Object Design with GoF Patterns, continued

Curt Clifton

Rose-Hulman Institute of Technology

Applying Patterns to NextGen POS Iteration 3

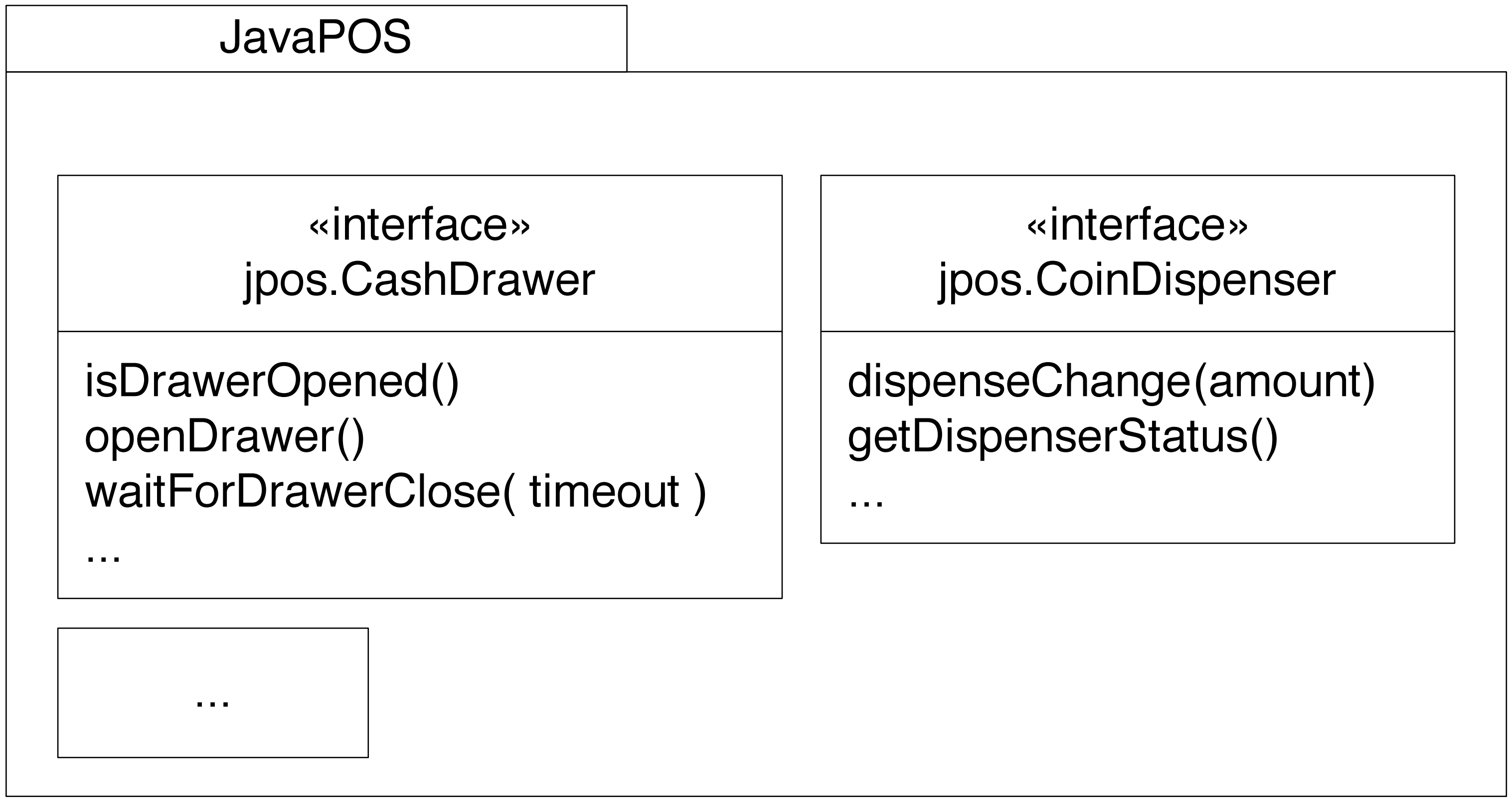
- Local caching
  - Used Adapter and Factory
- Failover to local services
  - Used Proxy, Adapter, and Factory
- Support for third-party POS devices
- Handling payments

