Swinging on the Bleeding Edge



Some Java Best Practices

Dr. Erich Gamma
Object Technology International

erich_gamma@oti.com

Introduction

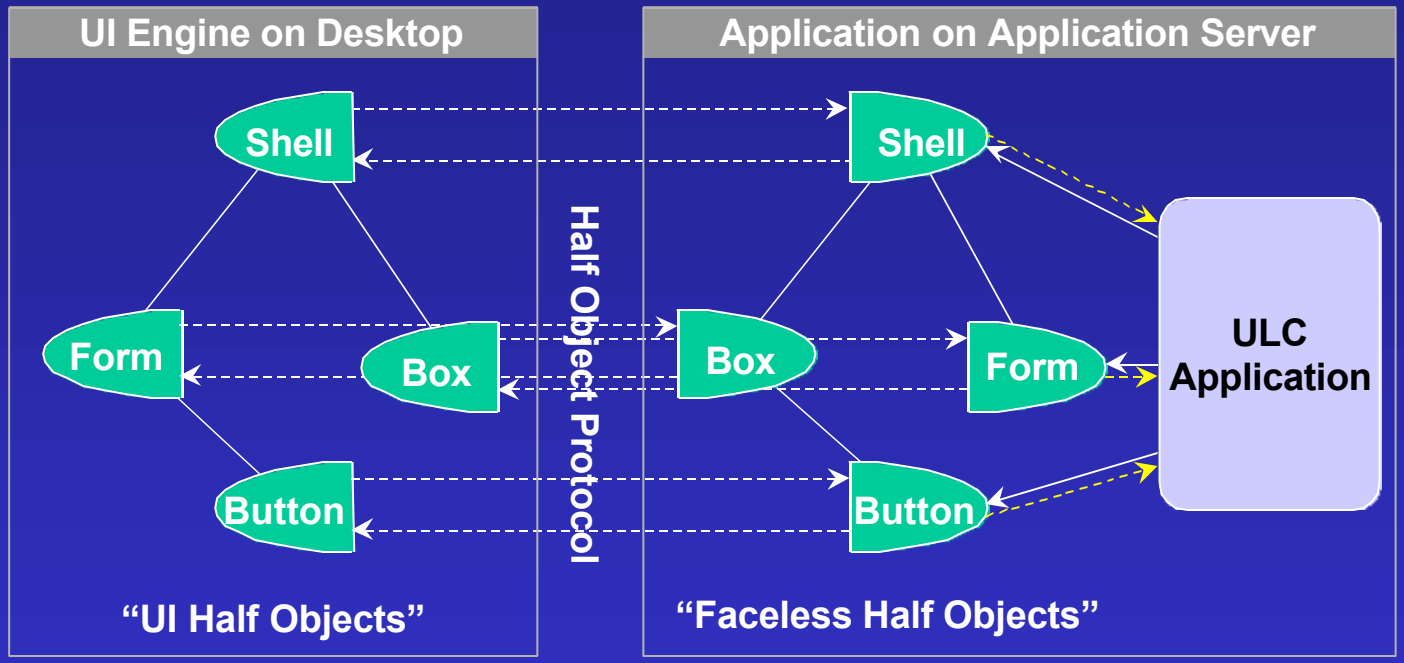
- Software development is still hard!

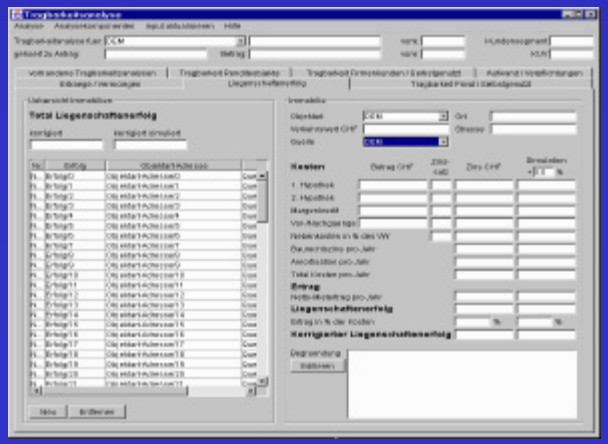
> Our enemy is complexity, and our job is to kill it.
>
> Jan Baan

- Software physics
  - younger code - more bugs
  - more code - more bugs, less performance
- Continuously evolving systems will become the norm
  - improving an architecture

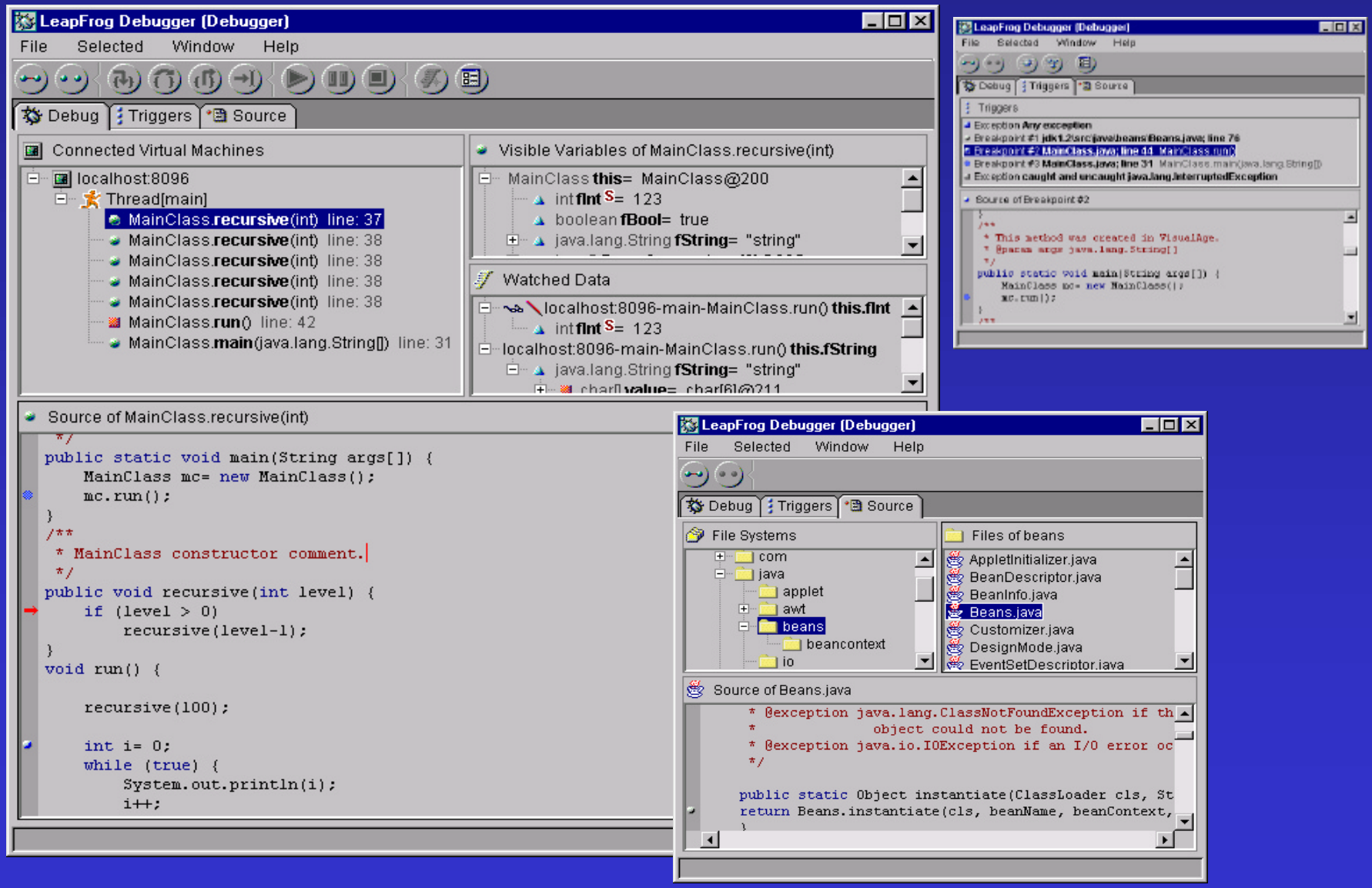
  ⇒“Make Change Your Friend”

