CSC301: Introduction to Software Engineering

Lecture 3

Wael Aboulsaadat

Object Oriented Design: design patterns

Design Patterns Types

- Creational Patterns
  - Focus: Creation of complex objects
  - Here we our goal is to provide a simple abstraction for a complex instantiation process.
  - We want to make the system independent from the way its objects are created, composed and represented.
  - Problems solved:
    - Hide how complex objects are created and put together

Creational
Factory Method
Abstract Factory
Builder
Prototype
→ Singleton

Design Patterns Types

- Structural Patterns
  - Focus: How objects are composed to form larger structures
  - They reduce the coupling between two or more classes
  - They introduce an abstract class to enable future extensions
  - They encapsulate complex structures
  - Problems solved:
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility

Structural
Adapter
Bridge
→ Composite
Decorator
Flyweight
Facade
Proxy

Design Patterns Types

- Behavioral Patterns
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Here we are concerned with algorithms and the assignment of responsibilies between objects: Who does what?
  - Behavioral patterns allow us to characterize complex control flows that are difficult to follow at runtime.
  - Problem solved:
    - Too tight coupling to a particular algorithm

Behavioural
Interpreter
→ Template Method
Chain of Responsibility
→ Command
Iterator
Mediator
Memento
→ Observer
State
Strategy
Visitor

Pattern: proxy

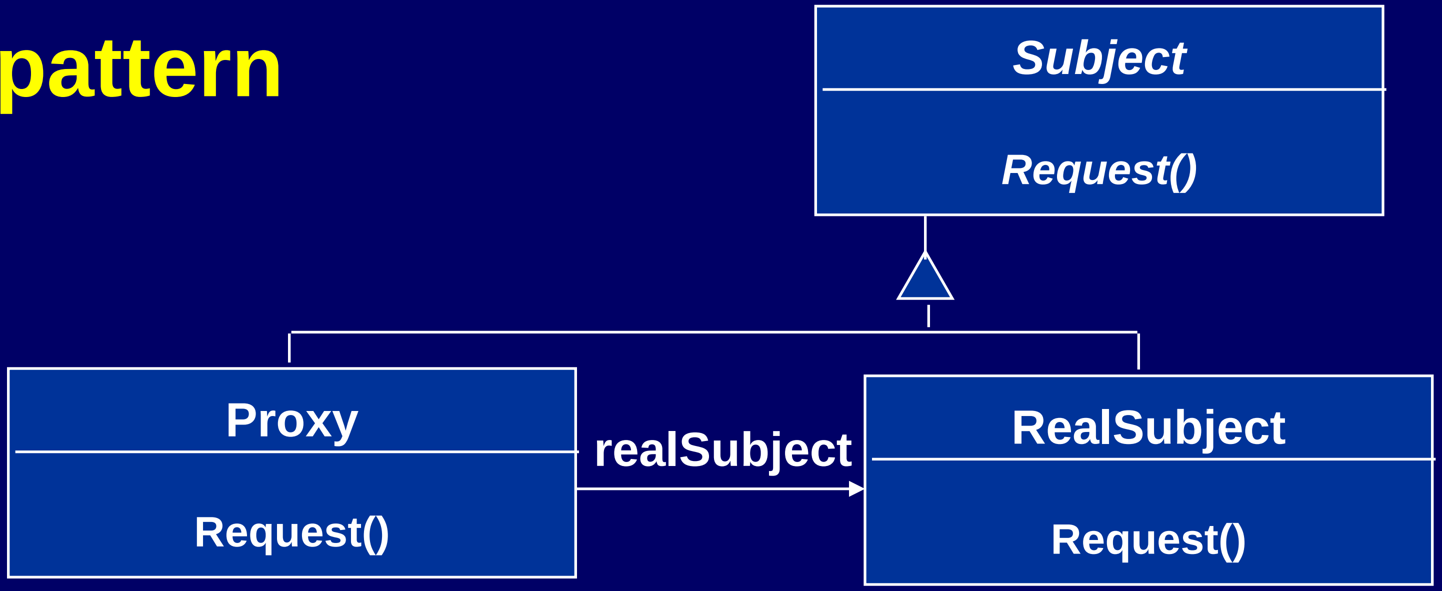
Proxy Pattern: Motivation

- 15:00pm: prime web time. Users with 14.4 baud modem connection can not access web pages with a lot of graphics – their browser times out.
- Which pattern help in this scenario?

Proxy Pattern

- What is expensive?
  - Object download
  - Object Creation
  - Object Initialization
- Defer to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object (“the proxy”) that acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it