Accessing External Physical Devices

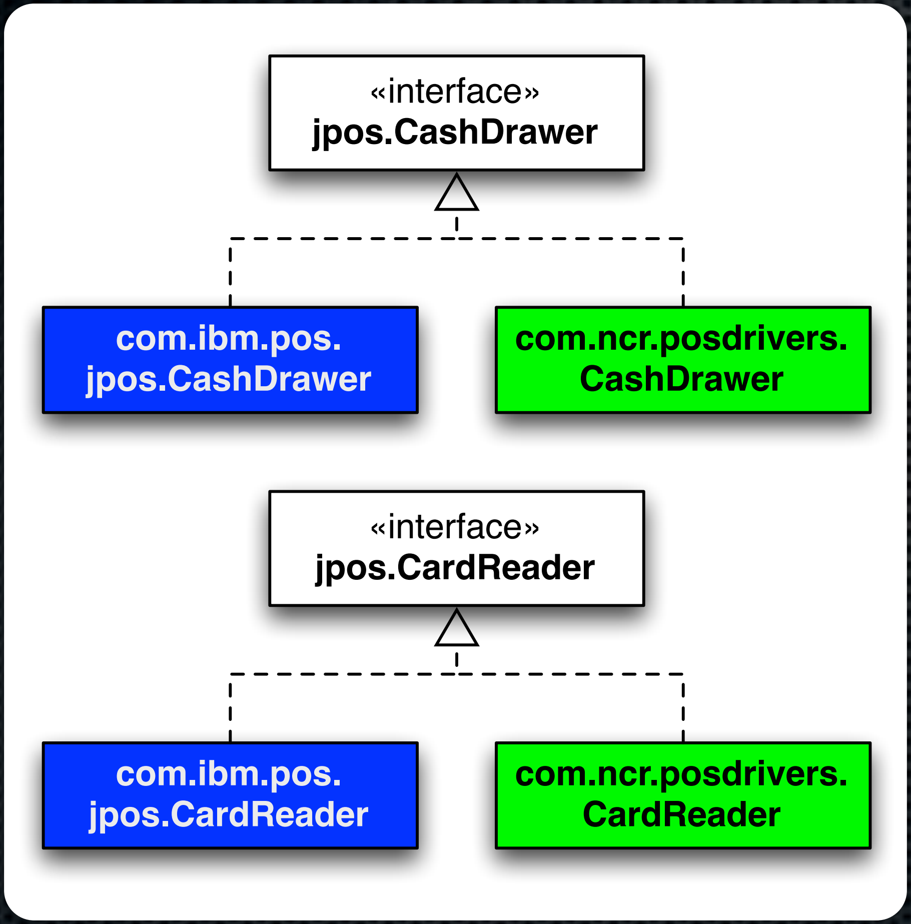
- Some physical POS devices:
  - Cash drawer, coin dispenser, digital signature pad, card reader
- NextGen POS must work with devices from a variety of vendors
- UnifiedPOS is an industry standard OO interface
  - JavaPOS provides a Java mapping as a set of Java interfaces

Architect would document decision to use these in a technical memo

Sample JavaPOS Interfaces

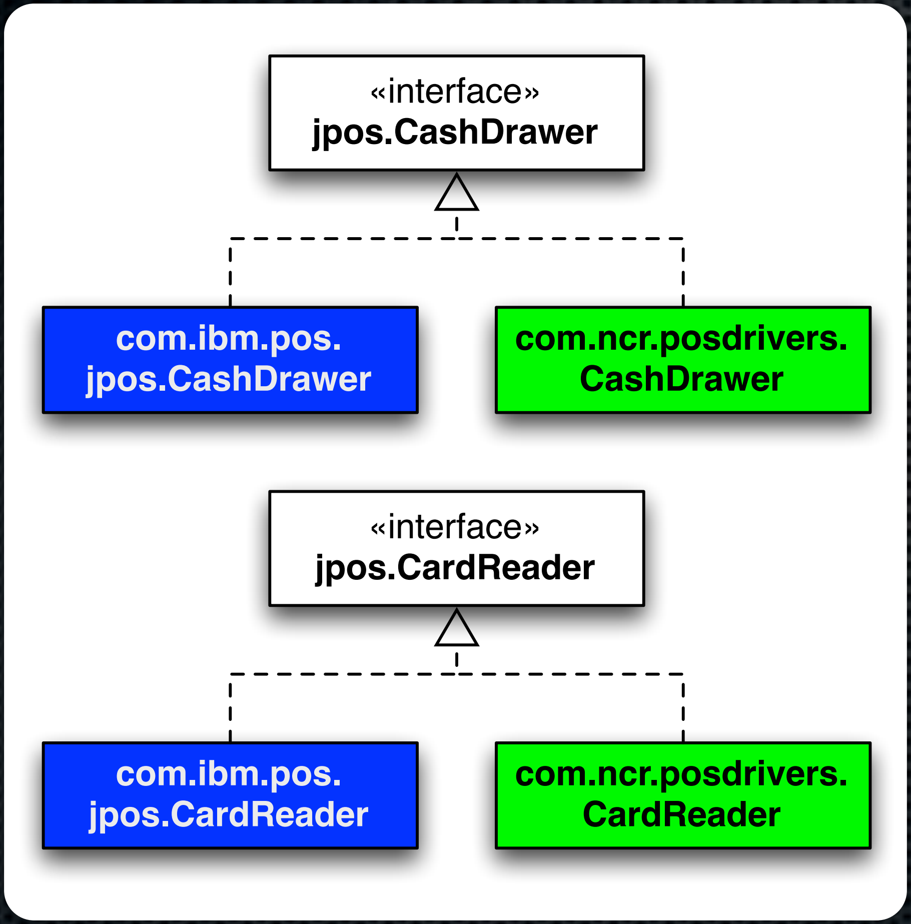


Equipment Manufacturers Provide Implementations



- Manufacturer provides:
  - Device driver for hardware
  - Java class implementing JavaPOS interface
- Class uses Java Native Interface to talk to device driver

What does this mean for NextGen POS?



