Software Engineering

Lecture 06 – Design Patterns: Introduction

© 2015-20 Dr. Florian Echtler
Bauhaus-Universität Weimar
<florian.echtler@uni-weimar.de>

Design Patterns

- “Recipes” for common problems
- Available on different levels (e.g. system architecture, user interface, implementation)
- Usually without any code, at most UML
- Often just “box-and-line”-diagrams, esp. for architectural patterns

Description of a Design Pattern

- Problem
  - Motivation/application area
- Solution
  - Structure (class diagram)
  - Pieces (names of classes or operations)
  - Object interaction (e.g. sequence diagram)
- Discussion
  - Advantages, disadvantages, dependencies
  - Constraints, special cases, known uses

Design Patterns: History

- Originally from architecture and construction
- “A Pattern Language: Towns, Buildings, Construction”, Christopher Alexander, 1977
  - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

“A Pattern Language” examples

Image source (FU): “A Pattern Language”, C. Alexander

141 A Room of One's Own

May be part of Intimacy Gradient (127), The Family (75), House for a Small Family (76), Common Areas at the Heart (129).

Conflict

No one can be close to others, without also having frequent opportunities to be alone.

Resolution

Give each member of the family a room of his own, especially adults. A minimum room of one's own is an alcove with desk, shelves, and curtain. The maximum is a cottage- like a Teenagers Cottage(154), or an old age cottage(155). In all cases, especially the adult ones, place these rooms at far ends of the intimacy gradient- far from the common rooms.

22 Nine Percent Parking

May be part of Local Transport Areas (11), Community of 7000 (12), Identifiable Neighbourhood (14).

Conflict

Very simply- when the area devoted to parking is too great, it destroys the land.

Resolution

Do not allow more that 9% of the land in any given area to be used for parking. In order to prevent the bunching of parking in huge neglected areas, it is necessary for a town or a community to subdivide its land into parking zones no larger than 10 acres each and to apply the same rule in each zone.

May contain Shielded Parking (97), Small Parking Lots (103).

Architecture Patterns

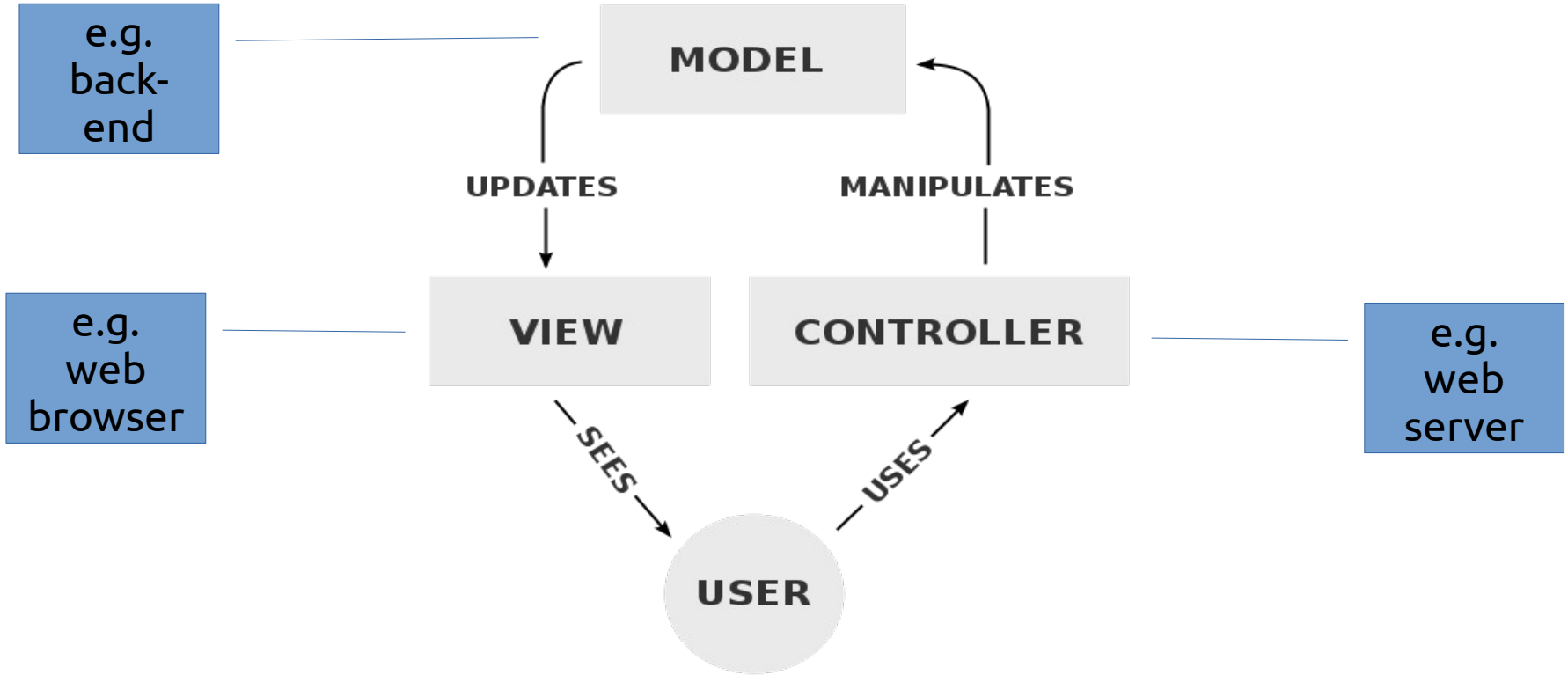
- High-level patterns: view of entire system
- Multiple overlapping views possible
- Examples:
  - Model-View-Controller
  - Client-Server architecture
  - Layered architecture
  - Repository pattern
  - Pipe-and-Filter architecture