Proxy pattern

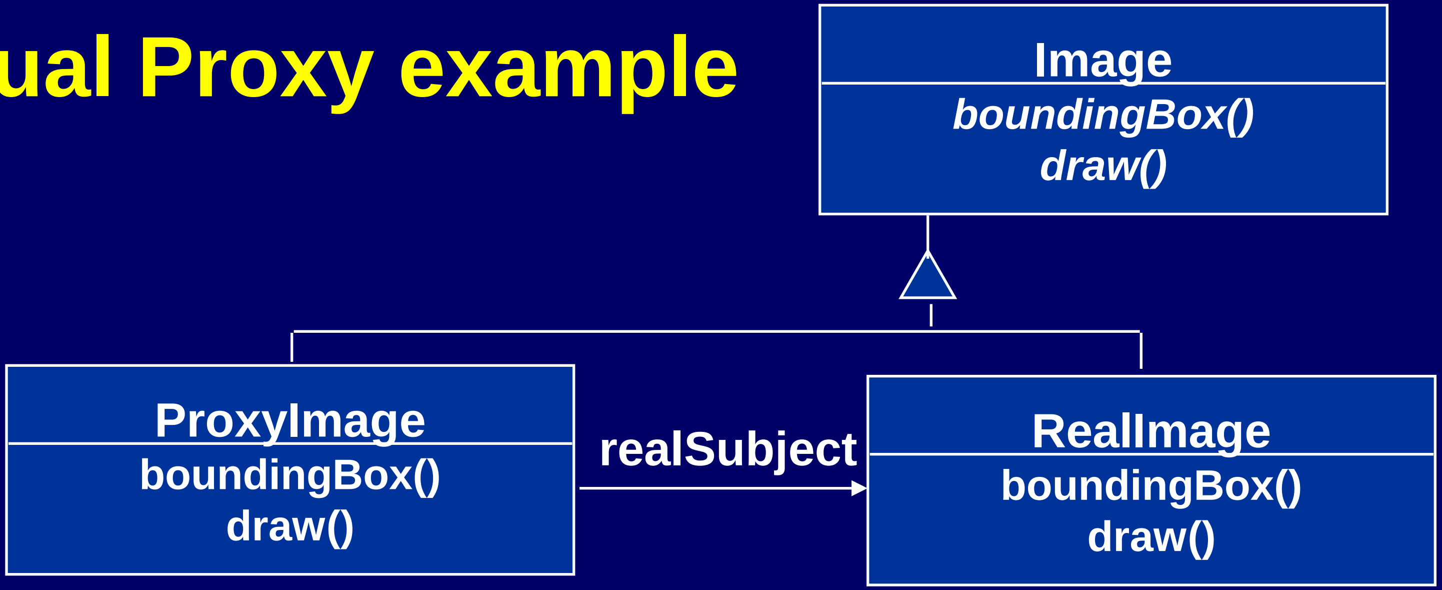


- Interface inheritance is used to specify the interface shared by Proxy and RealSubject.
- Delegation is used to catch and forward any accesses to the RealSubject (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Proxy Applicability

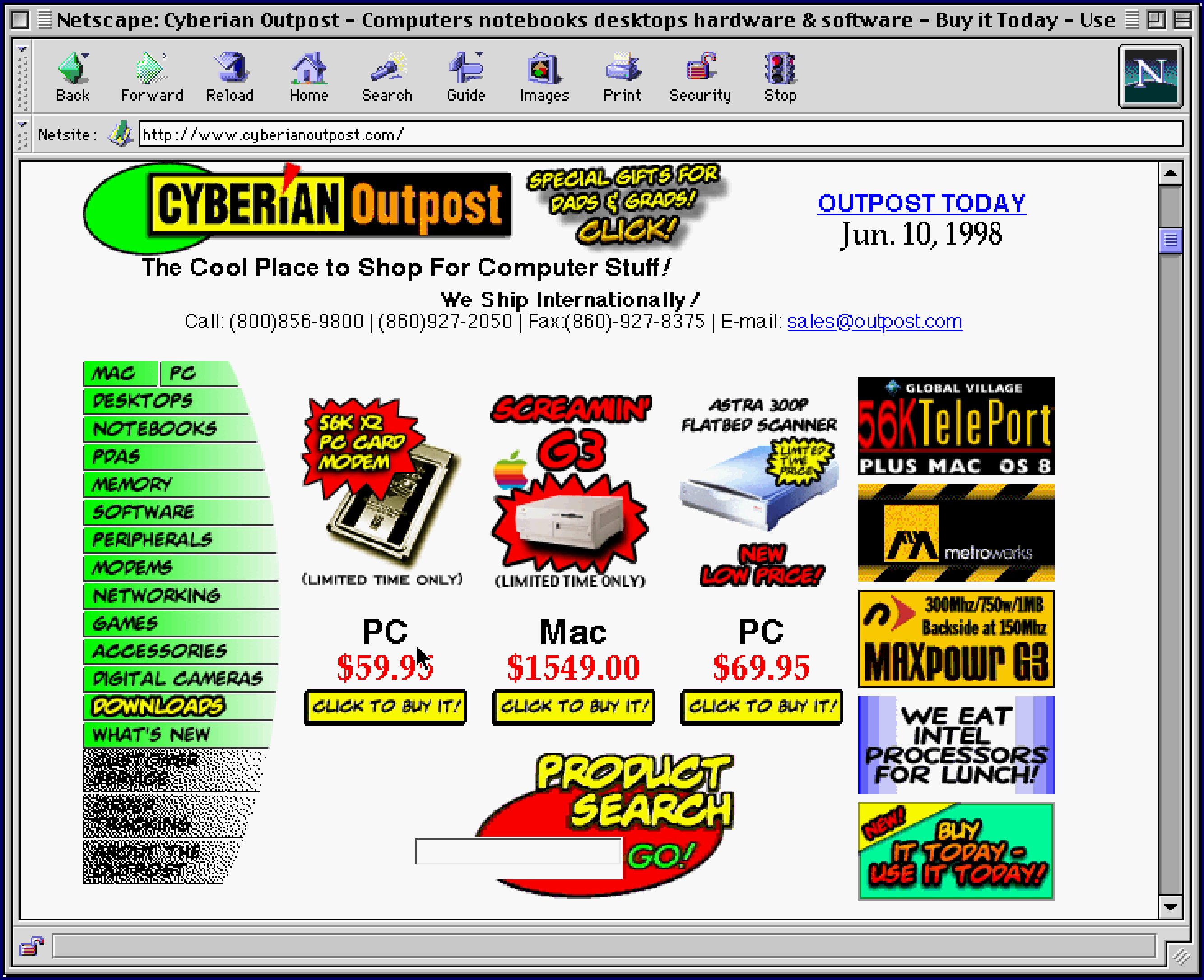
- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.
  - Example: Grade information for a student shared by administrators, teachers and students.

Virtual Proxy example



- Images are stored and loaded separately from text
- If a RealImage is not loaded a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a ProxyImage instead of a RealImage

Before



Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use

http://www.cyberianoutpost.com/

CYBERIAN Outpost

SPECIAL GIFTS FOR DADS & GRADS! CLICK!