- What types does NextGen POS use to communicate with external devices?
- How does NextGen POS get the appropriate instances?

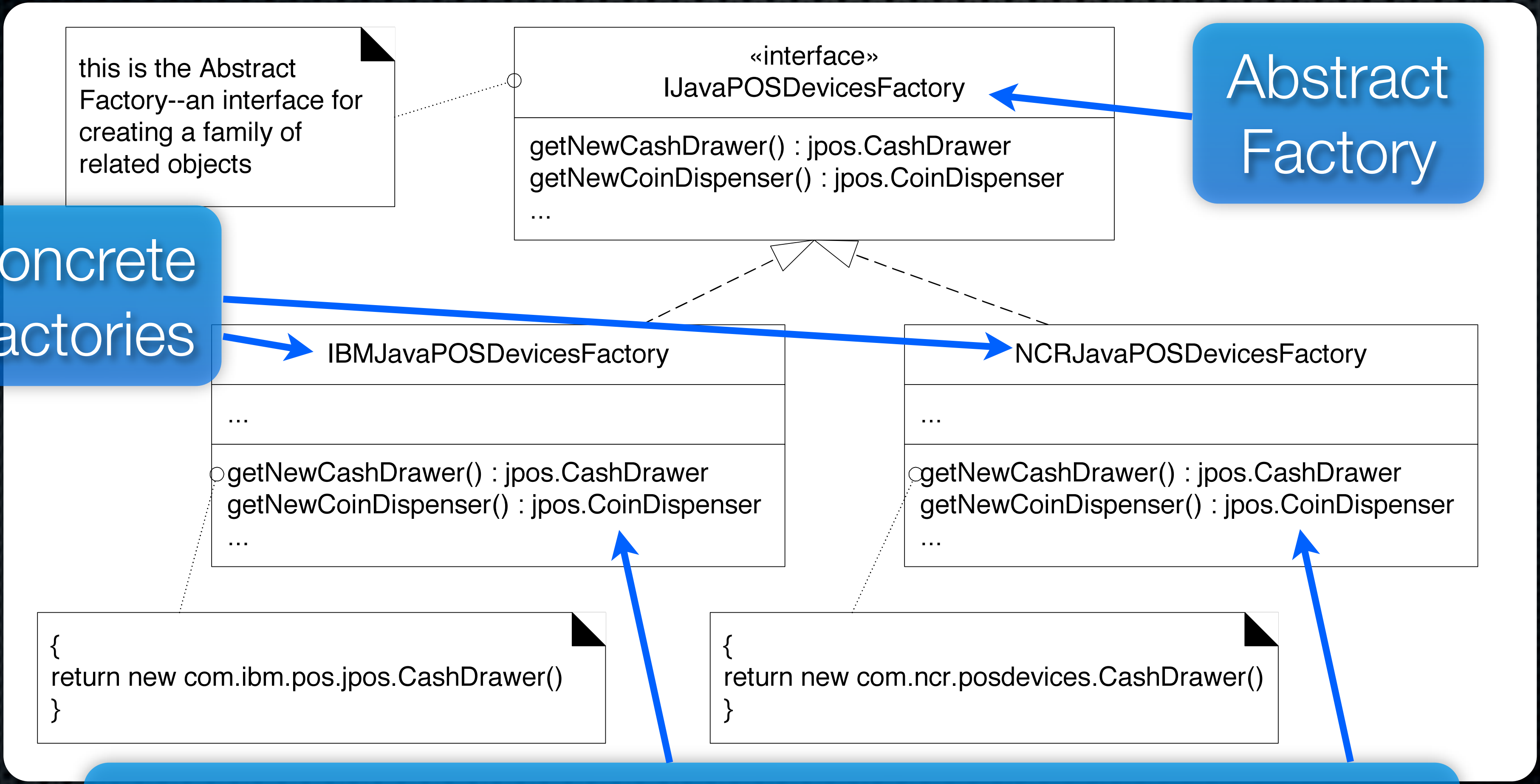
Assume: A given store uses a single manufacturer

Abstract Factory

- **Problem**: How can we create families of related classes while preserving the variation point of switching between families?
- **Solution**: Define an *abstract factory* interface. Define a *concrete factory* for each family.
- Example…

Q1,2

Abstract Factory Example


this is the Abstract Factory--an interface for creating a family of related objects
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
Abstract Factory
Concrete Factories
IBMJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
NCRJavaPOSDevicesFactory
...
getNewCashDrawer() : jpos.CashDrawer
getNewCoinDispenser() : jpos.CoinDispenser
...
{ return new com.ibm.pos.jpos.CashDrawer() }
{ return new com.ncr.posdevices.CashDrawer() }

Methods create vendor-specific instances, but use standard interface types.

First Attempt at Using Abstract Factory

```
class Register {
    ...
    public Register() {
        IJavaPOSDevicesFactory factory =
            new IBMJavaPOSDevicesFactory();
        this.cashDrawer =
            factory.getNewCashDrawer();
        ...
    }
}
```

Constructs a vendor-specific concrete factory

Uses it to construct device instances

What if we want to change vendors? Can we do better?

