

JDK1.0

See Also:

`getRuntime()`

Method Summary

void	`addShutdownHook(Thread hook)` Registers a new virtual-machine shutdown hook.
int	`availableProcessors()` Returns the number of processors available to the Java virtual machine.
Process	`exec(String command)` Executes the specified string command in a separate process.
Process	`exec(String[] cmdarray)` Executes the specified command and arguments in a separate process.
Process	`exec(String[] cmdarray, String[] envp)` Executes the specified command and arguments in a separate process with the specified environment.
Process	`exec(String[] cmdarray, String[] envp, File dir)`

```
class Singleton {                                  // Other versions of Singleton
  private static Singleton instance = new Singleton(); // eager init
  private Singleton() {
  }
  public static Singleton getInstance() {
    return instance;
  }
  // alternative - lazy init
  public synchronized static Singleton getInstanceSync() {
    if (instance == null) {
      instance = new Singleton();
    }
      return instance;
    }
}
public class Main {
  public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();  // getInstanceSync();
    Singleton s2 = Singleton.getInstance();  // getInstanceSync();
    System.out.println(s1.hashCode());
    System.out.println(s2.hashCode());
        System.out.println(SingletonEnum.Elvis.getSong());
  }
}
enum SingletonEnum {
  // there was only one Elvis ...
  Elvis;
  public String getSong() {
    return "Heartbreak";
  }
}
```

```
366712642
366712642
Heartbreak
```

Common Design Patterns

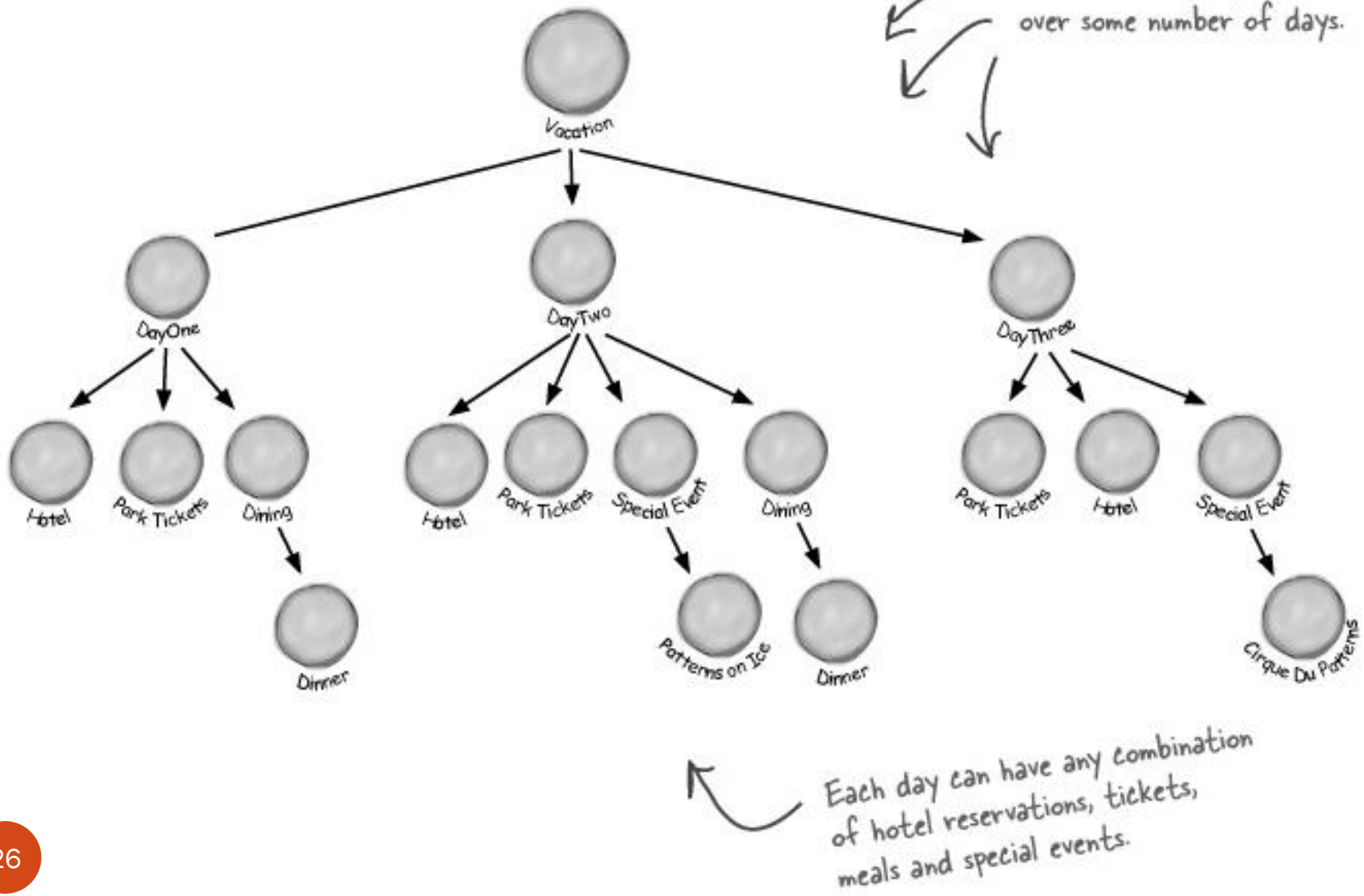
Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