OUTPOST TODAY
Jun. 10, 1998

The Cool Place to Shop For Computer Stuff!

We Ship Internationally!

Call: (800)856-9800 | (860)927-2050 | Fax:(860)-927-8375 | E-mail: sales@outpost.com

56K X2 PC CARD MODEM
(LIMITED TIME ONLY)
PC
$59.95
CLICK TO BUY IT!

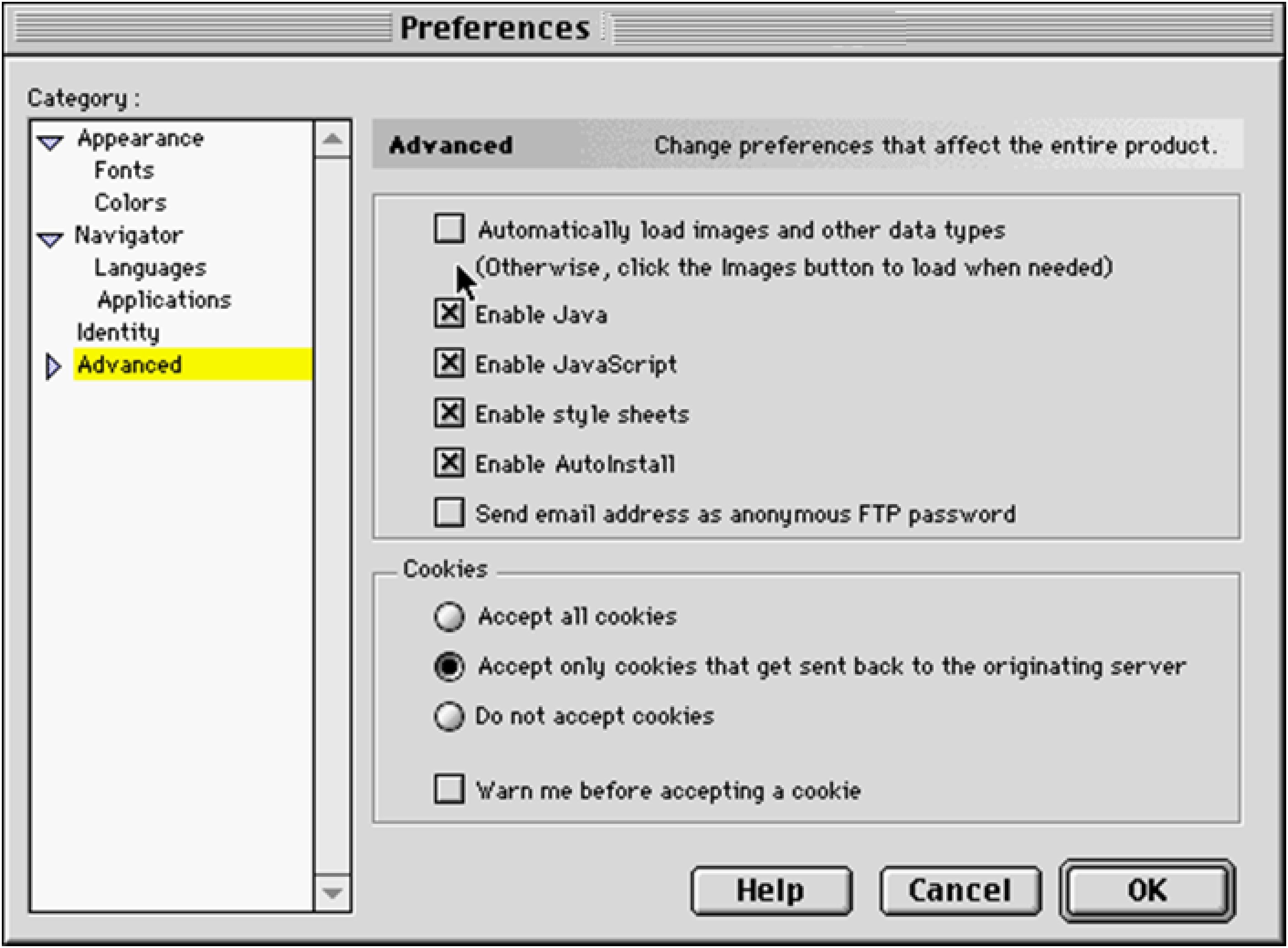
SCREAMIN' G3
(LIMITED TIME ONLY)
Mac
$1549.00
CLICK TO BUY IT!

ASTRA 300P FLATBED SCANNER
LIMITED TIME PRICE
NEW LOW PRICE!
PC
$69.95
CLICK TO BUY IT!

PRODUCT SEARCH
GO!

GLOBAL VILLAGE
56K TelePort
PLUS MAC OS 8
metrowerks
300Mhz/750w/1MB Backside at 150Mhz
MAXpowr G3
WE EAT INTEL PROCESSORS FOR LUNCH!
NEW!
BUY IT TODAY - USE IT TODAY!

Controlling Access



Category :

- Appearance
  - Fonts
  - Colors
- Navigator
  - Languages
  - Applications
- Identity
- Advanced

Advanced Change preferences that affect the entire product.