First Attempt at Using Abstract Factory

```
class Register {
    ...
    public Register() {
        IJavaPOSDevicesFactory factory =
            new IBMJavaPOSDevicesFactory();
        this.cashDrawer =
            factory.getNewCashDrawer();
        ...
    }
}
```

Constructs a vendor-specific concrete factory

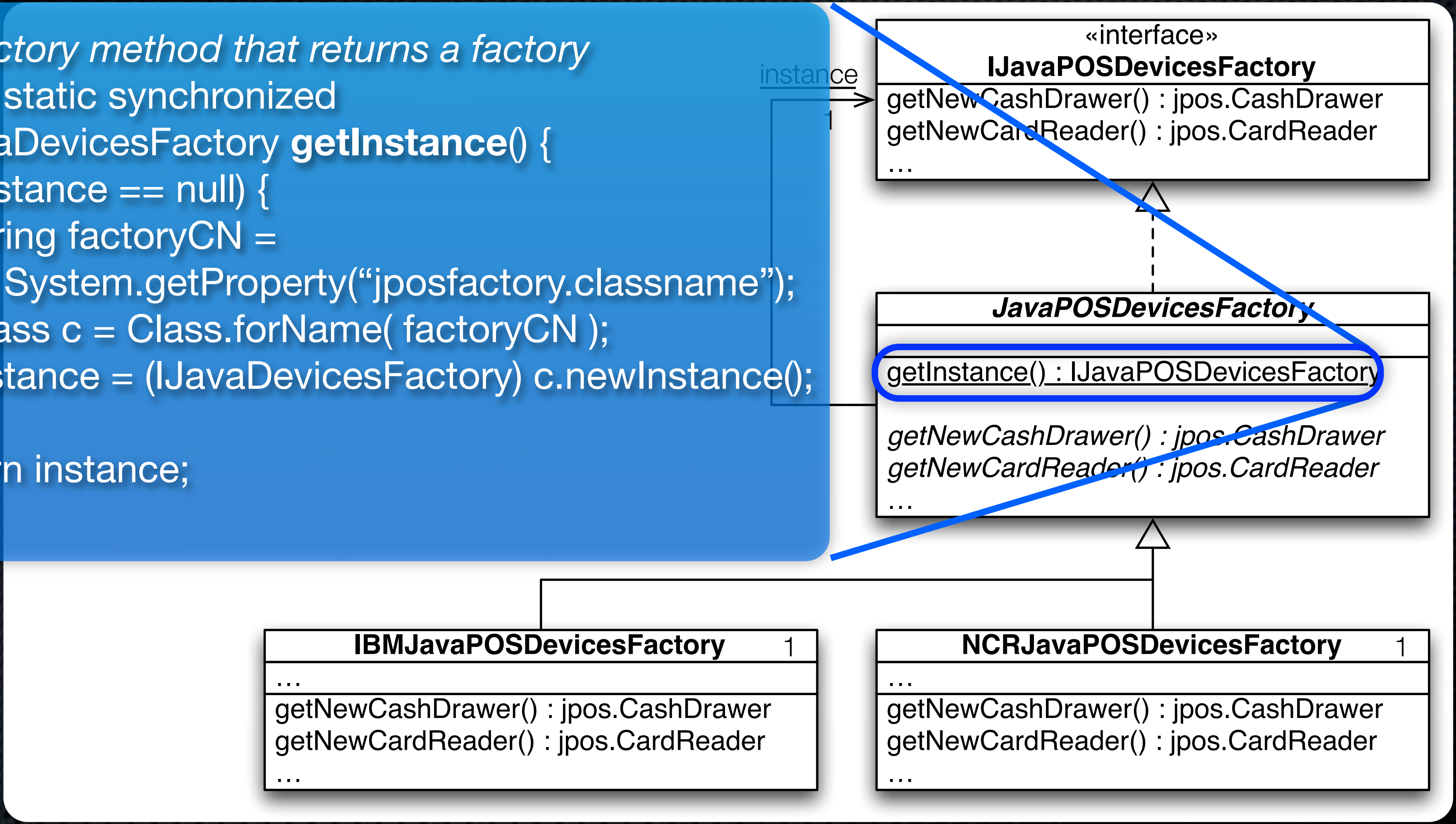
Uses it to construct device instances

Use a Factory Factory!

What if we want to change vendors? Can we do better?

Using a Factory Factory

```
// A factory method that returns a factory
public static synchronized
   IJavaDevicesFactory getInstance() {
   if (instance == null) {
      String factoryCN =
           System.getProperty("jposfactory.classname");
      Class c = Class.forName( factoryCN );
      instance = (IJavaDevicesFactory) c.newInstance();
   }
   return instance;
}
```



instance
1
«interface»
IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCardReader() : jpos.CardReader
…
JavaPOSDevicesFactory
getInstance() : IJavaPOSDevicesFactory
getNewCashDrawer() : jpos.CashDrawer
getNewCardReader() : jpos.CardReader
…
IBMJavaPOSDevicesFactory
1
…
getNewCashDrawer() : jpos.CashDrawer
getNewCardReader() : jpos.CardReader
…
NCRJavaPOSDevicesFactory
1
…
getNewCashDrawer() : jpos.CashDrawer
getNewCardReader() : jpos.CardReader
…