Background: ULC - Ultra Light Client





Debugger for VA for Embedded



Overview

Best Practices

- Leveraging patterns in Java
- Refactoring
- Unit testing

Design Patterns

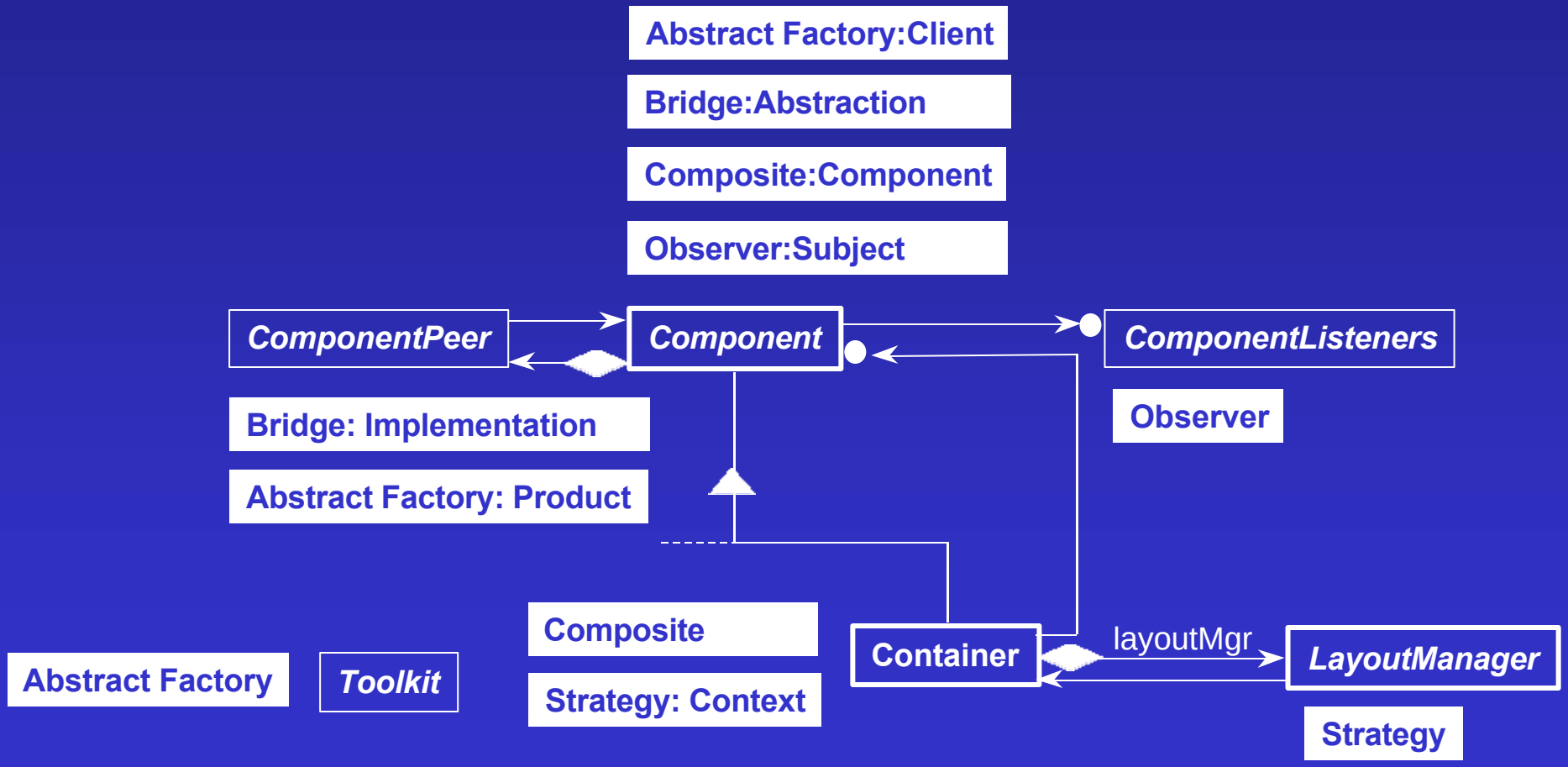
- Design patterns
  - descriptions of communicating objects that are customized to solve reoccurring design problems
- Characteristics
  - enable to repeat a successful design done by someone else
  - define a vocabulary for talking about design
    - design conversations today leverage patterns
  - design patterns are not a UML diagram with a name

Design Patterns and Java

- Java API has a high *pattern density*
  - major parts of the API design can be covered with design patterns
- Java provides advanced language features like threads
  - many degrees of freedom for how to use them
  - developers need guidance

  ⇒“Concurrent Programming in Java - Design Principles and Patterns”, Doug Lea, Addison Wesley