□ Automatically load images and other data types
(Otherwise, click the Images button to load when needed)

☒ Enable Java

☒ Enable JavaScript

☒ Enable style sheets

☒ Enable AutoInstall

□ Send email address as anonymous FTP password

Cookies

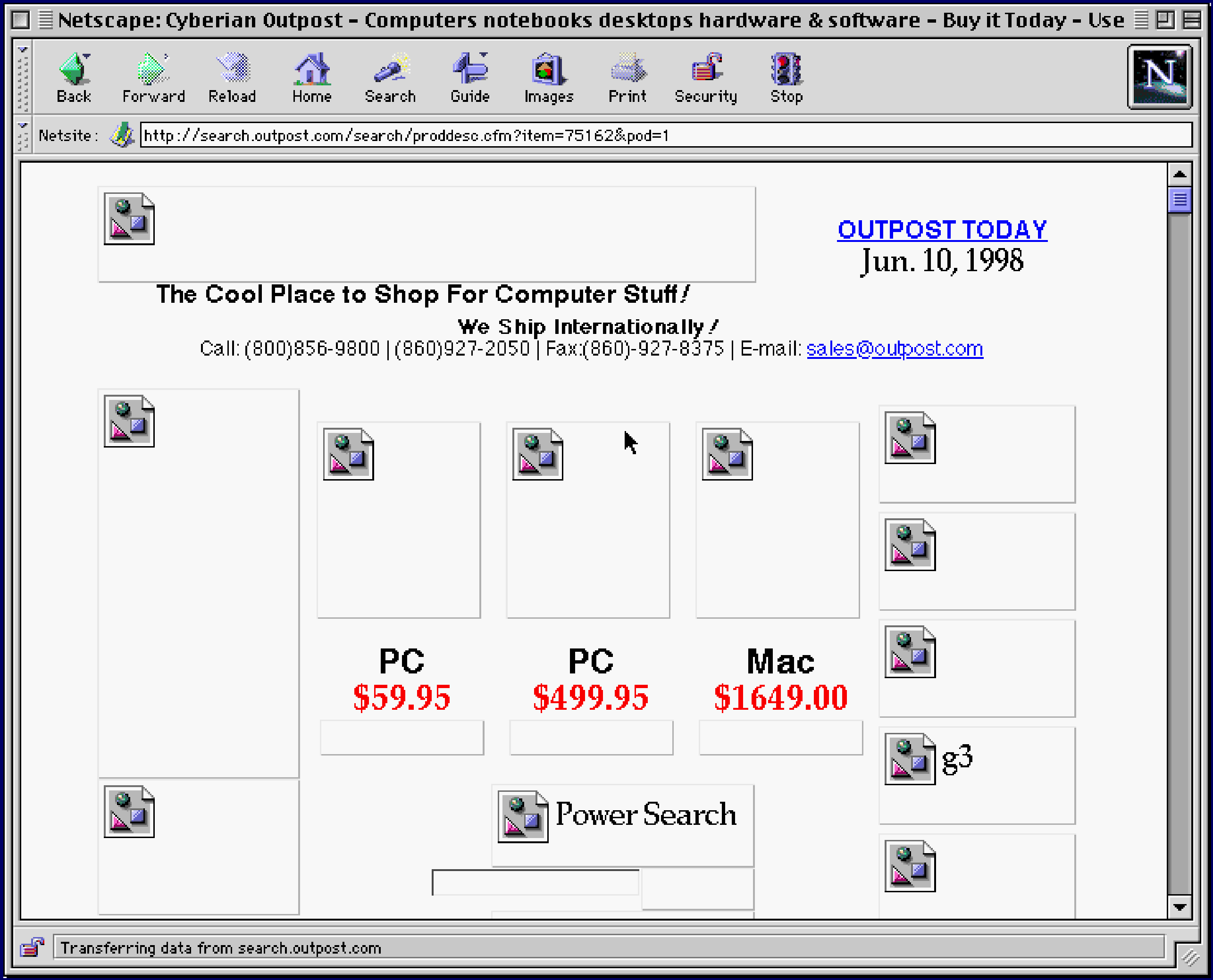
○ Accept all cookies

● Accept only cookies that get sent back to the originating server

○ Do not accept cookies

□ Warn me before accepting a cookie

After



Netscape: Cyberian Outpost - Computers notebooks desktops hardware & software - Buy it Today - Use

http://search.outpost.com/search/proddesc.cfm?item=75162&pod=1

OUTPOST TODAY
Jun. 10, 1998

The Cool Place to Shop For Computer Stuff!

We Ship Internationally!
Call: (800)856-9800 | (860)927-2050 | Fax:(860)-927-8375 | E-mail: sales@outpost.com