Using a Factory Factory

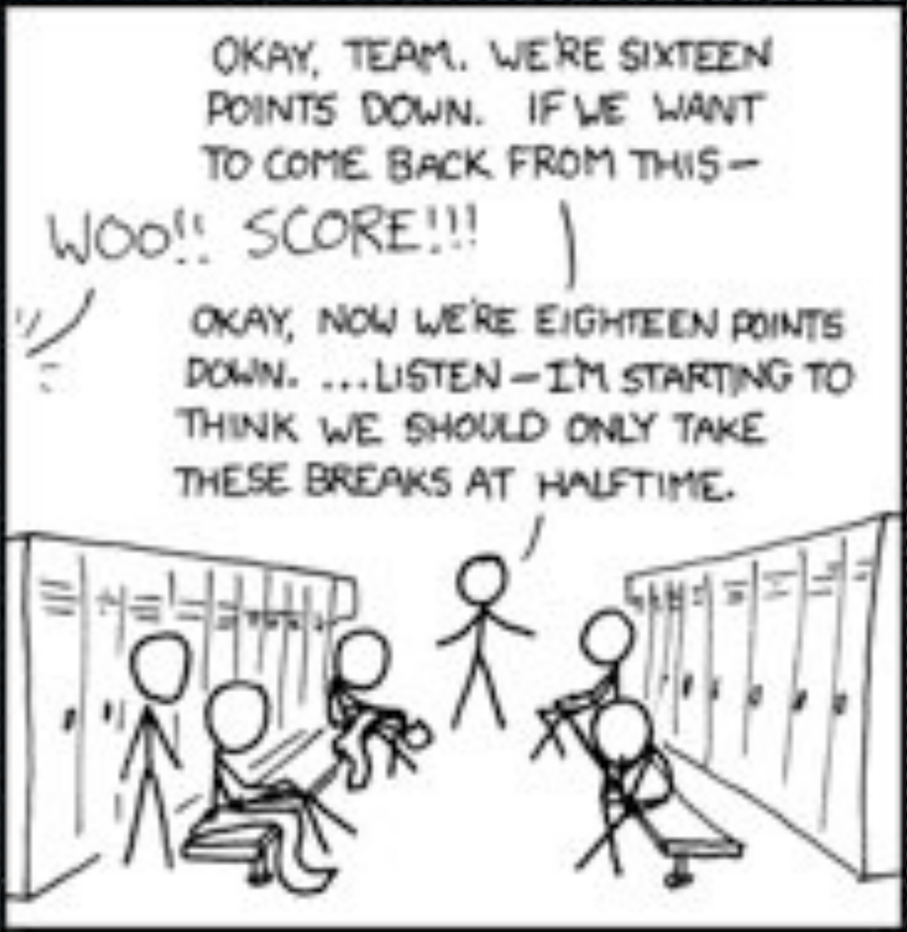
```
class Register {
    ...
    public Register() {
        IJavaPOSDevicesFactory factory =
            JavaPOSDevicesFactory.getInstance();
        this.cashDrawer =
            factory.getNewCashDrawer();
        ...
    }
}
```

Gets a vendor-specific concrete factory singleton

Uses it to construct device instances

Q3

Pep Talk


OKAY, TEAM. WE'RE SIXTEEN POINTS DOWN. IF WE WANT TO COME BACK FROM THIS—
WOO!! SCORE!!!
OKAY, NOW WE'RE EIGHTEEN POINTS DOWN. ...LISTEN—I'M STARTING TO THINK WE SHOULD ONLY TAKE THESE BREAKS AT HALFTIME.