Model-View-Controller (MVC)

- Suitable for UI applications
- 3 logical components:
  - Model: manages data & associated operations
  - View: defines/manages presentation to user
  - Controller: manages user interaction
- Advantages:
  - Data can change independently from presentation
- Disadvantages:
  - Can cause extra complexity for simple data models

Model-View-Controller

Image source (PD): https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller



MVC Examples

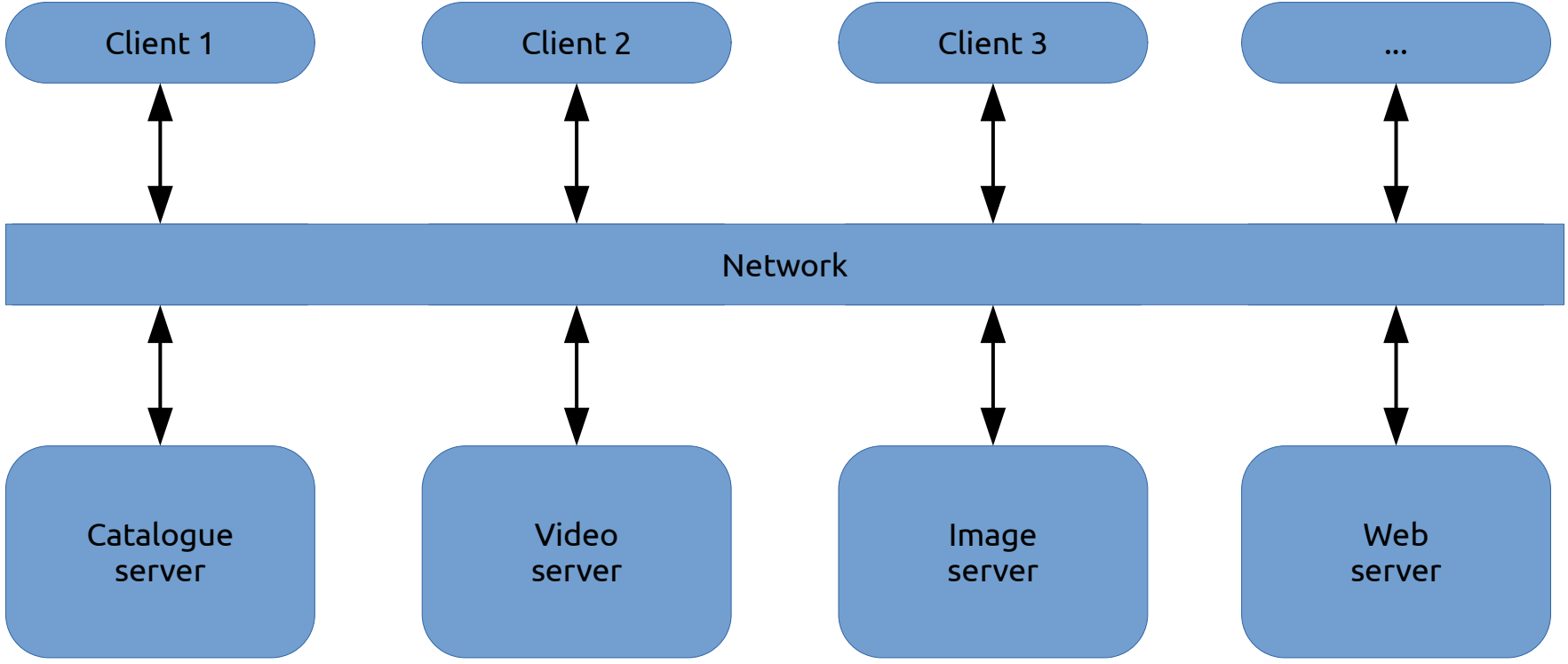
- Website
  - HTML (M), CSS (V), Browser (C)
- Android app:
  - Database (M), Fragments (V), Java code (C)