PC $59.95

PC $499.95

Mac $1649.00

g3

Power Search

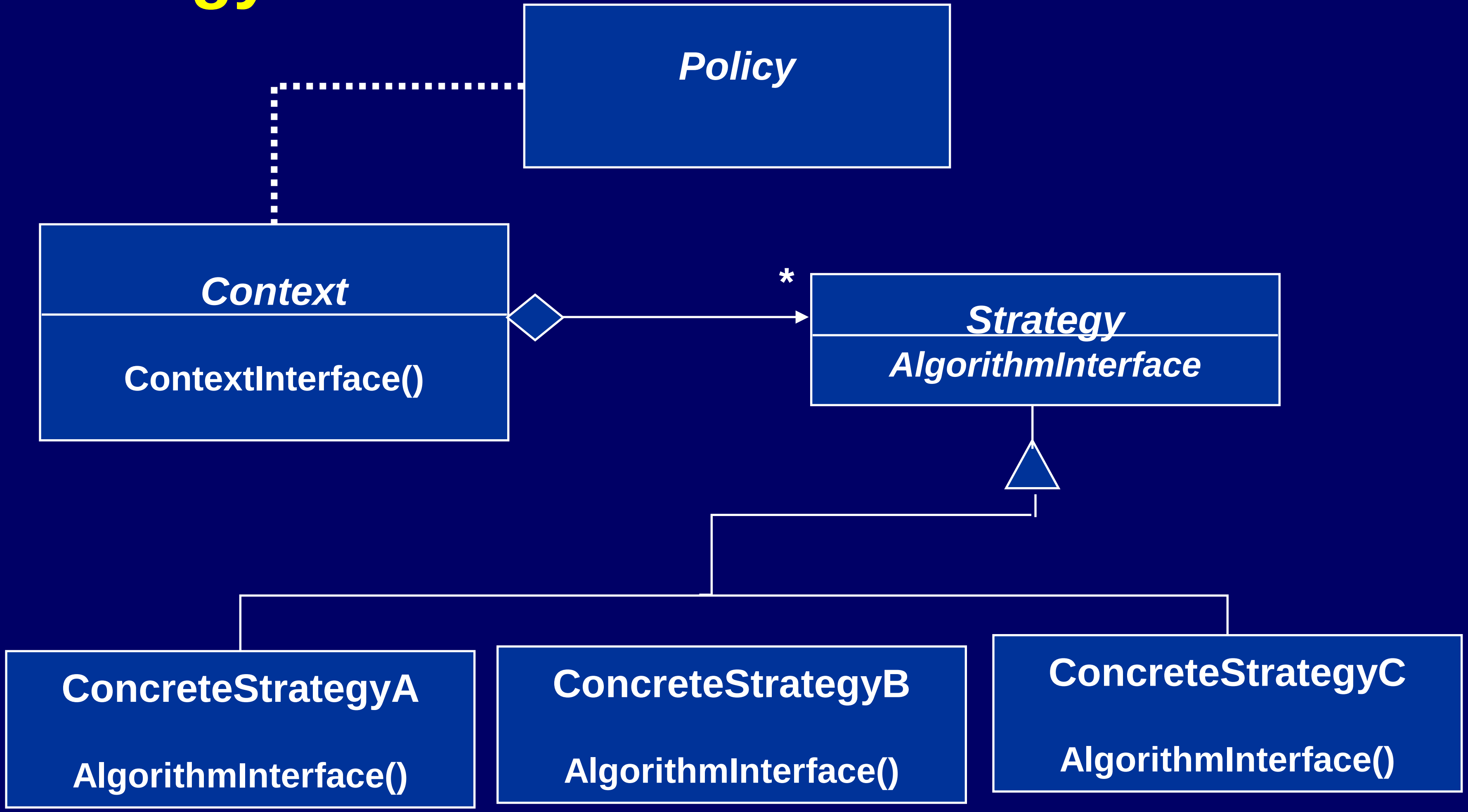
Transferring data from search.outpost.com

Pattern: strategy

Strategy Pattern

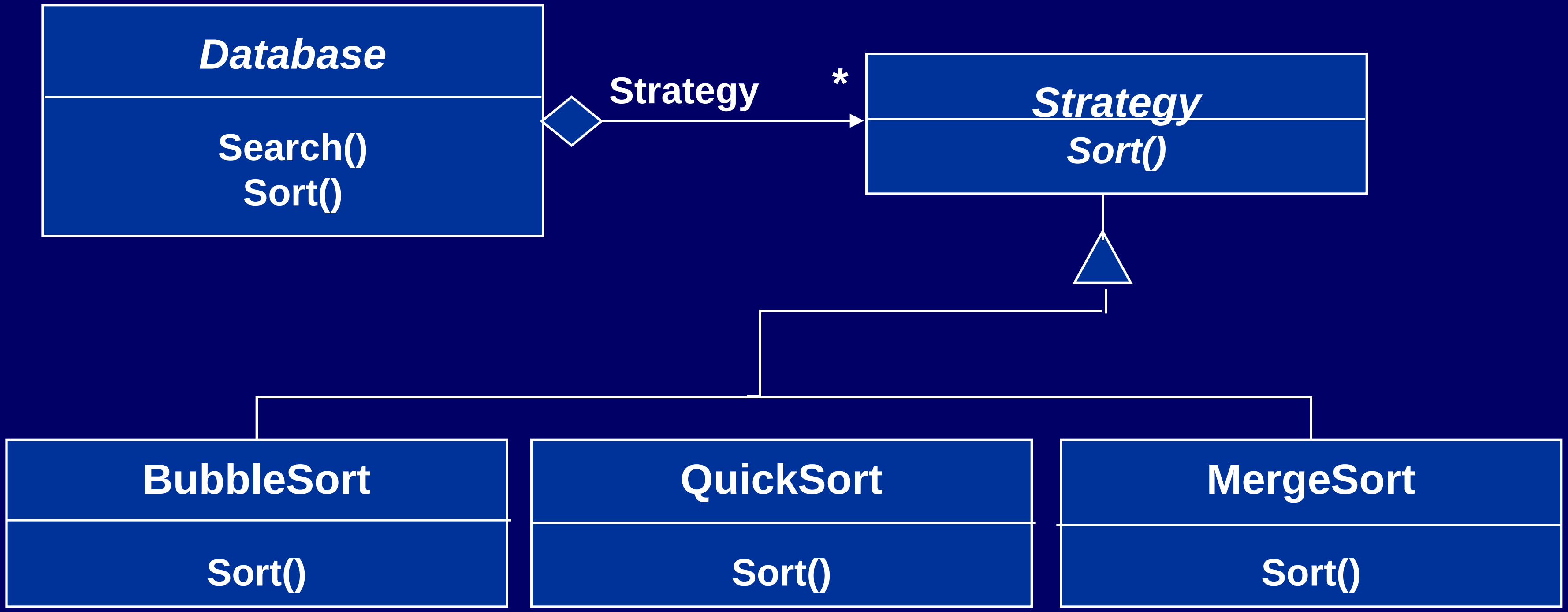
- Many different algorithms exists for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
  - Rapid prototyping vs delivery of final product
- We don’t want to support all the algorithms if we don’t need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm