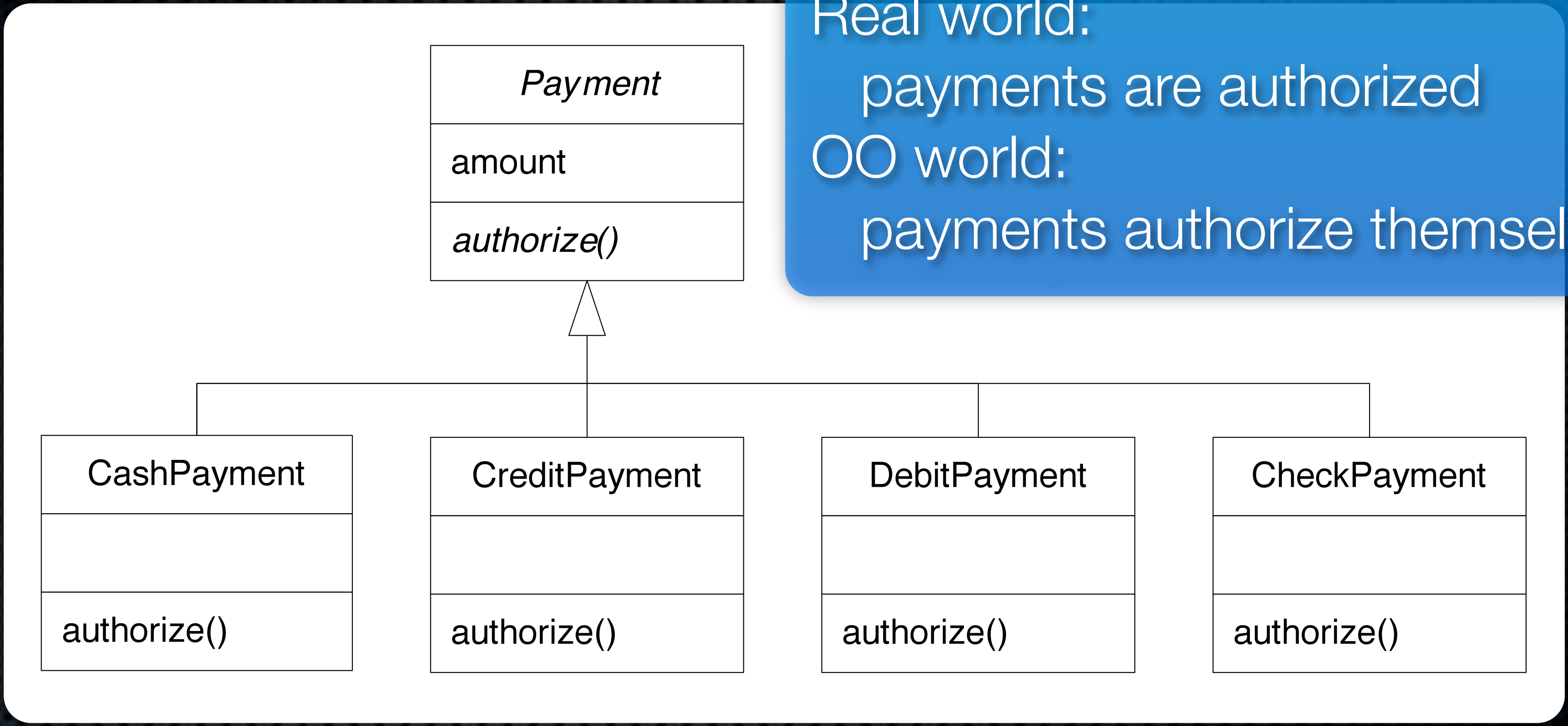
http://xkcd.com/544/

Listen! They said a team of chess players coached by someone with no understanding of basketball would never be competitive in the NBA! Well, it turns out they're pretty perceptive.

Handling Payments

- Follow the “Do It Myself” Guideline:
  - “As a software object, **I do** those things that are normally **done to** the actual object I represent.”
- A common way to apply Polymorphism and Information Expert
- Example…