The Builder Pattern

- Use the Builder Pattern to:
  - encapsulate the construction of a complex product
  - allow it to be constructed in steps
- Good for complex object construction
  - objects that require lots of custom initialized pieces
- Example Scenario:
  - build a vacation planner for a theme park
    - guests can choose a hotel, tickets, events, meals, travel, etc.
      - guests may want zero to many of each
    - create a plan builder to encapsulate all this info incrementally

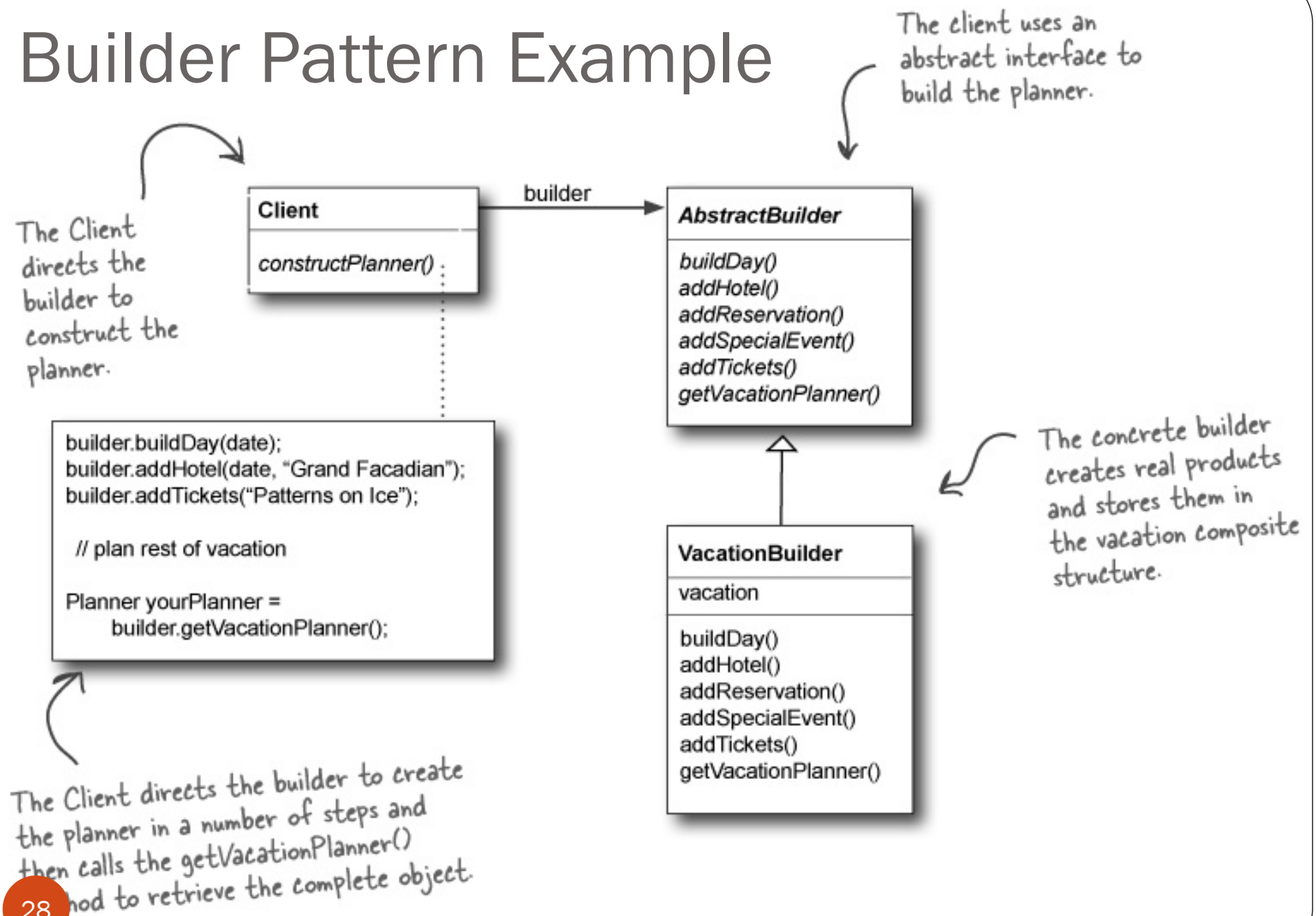
A Builder Pattern Example



So what's the problem?

- A flexible construction design is needed
- Lots of Customization:
  - some customers might not want a hotel
  - some might want multiple rooms in multiple hotels
  - some might want restaurant reservations
  - some might want stuff no one else does
- We need:
  - a flexible data structure that can represent guest planners and all their variations
  - a sequence of potentially complex steps to create the planner

Builder Pattern Example



A (simpler) Builder example

- We want to create an **immutable** user, but we don't know all the properties about the user:

```
public class User {
    private final String firstName;    //required
    private final String lastName;     //required
    private final int age;             //optional
    private final String phone;        //optional
    private final String address;      //optional
```

- A first and valid option would be to have a constructor that only takes the required attributes as parameters, one that takes all the required attributes plus the first optional one, another one that takes two optional attributes and so on.

```
public User(String firstName, String lastName) {
    this(firstName, lastName, 0);
}
public User(String firstName, String lastName, int age) {
    this(firstName, lastName, age, "");
}
public User(String firstName, String lastName, int age,
          String phone) {
    this(firstName, lastName, age, phone, "");
}
public User(String firstName, String lastName, int age,
          String phone, String address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.phone = phone;
    this.address = address;
}...
```

- The Builder Pattern solution:

```
public class User {
    private final String firstName; // required
    private final String lastName;  // required
    private final int age;          // optional
    private final String phone;     // optional
    private final String address;   // optional
    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public int getAge() {
        return age;
    }
```

```
public static class UserBuilder { // inner class
      private final String firstName;
      private final String lastName;
      private int age;
      private String phone;
      private String address;
      public UserBuilder(String firstName,
              String lastName) {
          this.firstName = firstName;
          this.lastName = lastName;
      }
      public UserBuilder age(int age) {
          this.age = age;
          return this;
      }
      public UserBuilder phone(String phone) {
          this.phone = phone;
          return this;
      }
      public UserBuilder address(String address) {