Strategy Pattern



Policy decides which Strategy is best given the current Context

Applying a Strategy Pattern in a Database Application



Applicability of Strategy Pattern

- Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors

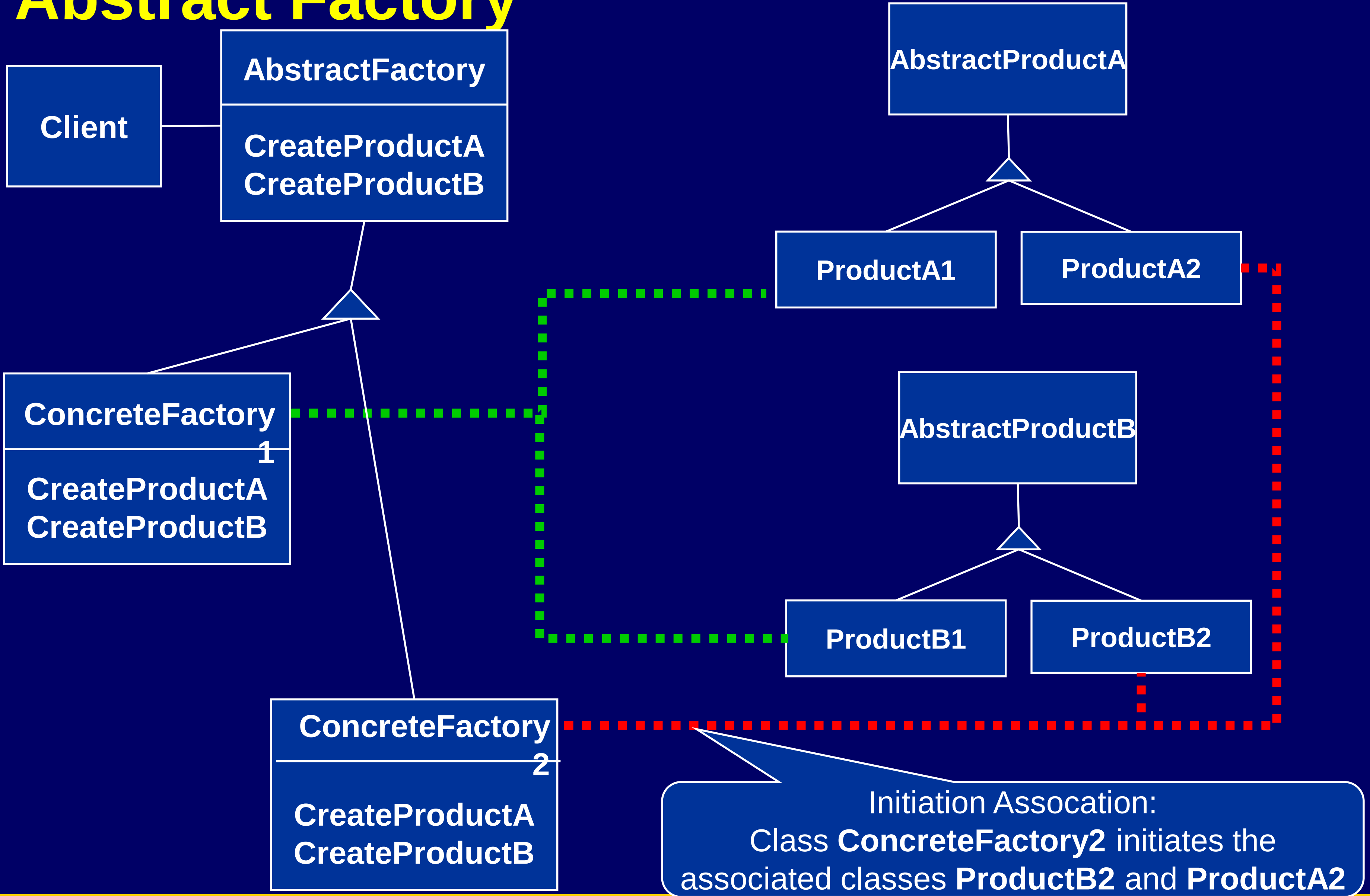
- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms

Pattern: abstract factory

Abstract Factory Motivation

- 2 Examples…
- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in MacOS.
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens’ Instabus, Johnson & Control Metasys or Zumtobe’s proprietary standard.
  - How can you write a single control system that is independent from the manufacturer?