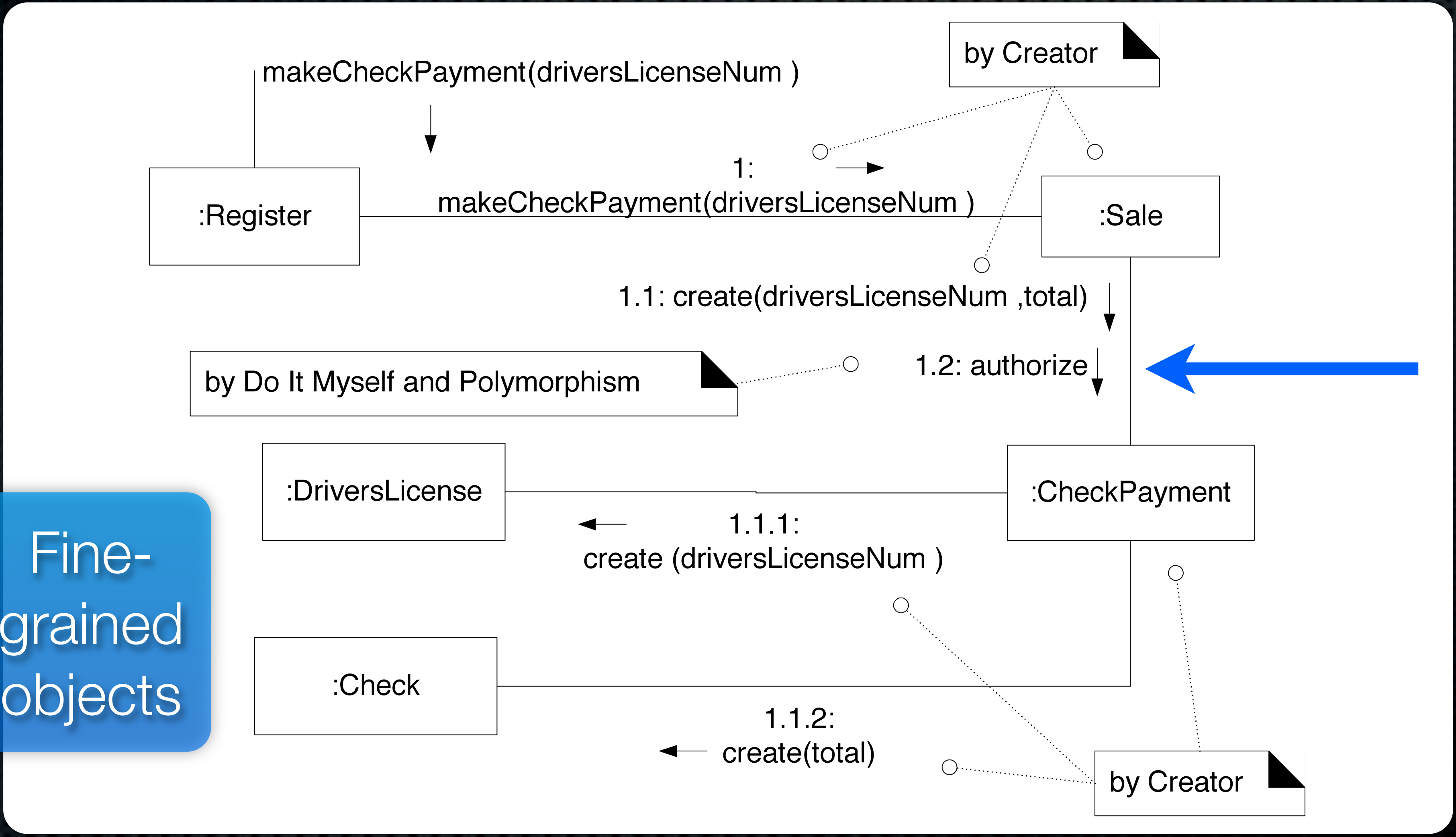
"Do It Myself" Example



Real world:
payments are authorized
OO world:
payments authorize themselves

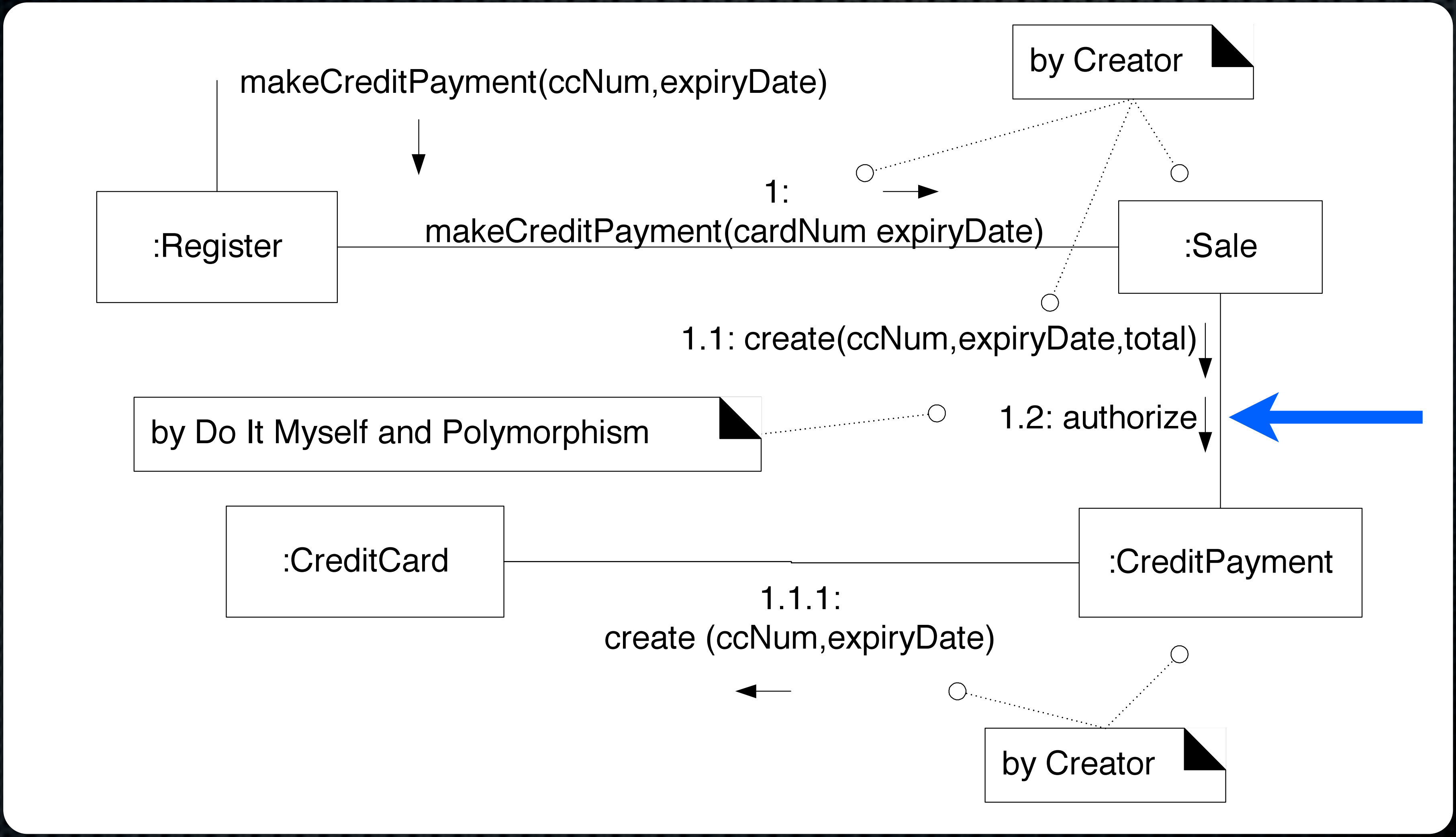
Q4

Creating a CheckPayment


makeCheckPayment(driversLicenseNum)
by Creator
:Register
1:
makeCheckPayment(driversLicenseNum)
:Sale
1.1: create(driversLicenseNum ,total)
by Do It Myself and Polymorphism
1.2: authorize
:DriversLicense
:CheckPayment
1.1.1:
create (driversLicenseNum)
:Check
1.1.2:
create(total)
by Creator