Pattern Density in the Java API - AWT



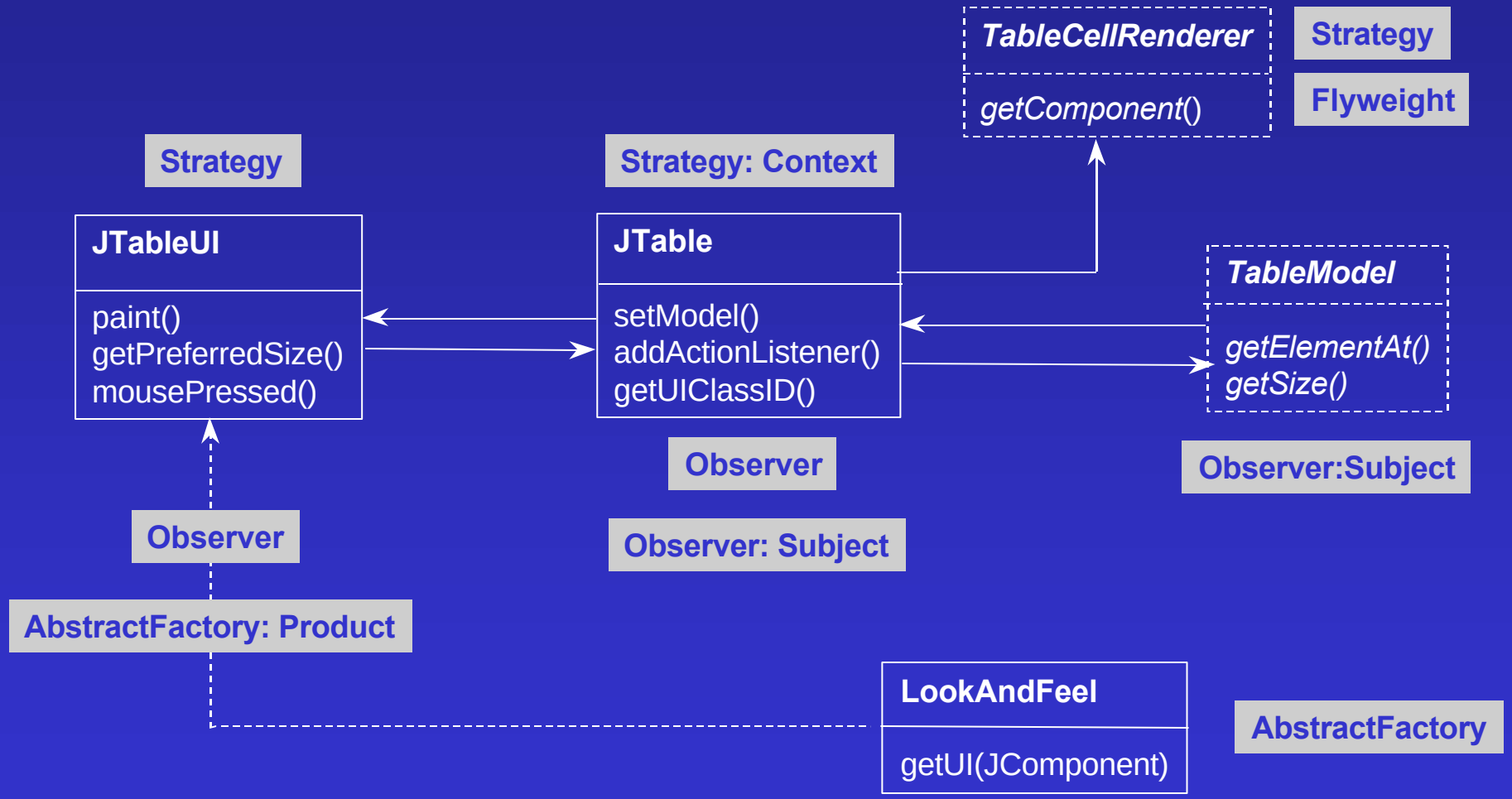
Understanding the Swing Architecture

- Swing is huge
  ⇒ > 1000 classes
- Fine granular design
  - JTable/JTableUI
  - TableModel
  - TableCellRenderer
  - TableCellEditor

  ⇒how are they related?


Schaukeln
verboten
Défense de
se balancer
Non dondolarsi
Do not swing
GARAVENTA AG

Swing from the Patterns Perspective



Understanding an Architecture

- Design patterns provide words for telling a story about a design
  - developers don't have to reinvent a higher-level language to discuss a design

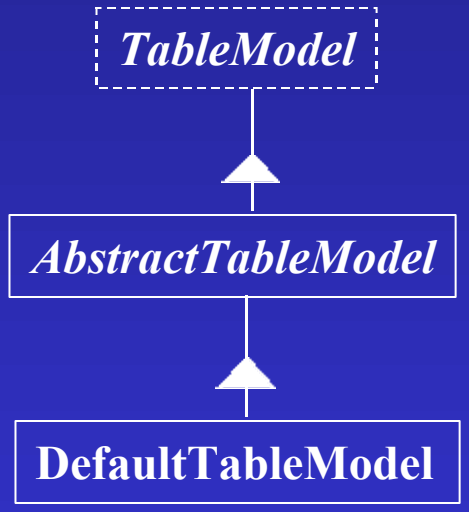
⇒provide dictionary to propose, compare, and reject designs

- Speeds-up understanding of a design
  - hooks for extension
  - design limitations

  ⇒avoids **fighting** against a design
- Complexity reduction
  - client can focus on pattern variations