Abstract Factory

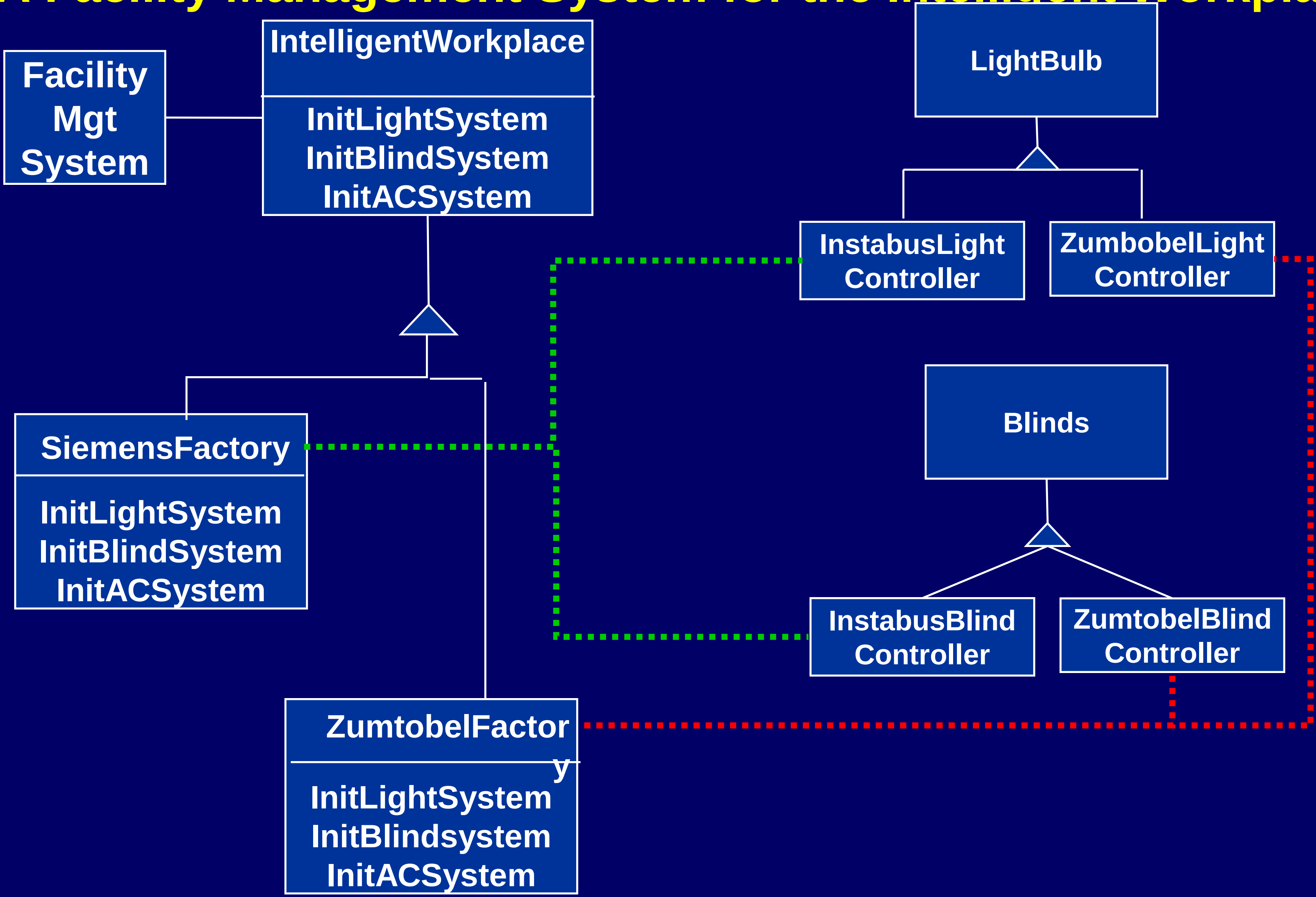


Initiation Assocation:
Class **ConcreteFactory2** initiates the associated classes **ProductB2** and **ProductA2**

Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
  - The system should be independent of how its products are created, composed or represented
- Manufacturer Independence:
  - A system should be configured with one family of products, where one has a choice from many different families.
  - You want to provide a class library for a customer (“facility management library”), but you don’t want to reveal what particular product you are using.
- Constraints on related products
  - A family of related products is designed to be used together and you need to enforce this constraint
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