- Web application:
  - Backend (M), Browser (V), Webserver (C)
  - Database (M), Browser (V), PHP app (C)

Client-server architecture

- Functionality split into services
- Every service provided by one (logical) server
  - Accessing shared data from multiple locations
  - Load balancing → replicate servers
- Advantages:
  - Distributed, network-transparent architecture
- Disadvantages:
  - Every server → single point of failure
  - Network performance unpredictable

Client-server architecture

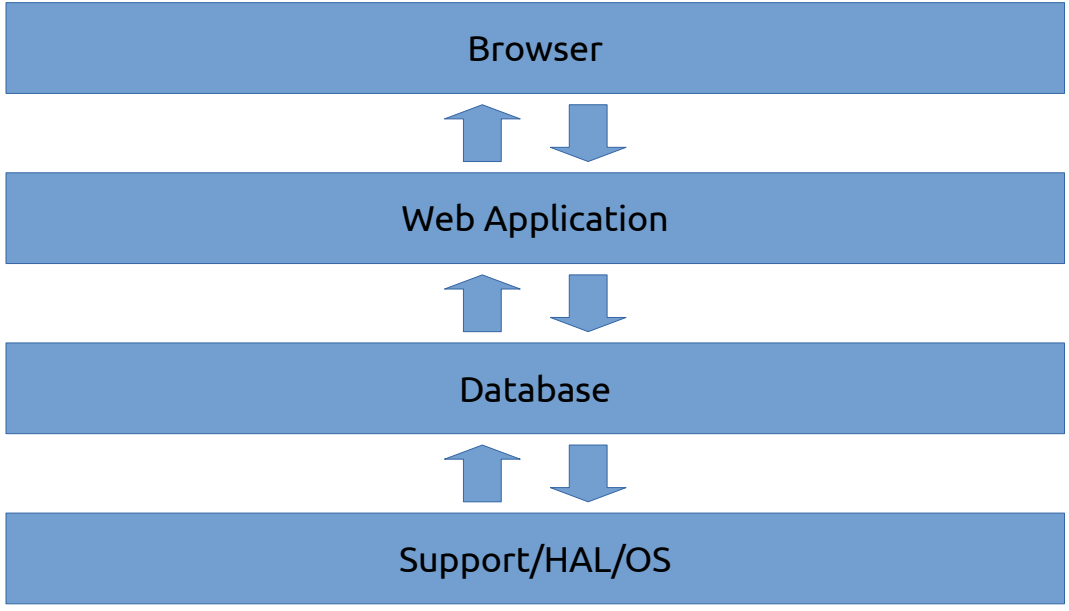
Image source (FU): Sommerville, Software Engineering, Chapter 6