Pattern Discovery in Swing - Interface/Implementation Pair

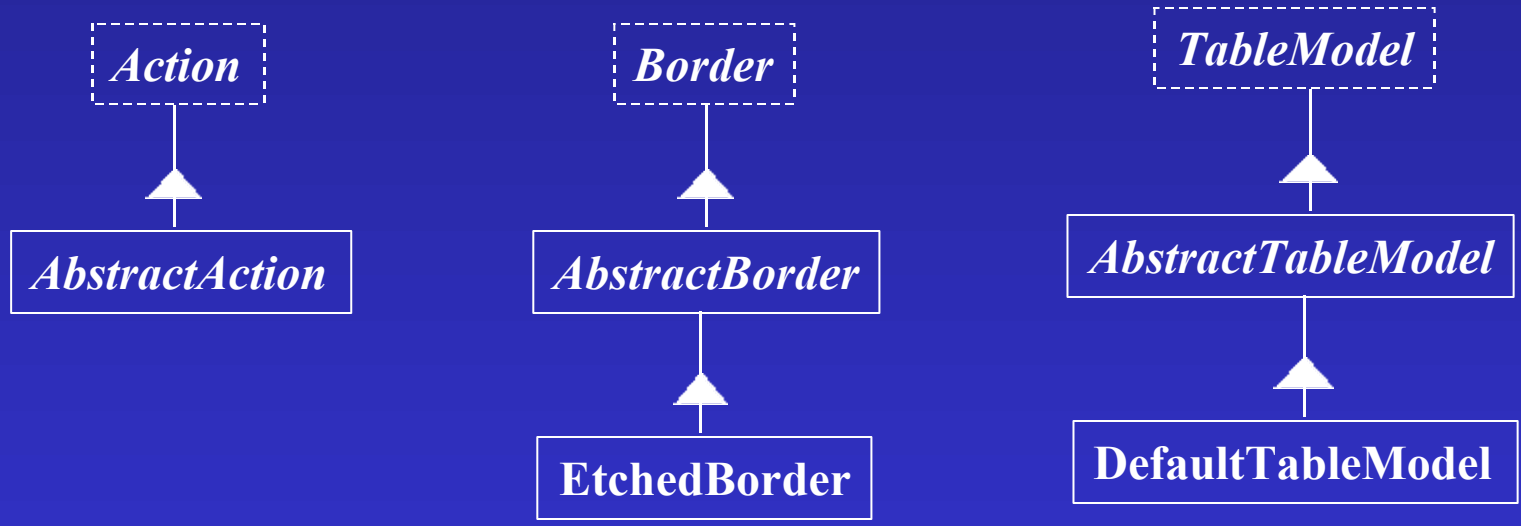
- Interface implementation pair is a recurring pattern in the Swing API
- Pattern
  - interface defines API (TableModel)
  - abstract class (AbstractTableModel) provides default implementation
  - concrete class provides out of the box usable implementation (class DefaultTableModel)



Consequences of Interface-Implementation Pair

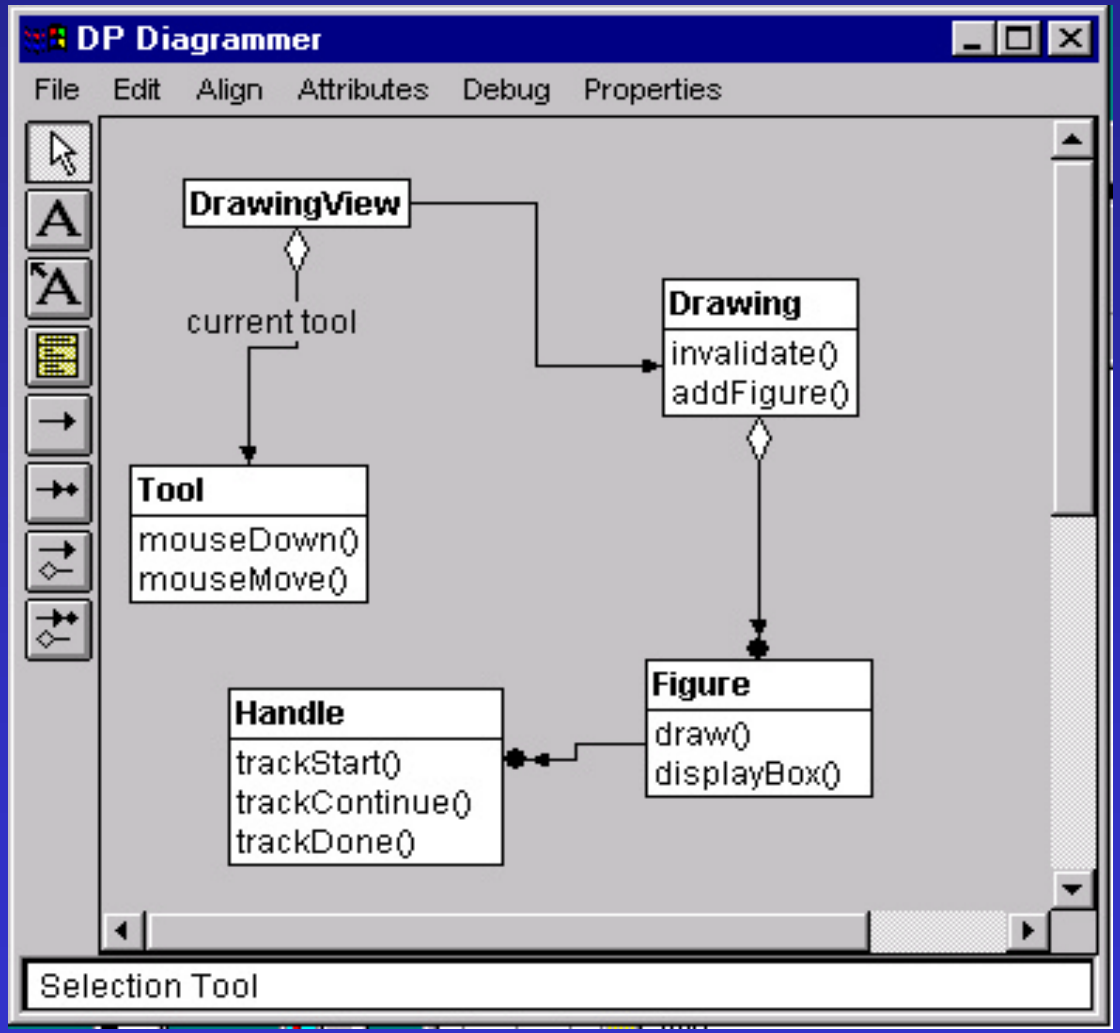
- ✔ Explicit separation of design (interface) from implementation
  - ⇒interfaces cannot have default a implementation
- ✔ Provides insulation layer on top of interface
  - ⇒every interface change is a breaking change!
  - ⇒abstract class can provide compatibility implementation
- ✘ Adds more types to a system
  - ⇒use it for key abstractions only

Known Uses


Action
AbstractAction
Border
AbstractBorder
EtchedBorder
TableModel
AbstractTableModel
DefaultTableModel