          this.address = address;
          return this;
      }
```

```
        public User build() {
            return new User(this);
        }
    }
```

/* The User constructor is private, which means that this class can not be directly instantiated from the client code.
- The class is immutable. All attributes are final and they're set in the constructor. We only provide getters for them. */

```
    public User getUser() {
        return new
                User.UserBuilder("John", "Smith")
                .age(50)
                .phone("1234567890")
                .address("Main St. 1234")
                .build();
    }
}
```

Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out if the client only sees an abstract interface.

```
class Vacation {                    // VacationBuilder Example
        private List<Person> persons = new ArrayList<Person>();
        private Hotel hotel;
        private Reservation reservation;
        private List<Activity> activities = new ArrayList<Activity>();
        public void addPerson(Person person) {
                this.persons.add(person);
        }
        public void setHotel(Hotel hotel) {
                this.hotel = hotel;
        }
        public void setReservation(Reservation reservation) {
                this.reservation = reservation;
        }
        public void addActivity(Activity activity) {
                this.activities.add(activity);
        }
        public String show() {
                String result = "";
                result += persons;
                result += hotel;
                result += reservation;
                result += activities;
                return result;
        }
}
```

```
enum Activity {
        RUNNING, RELAXING, SWIMMING
}
class Hotel {
        private String name;
        public Hotel(String name) {
                setName(name);
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
}
class Person {
        private String lastName;
        private String firstName;
        private Date dateOfBirth;
        public Person(String lastName,String firstName,Date dateOfBirth){
                this.lastName = lastName;
                this.firstName = firstName;
                this.dateOfBirth = dateOfBirth;
        }
}
```

```
class Reservation {
        private Date startDate;
        private Date endDate;
        public Reservation(Date in, Date out) {
                this.startDate = in;
                this.endDate = out;
        }
}
class VacationBuilder {
        private static VacationBuilder builder = new VacationBuilder();
        private VacationBuilder() {
        }
        public static VacationBuilder getInstance() {
                return builder;
        }
        private Vacation vacation = new Vacation();
        public void addPerson(String firstName, String lastName) {
                Person p = new Person(lastName, firstName, new Date());
                this.vacation.addPerson(p);
        }
        public void setHotel(String name) {
                this.vacation.setHotel(new Hotel(name));
        }
        public void addActivity(Activity activity) {
                this.vacation.addActivity(activity);
        }
```