Layered architecture

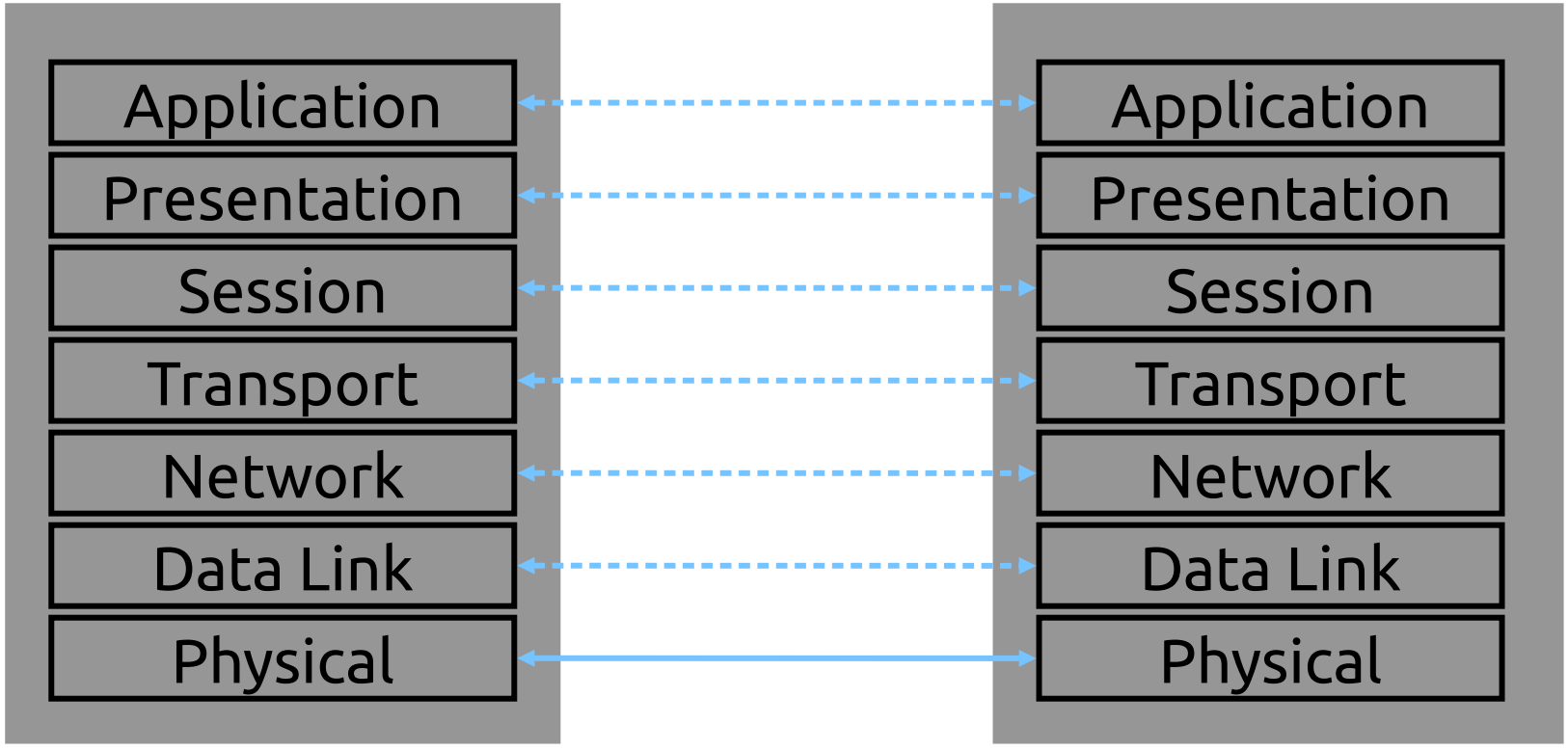
- System as a stack of interconnected layers
  - Layers only communicate with neighbours
  - Increasing complexity from top to bottom
- Often used for network protocols
- Advantages:
  - Individual layers can be replaced
  - Clear interface specification built in
- Disadvantages:
  - Clean separation can be difficult