Fine-grained objects

Creating a CreditPayment


by Creator
makeCreditPayment(ccNum,expiryDate)
:Register
1:
makeCreditPayment(cardNum expiryDate)
:Sale
1.1: create(ccNum,expiryDate,total)
by Do It Myself and Polymorphism
1.2: authorize
:CreditCard
:CreditPayment
1.1.1:
create (ccNum,expiryDate)
by Creator

Frameworks and Patterns

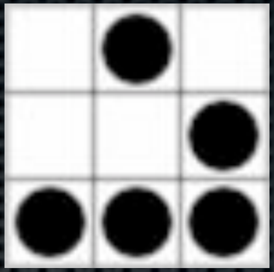
Framework

- An extendable set of objects for related functions
- Examples:
  - Swing GUI framework
  - Java collections framework
  - Hibernate persistence framework

Frameworks Typically

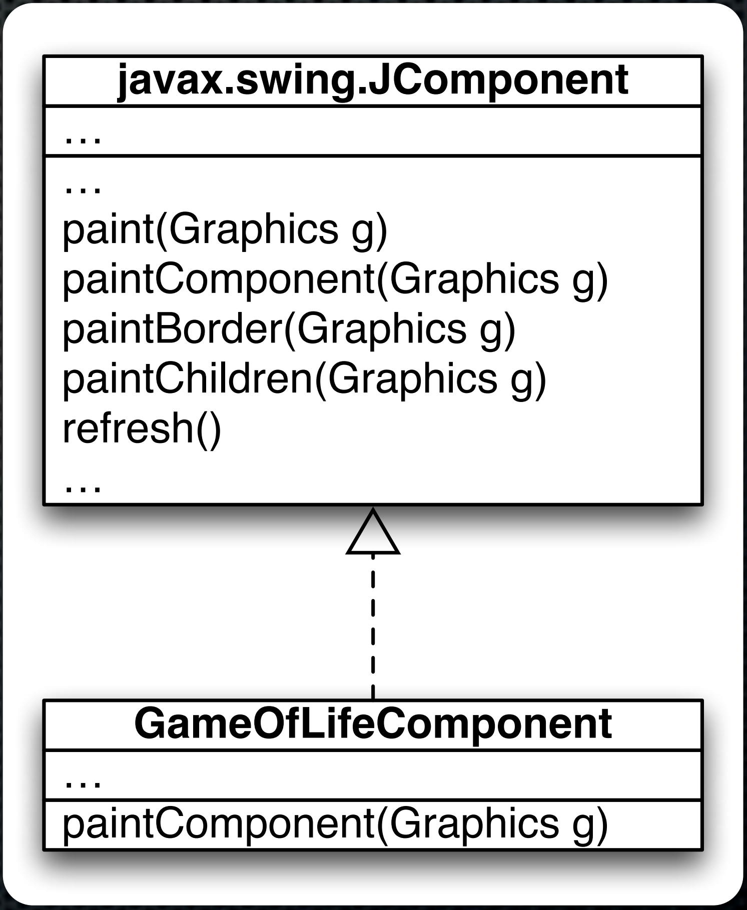
- Provide a cohesive set of interfaces and classes
  - Capture the unvarying parts
  - Provide extension points to handle variation
- Used by extending provided classes
- Rely on the **Hollywood Principle**:
  - “Don’t call us, we’ll call you.”

Hollywood Principle in Action



- Consider creating a UI for Conway's Game of Life…
- We inherit a metric ton of stuff from the framework
- We override one method
- We never call that method!

"Don't call us, we'll call you."



Template Method Pattern

- **Problem**: How can we record the basic outline of an algorithm in a framework (or other) class, while allowing extensions to vary the specific behavior?
- **Solution**: Create a *template method* for the algorithm that calls (often abstract) *hook methods* for the steps. Subclasses can override/implement these hook methods to vary the behavior.
- Example…

Q5,6

Template Method Example

- In JComponent:

```
public void paint(Graphics g) {
        paintComponent(g);
        paintBorder(g);
        paintChildren(g);
}
public void paintComponent(Graphics g) { /* empty */ }
public void paintBorder(Graphics g) { /* empty */ }
public void paintChildren(Graphics g) { /* empty */ }
```

Template Method

Hook Methods

Template Methods in Your Designs

- **Bad code smell**: polymorphic methods in related subclasses are copied and pasted with minor differences
- **Solution**: use the Template Method pattern
  - Refactor the differences into helper methods (hooks)
  - Add abstract hook methods to the superclass
  - Pull the common code up to a template method in the superclass

Design Studio: Log File Parser

Team describes problem and perhaps current solution (if any)	~5 min.
Class thinks about questions, alternative approaches. **Q7**	~3 min.
On-board design	~12 min.