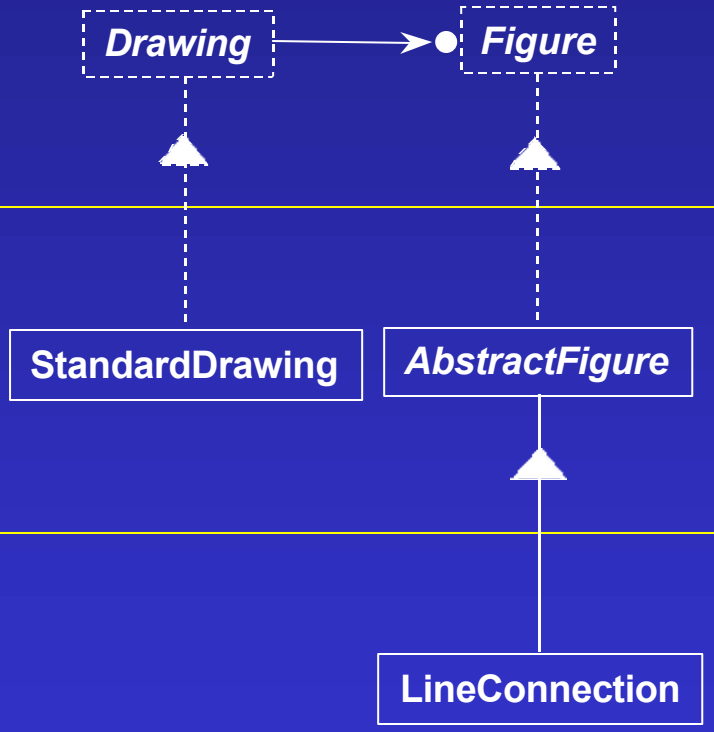
More Known Uses: JHotDraw*

- JHotDraw a framework for direct manipulation graphics editors



*http://members.pingnet.ch/gamma/JHD-5.1.zip

Layering a Framework with Interface-Implementation



Framework Package
- reuse of design
- stable
  - → no implementation
- specifies public interfaces

Defaults/Standard Package
- reuse of design and code
- less stable
  - → more implementation
- specifies protected interfaces