Example: layered UI architecture



Question: where to put security features?

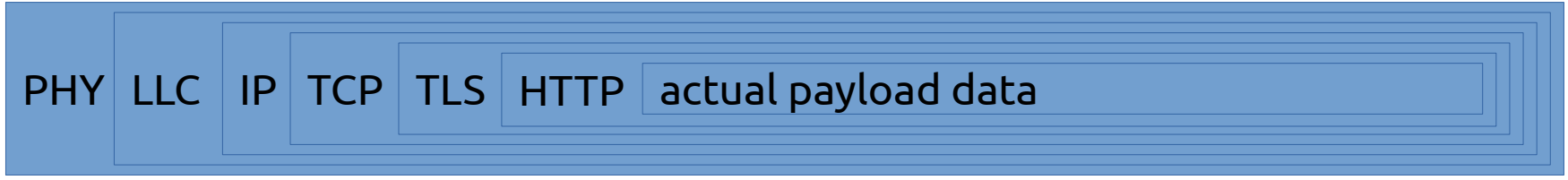
ISO/OSI network model



Internet network model

Internet network model (2)

- Layered protocols → nested data packets (think of Russian Dolls)
- Packets consist of header + payload
- Payload of protocol 1 = packet of protocol 2

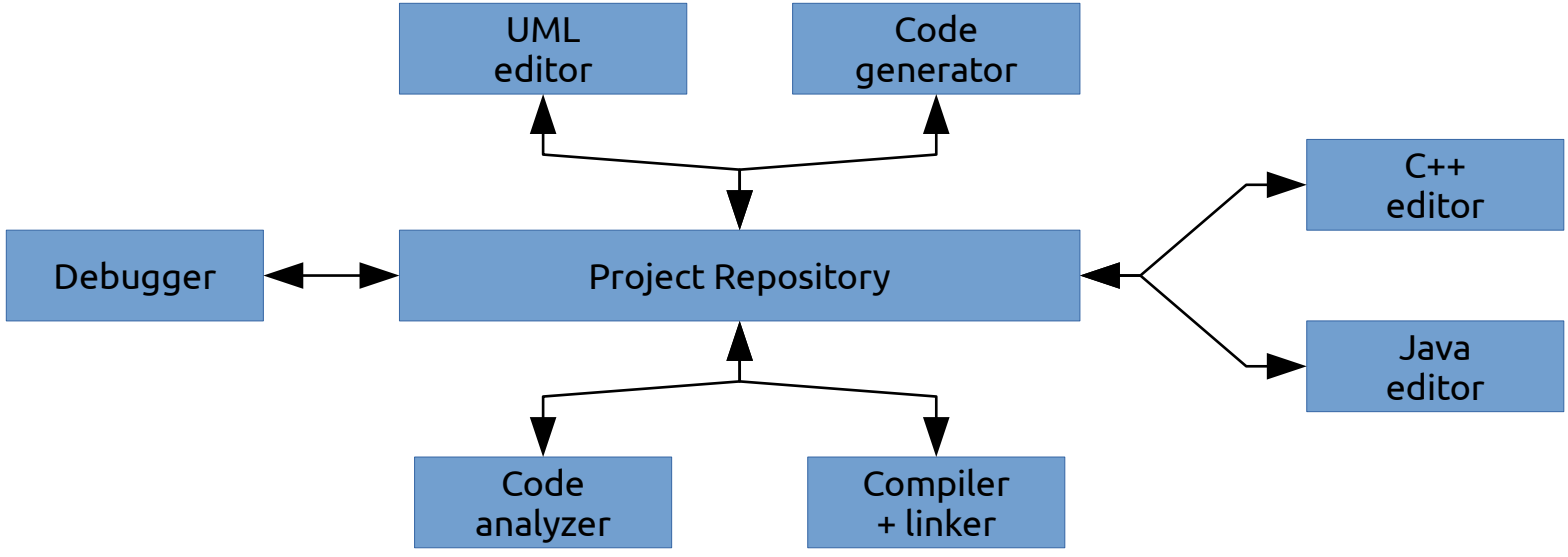


Repository pattern

- All data is managed in a central repository/DB
  - Components do not interact directly
  - Only communicate with the repository
- Advantages:
  - Components can be independent
  - Easy data management (e.g. backups)
- Disadvantages:
  - Single point of failure, may be inefficient
  - Distribution/synchronization difficult

Repository pattern: CASE system

Image source (FU): Sommerville, Software Engineering, Chapter 6



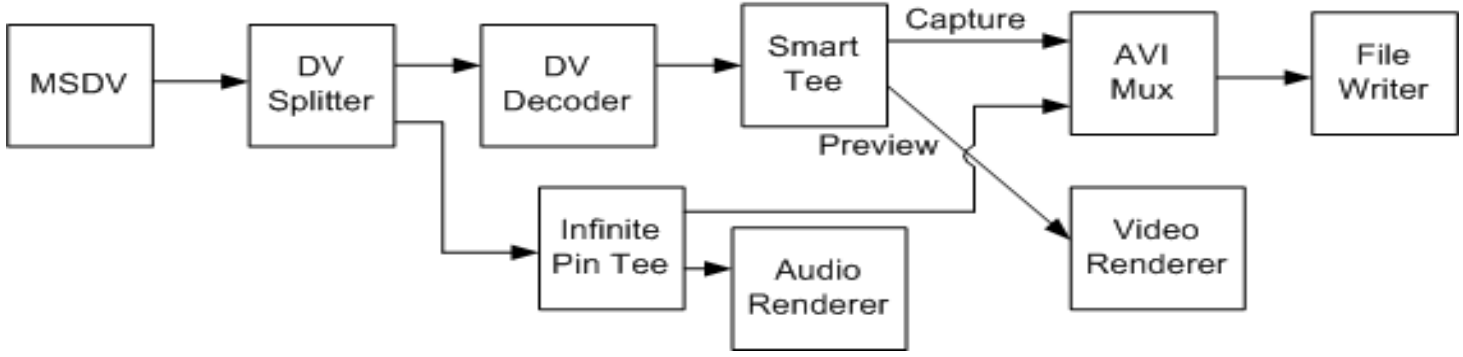
CASE = Computer Aided Software Engineering, e.g. Eclipse

Pipe-and-Filter architecture

- Suitable for audio/video processing, computer vision, text processing …
  - Many components (filters) which provide data transformations
  - Data flows through pipes/network of components
- Advantages:
  - Easy to extend, matches many real-world processes
- Disadvantages:
  - Requires either common data format or costly conversions

Pipe-and-Filter architecture

Image source (FU): https://msdn.microsoft.com/en-us/library/dd318616%28v=vs.85%29.aspx



Architecture Considerations

- *Performance* – Minimize communications, use large-grained components
- *Security* – use e.g. layered architecture with critical assets in inner layers
- *Safety* – Localize safety-critical features in few subsystems
- *Availability* – Include redundant components
- *Maintainability* – Use small, replaceable components

Design patterns for program code

- Reference: "*Design Patterns – Elements of Reusable Object-Oriented Software*" by Gamma, Helm, Johnson and Vlissides
- Also known as Gamma-, "GangOfFour"- or "GoF"-Book
- "Distillation" of 23 common patterns from existing code
- 3 major categories: creational, structural and behavioral patterns