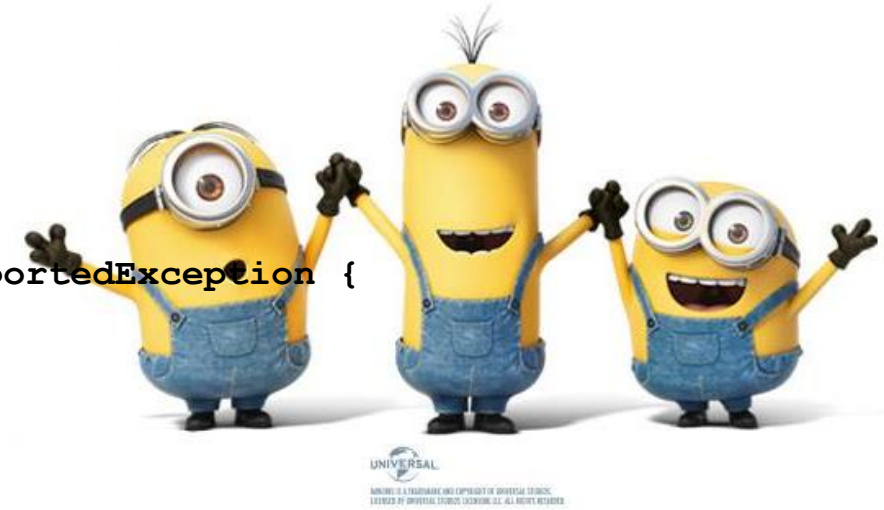
```
        public void setReservation(String in, String uit) {
                Date inDate = new Date();
                Date outDate = new Date(new Date().getTime() + 10000);
                Reservation reservation = new Reservation(inDate, outDate)
                this.vacation.setReservation(reservation);
        }

        public Vacation getVacation() {
                return this.vacation;
        }
        public static void main(String[] args) {
                VacationBuilder builder = VacationBuilder.getInstance();
                builder.addActivity(Activity.RUNNING);
                builder.addPerson("Smith", "John");
                builder.setHotel("ACME Hotel");
                builder.setReservation("1-2-2015", "1-8-2015");
                Vacation vacation = builder.getVacation();
                String show = vacation.show();
                System.out.println(show);
        }
}
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

```
// A factory might store a set of Prototypes from which to clone and return product objects.
interface Minion extends Cloneable {
  Object clone() throws CloneNotSupportedException;
}
class Stuart implements Minion {
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
  public String toString() {
    return "Stuart";
  }
}
class Kevin implements Minion {
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
  public String toString() {
    return "Kevin";
  }
}
class Bob implements Minion {
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
  public String toString() {
    return "Banana";
  }
}
```



```
class GrusLab {
  private java.util.Map<String,Minion> prototypes = new java.util.HashMap();

  public GrusLab() {
    prototypes.put("Stuart", new Stuart());
    prototypes.put("Kevin", new Kevin());
    prototypes.put("Bob", new Bob());
  }

  public Minion makeObject(String s) throws CloneNotSupportedException {
    return (Minion) (prototypes.get(s).clone());
  }
}