Kit Packages
- reuse of implementation
- least stable
  - → implementation only

more concrete

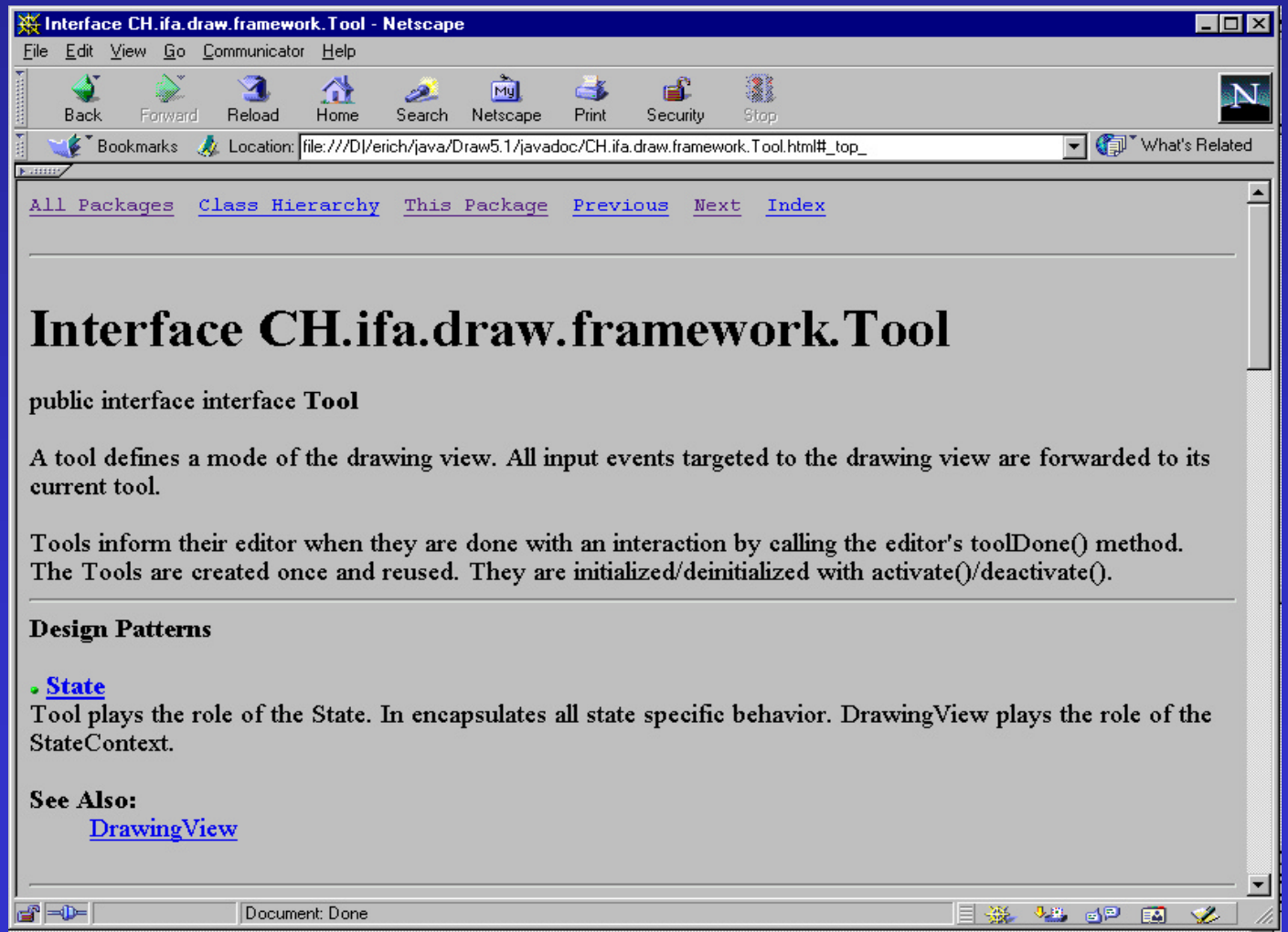
more stable

Documenting the Applied Patterns

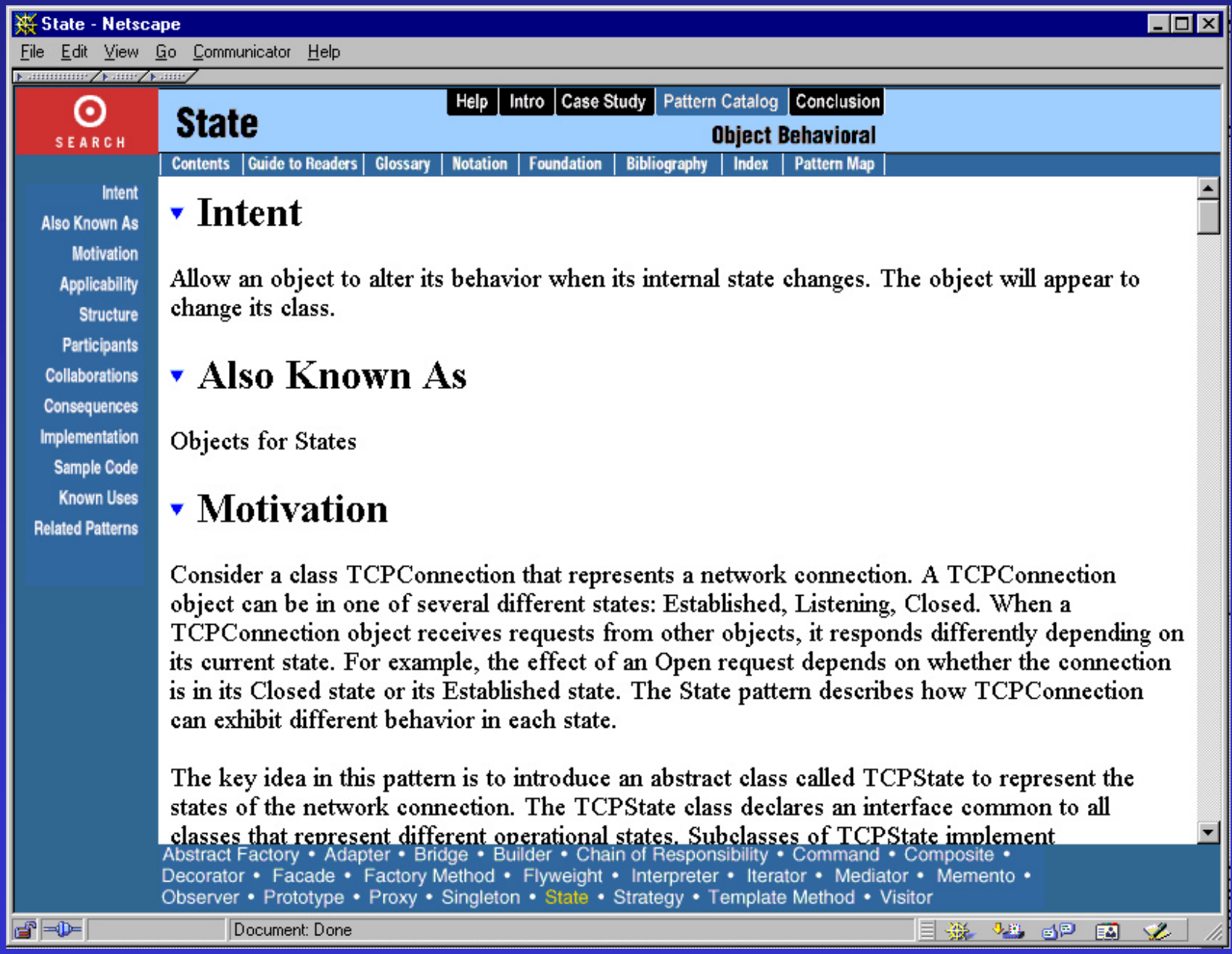
- Goal: bring design patterns closer to code
  - explain roles
  - relationships
- JavaDoc
  - comments can include URLs
  - URLs refer to a pattern description or *patlet*
  - integrated documentation helps to keep pattern annotations up to date

Using javadoc comments

```
<b>Design Patterns</b><P>
<a href=../pattlets/sld032.htm>State</a>
Tool plays the role of the State...
```



Link to Pattern Catalog



Refactoring

- Changing the the internal structure to make it easier to understand and modify
  - improve design
  - eliminate duplication - say everything “once and only once”

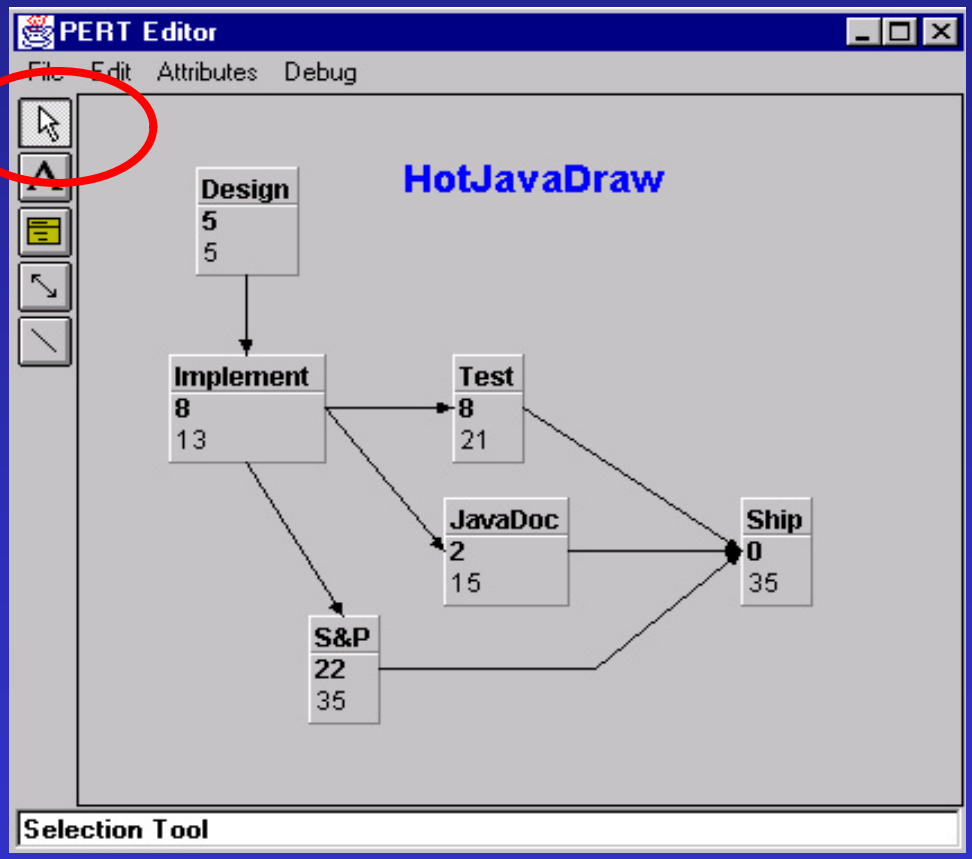
⇒“Refactoring - Improving the Design of Existing Code”, Martin Fowler, Addison Wesley

Patterns as Targets for Refactorings

- Patterns show transformations - are targets for *refactorings*
  - From conditional code to messages
    - state, strategy
  - From duplicated code to *once and only once* code
    - template method, composite
  - From coarse grained to finer grained objects

Example: JHotDraw

- The selected tool in the palette defines how input is parsed
- Challenges
  - it should be easy to define new tools
  - different editors need different tools
  - enable to mix tools differently in different editors
- 1st implementation
  - Canvas/View defines constants for different tools



1st Implementation

```
public class DrawingView  {
  public final static int SELECTION  = 0;
  public final static int RECTANGLE  = 1;

  public void mouseDown(MouseEvent e, int x, int y){

        switch (tool()) {
        case SELECTION:
           …
           break;
        case RECTANGLE:
           break;
        }
    }

  public void mouseDrag(MouseEvent e, int x, int y){
        switch (tool()) {
        case SELECTION:
           …
           break;
        case RECTANGLE:
           break;
        }
    }
```