Creational patterns

- Used to *create* other objects
- Rules-of-thumb: http://www.vincehuston.org/dp/creational_rules.html
- Examples:
  - Singleton
  - AbstractFactory
  - FactoryMethod
  - Prototype

Singleton

- Common problem: use only a single instance of a certain class (e.g. a Factory)
- Creation of additional instances should not be possible
- Solution: make class responsible for its own singular instance

Singleton: Code example

Source (FU): http://www.vincehuston.org/dp/singleton.html

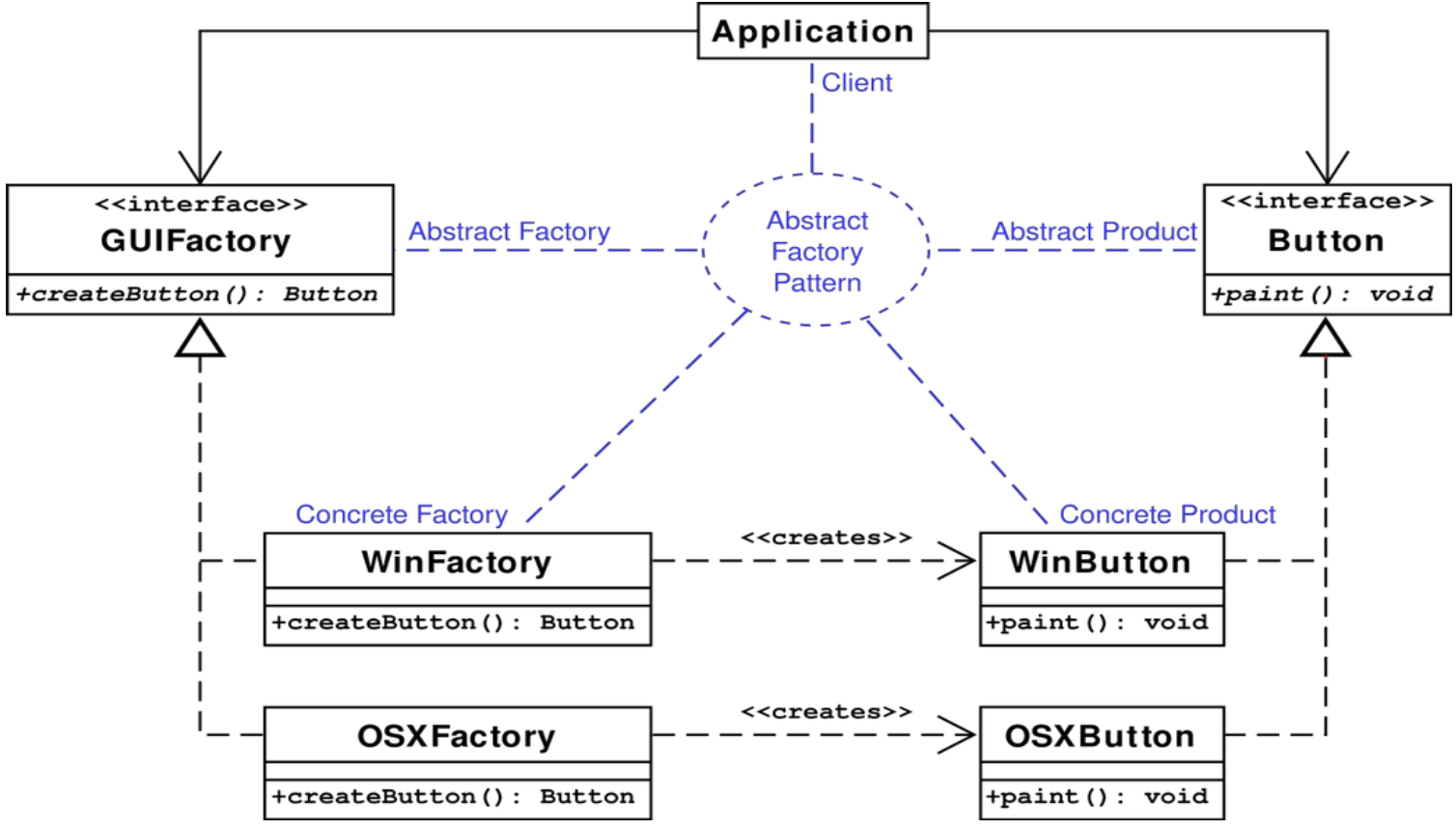
```
public class GlobalClass {

  private int m_value;
  private static GlobalClass s_instance = null;

  protected GlobalClass( int v ) { m_value = v; }

  public int  get() { return m_value; }
  public void set( int v ) { m_value = v; }

  public static GlobalClass instance() {
    if ( s_instance == null )
      s_instance = new GlobalClass(0);
    return s_instance;
  }
}
```