public class PrototypeFactory {
  public static void main(String[] args) throws CloneNotSupportedException {
    String[] minions = {"Kevin", "Stuart", "Bob", "Stuart", "Bob", "Stuart", "Bob"};
    GrusLab lab = new GrusLab();
    for (int i = 0; i < minions.length; i++) {
      System.out.println(lab.makeObject(minions[i]));
    }
  }
}
```

- Kevin
- Stuart
- Banana
- Stuart
- Banana
- Stuart
- Banana

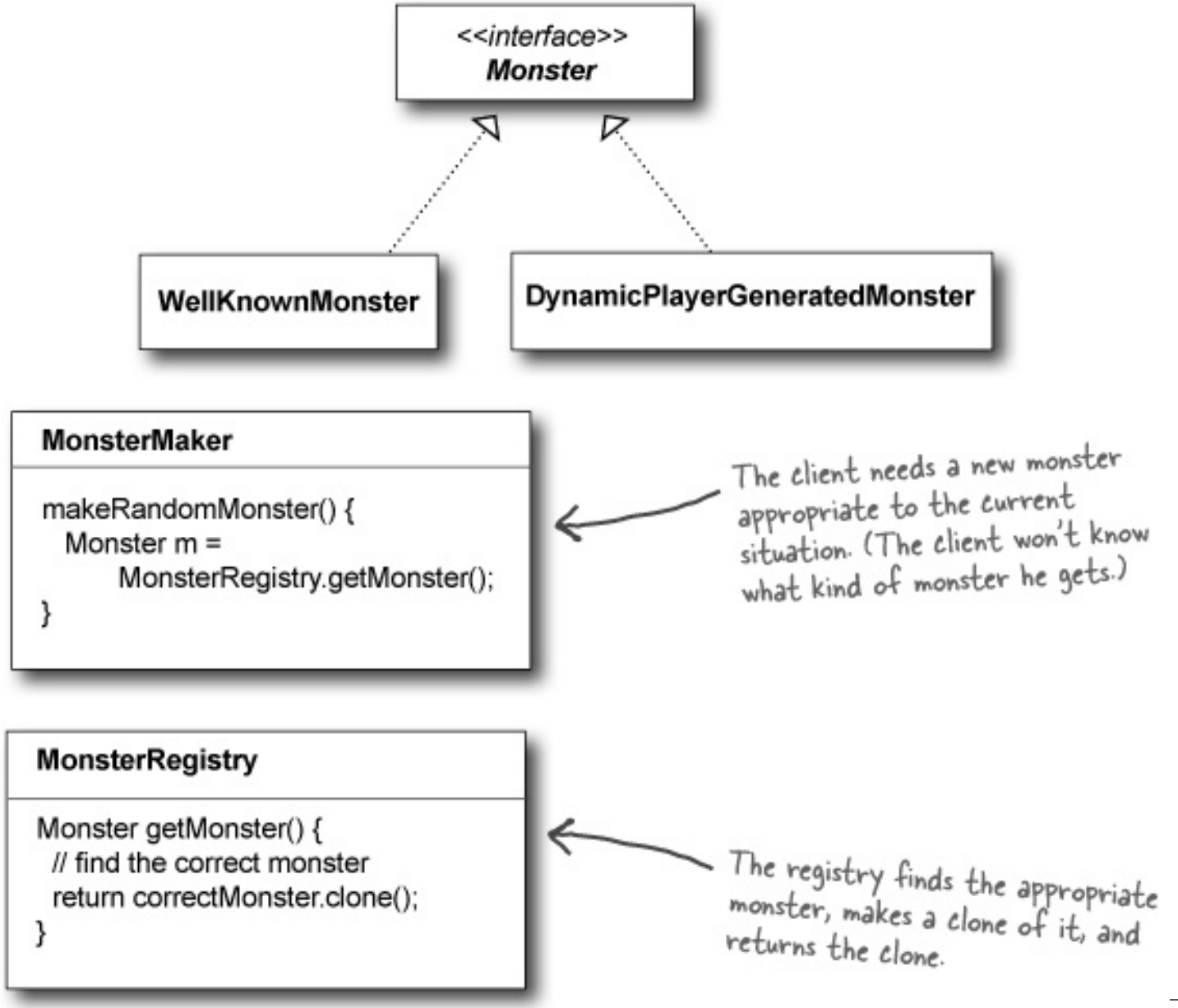
The Prototype Pattern

- Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.
- Scenario:
  - "Your interactive role playing game has an insatiable appetite for monsters. As your heroes make their journey through a dynamically created landscape, they encounter an endless chain of foes that must be subdued. You'd like the monster's characteristics to evolve with the changing landscape. It doesn't make a lot of sense for bird-like monsters to follow your characters into undersea realms. Finally, you'd like to allow advanced players to create their own custom monsters."
  - The client needs a new monster appropriate to the current situation (he does not know what kind of monster he gets).

So what's the problem?

- It's best to decouple the code that handles the details of creating the monsters from the code that actually needs to create them on the fly
  - Putting complicated combinations of state variables into constructors can be tricky
- The Prototype Pattern allows you to make new instances by copying existing instances
  - in Java this typically means using the clone() method, or de-serialization when you need deep copies
  - the client code can make new instances without knowing which specific class is being instantiated

A Prototype Pattern Example



Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.