⇒Problem

- addition of a new tool requires modification in several places

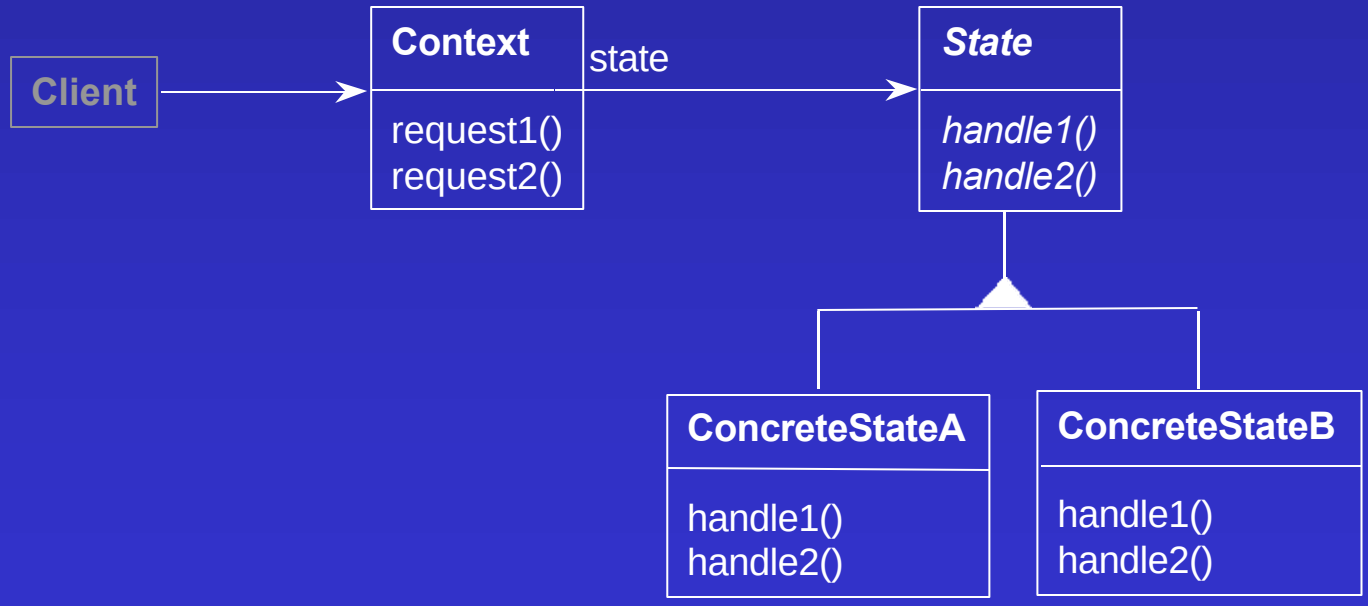
⇒refactor code and apply the State pattern

State Pattern

- Intent

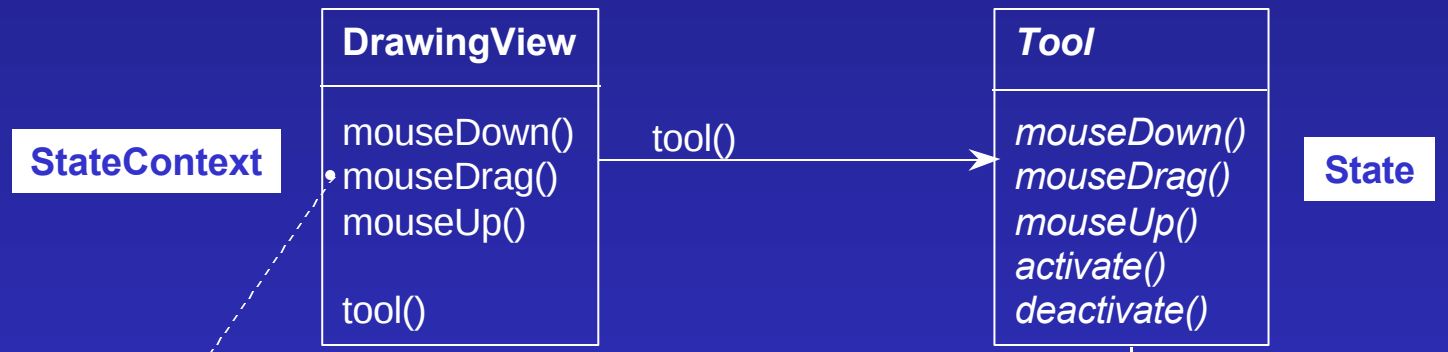
  Allow an object to alter its behavior when its internal state changes

- Applicability
  - an object's behavior depends on its state
  - operations have conditional statements that depend on the object's state
- Structure

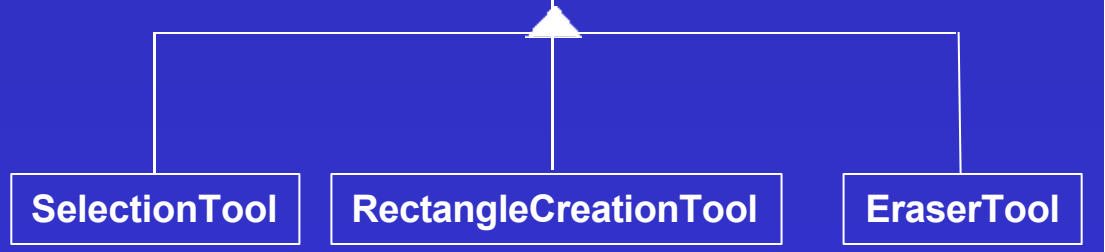


Tools as States

- Mapping the State participants to Tools




```
public void mouseDrag(MouseEvent e, int x, int y){
  tool().mouseDrag(e,x,y);
}
public void mouseDown(MouseEvent e, int x, int y){
  tool().mouseDrag(e,x,y);
}
```



Patterns as Targets for Refactorings

- Principle: *add* and not *modify* existing code
  - separate changeable from stable code
  - separate concepts with different change rates

⇒delegate control to enable variations

  - algorithms
    - Strategy
  - implementations
    - Bridge
  - instantiated classes
    - Abstract Factory, Factory Method
  - state specific behavior
    - State

Pattern Hit List

- Patterns “top ten”
  - State, Strategy (pluggable behavior)
  - Composite
  - Factory
  - Template method
  - Adapter
  - Observer
  - Mediator
  - Interface/Implementation pair
  - Command
  - Chain of Responsibility
- Patterns “bottom ten”
  - Singleton
  - Visitor