AbstractFactory

- Use Factory object to delegate object creation
- Useful for multiple families of similar objects
- Exact type determined by specific factory
- Example: different classes of UI elements

AbstractFactory (2)

Image source (CC): https://en.wikipedia.org/wiki/Abstract_factory_pattern



AbstractFactory: Code example

Source (FU): http://www.vincehuston.org/dp/abstract_factory.html

Before:

```
void display_window_one() {
  if (windows) {
    Widget w[] = {
      new WinButton(),
      new WinMenu()
    };
  } else if (OSX) {
    Widget w[] = {
      new OSXButton(),
      new OSXMenu()
    };
  } else if [...]
  w[0].paint();
  w[1].paint();
}
```

After:

```
GUIFactory factory;

if (windows) {
  Factory = new WinFactory();
} else if (OSX) { ... }

void display_window_one() {
  Widget w[] = {
    factory.createButton(),
    factory.createMenu()
  };
  w[0].paint();
  w[1].paint();
}
```

FactoryMethod

- Can be used by the AbstractFactory to create new objects
- Replaces explicit `new(...)` with method
- Central element: static *factory method* in base class, e.g.
  `static BaseClass create();`

FactoryMethod: Code example

Source (FU): http://www.vincehuston.org/dp/factory_method.html

```
class Stooge {
  // Factory Method
  public static Stooge create( int choice ) {
    if (choice == 1) return new Larry();
    else if (choice == 2) return new Moe();
    else if (choice == 3) return new Curly();
    else return null;
  }
  public String slap_stick() { return null; }
}

class Moe extends Stooge {
  public String slap_stick() { return "slap head"; }
}

[...]
```

Prototype

- Alternative way for AbstractFactory to create new objects
- Central element: virtual clone method in subclasses, e.g. `BaseClass clone();`
  - Maintain pool of prototype objects
  - Call `clone()` on appropriate prototype

Prototype: Code example

Source (FU): http://www.vincehuston.org/dp/prototype.html

```
interface Stooge {
  public Stooge clone();
  public String slap_stick();
}

class Factory {
  private static Stooge s_prototypes[] = {
    null, new Larry(), new Moe(), new Curly()
  }
  public static Stooge make_stooge( int choice ) {
    return s_prototypes[choice].clone();
  }
}

class Moe implements Stooge {
  public Stooge clone() { return new Moe(); }
[...]
```

Java Generics Recap

- Template class:

```
public class pair<Type1, Type2> {
  public Type1 first; public Type2 second;
  public pair(Type1 t1, Type2 t2) { first = t1; second = t2; }
}

pair<String,String> translation = new pair<>("foo", "bar");
```

- Template method:

```
public static <Type> pair<Type,Type> twice(Type v) {
   return new pair<>(v,v);
}

pair<Int,Int> meaning_of_life = twice(new Int(42));
```

Outlook – Patterns, Part 2

- Behavioural patterns, e.g. Iterator
- Structural patterns, e.g. Adapter
- UI patterns
- Antipatterns

Questions/suggestions?

Image source (FU): http://www.vincehuston.org/dp/

The Periodic Table of Patterns