Massive Unit Testing

- “I'm in too much in a hurry.”
  - more pressure less tests more stress
- Ad hoc testing is not sufficient

  ⇒non-repeatable tests: working now isn’t sufficient

  ⇒scroll-blindness

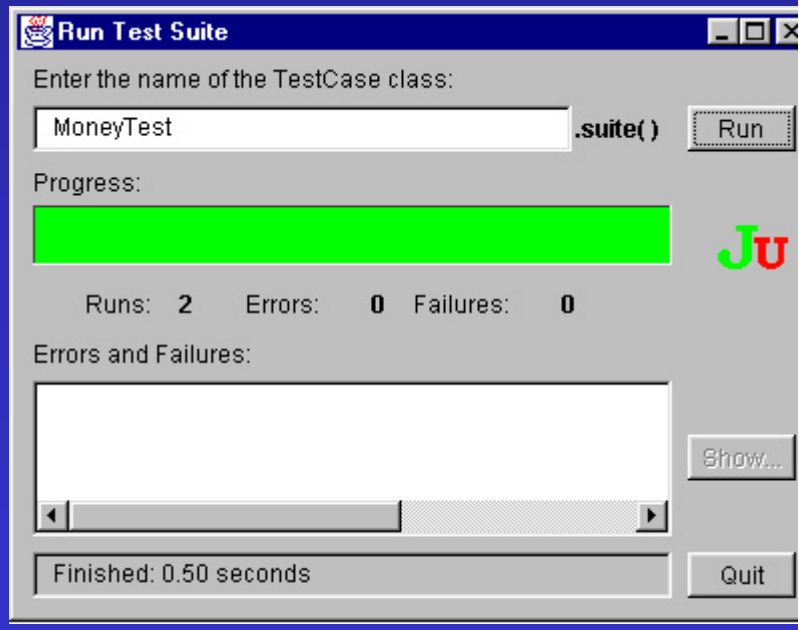
  ⇒cannot compose with tests from others
- Goal - *self testing software*
  - make writing unit tests as simple as possible
  - write tests programmatically
    - *“self-testing software”*

⇒Refactoring requires massive unit tests

JUnit*

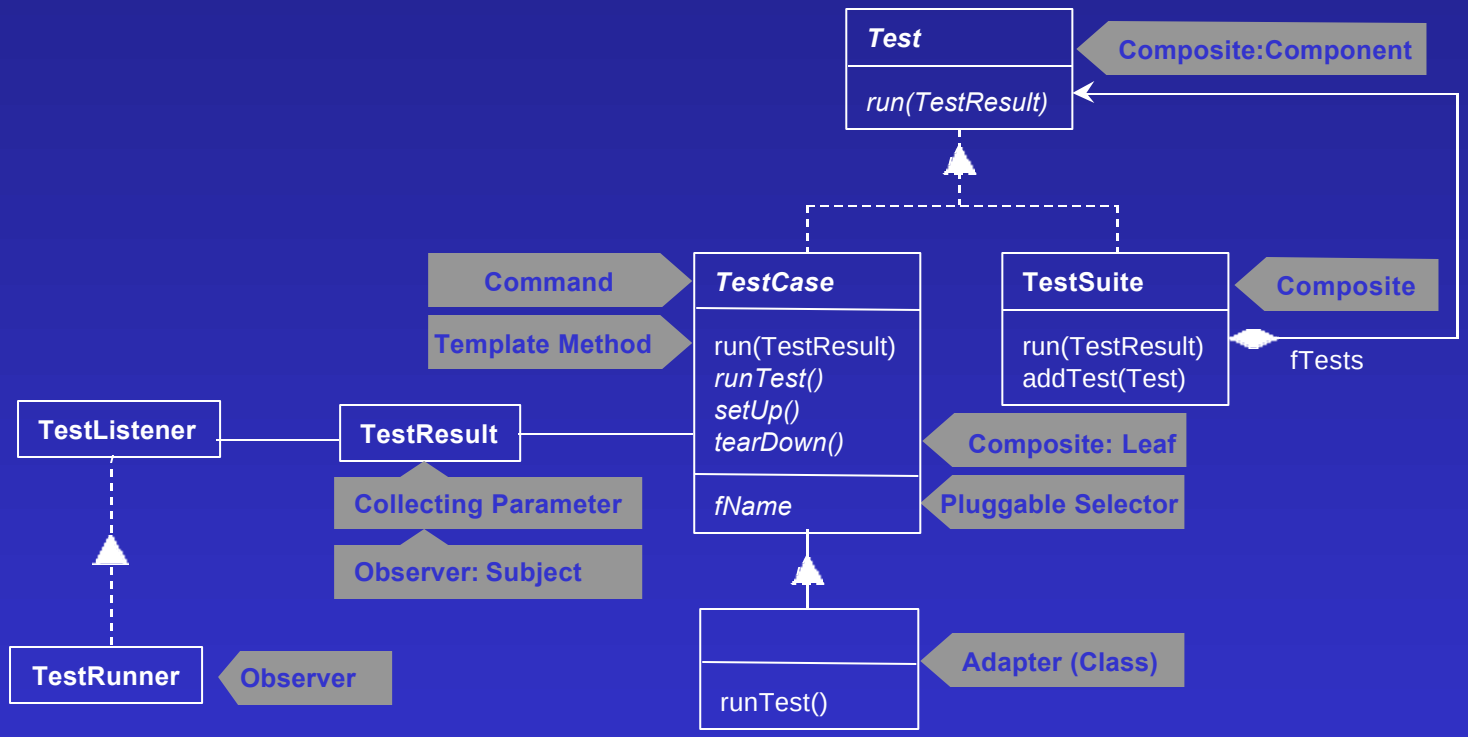
- JUnit a minimal unit testing framework
  - makes writing unit tests as simple as adding debug output to a program
  - "stimulate a fixture and verify the results"
  - allows to aggregate tests into test suites
  - run them with a simple tool

```
public class MoneyTest extends TestCase {
  public void testMoneyEquals() {
       assert(!f12CHF.equals(null));
       assert(f12CHF.equals(f12CHF));
       assert(!f12CHF.equals(f14CHF));
  }
  public void testAdd() {
    …
  }
}
```



* ftp://www.armaties.com/D/home/armaties/ftp/TestingFramework/JUnit/JUNIT21.ZIP

Architecture of JUnit was derived with Patterns



⇒Pattern density around key abstraction TestCase

Conclusions

- Leverage Design Patterns
  - Design patterns and Java are a powerful combination

  ⇒...but no language can reduce the importance of good design
- Continuously refactor and improve the code
- Do massive unit testing

⇒see also “Extreme Programming XP”, Kent Beck, Addison Wesley