A Facility Management System for the Intelligent Workplace

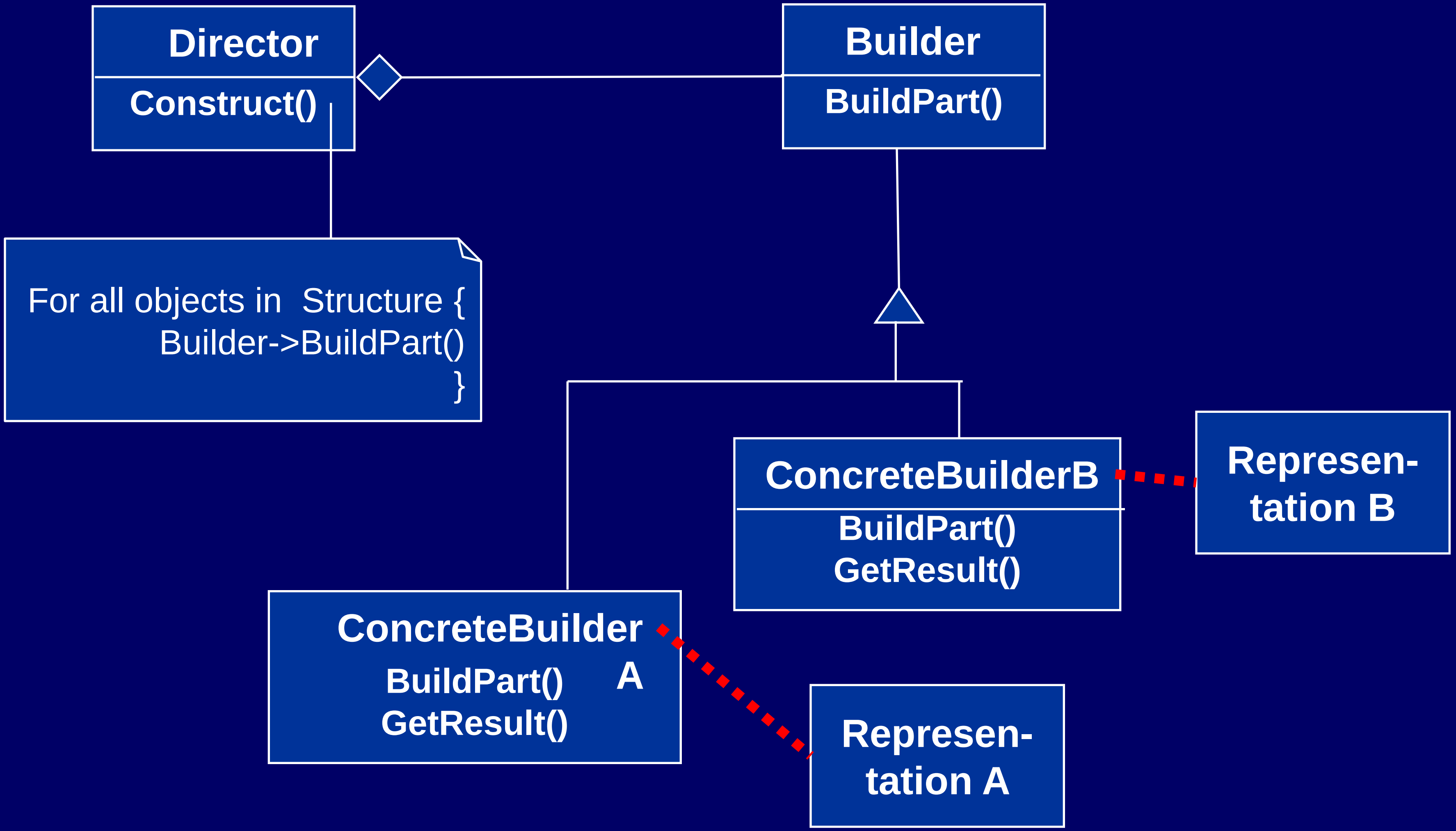


Pattern: builder

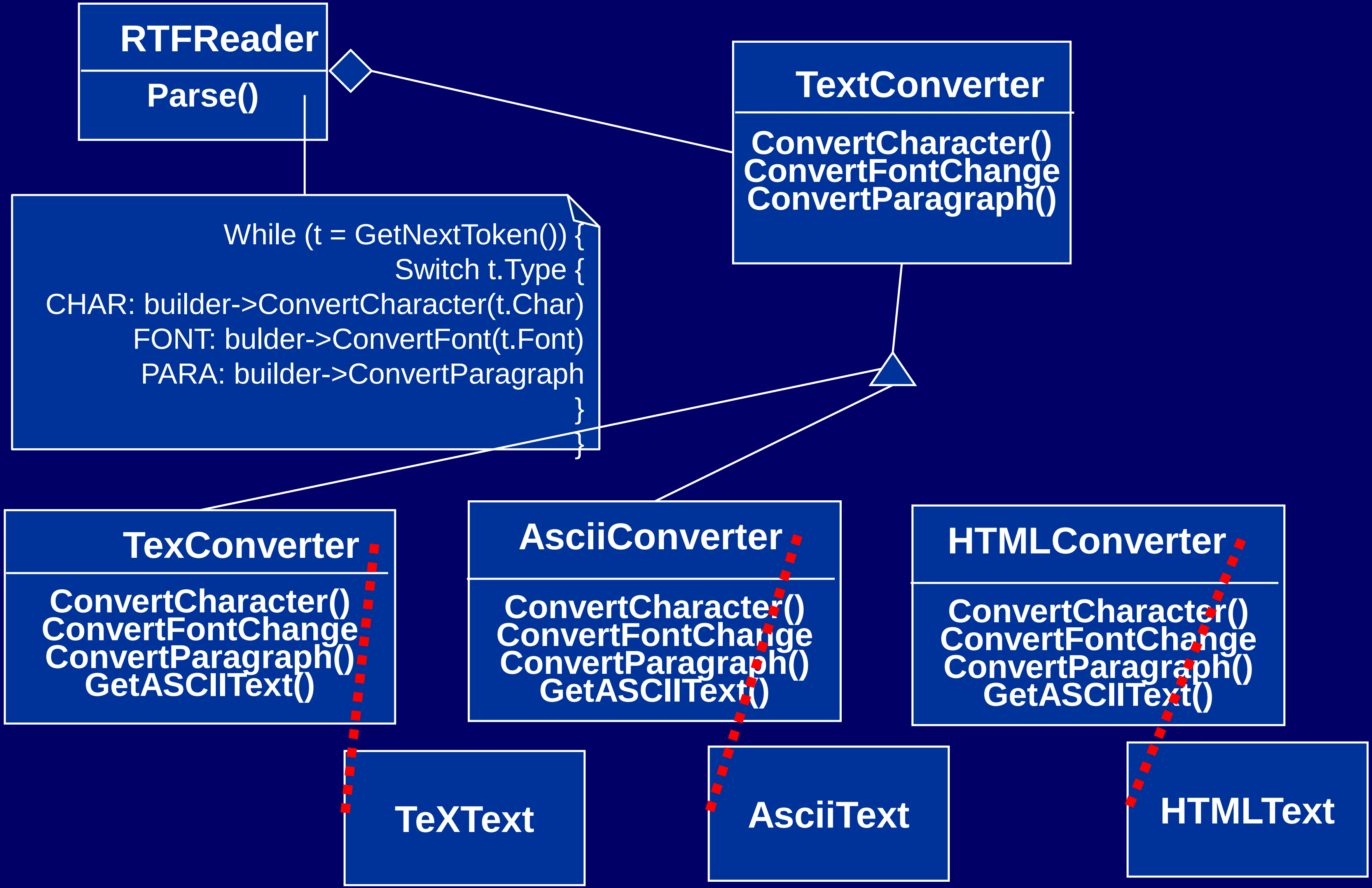
Builder Pattern Motivation

- Conversion of documents
- Software companies make their money by introducing new formats, forcing users to upgrades
  - But you don’t want to upgrade your software every time there is an update of the format for Word documents
- Idea: A reader for RTF format
  - Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ….)
    - *Problem: The number of conversions is open-ended.*
- Solution
  - Configure the RTF Reader with a “builder” object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market

Builder Pattern



Example



RTFReader

Parse()

```
While (t = GetNextToken()) {
Switch t.Type {
CHAR: builder->ConvertCharacter(t.Char)
FONT: bulder->ConvertFont(t.Font)
PARA: builder->ConvertParagraph
}
}
```

TextConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()

TexConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

AsciiConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

HTMLConverter

ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()

TeXText

AsciiText

HTMLText

When do you use the Builder Pattern?

- The creation of a complex product must be independent of the particular parts that make up the product
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)
- The creation process must allow different representations for the object that is constructed. Examples:
  - A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.
  - A skyscraper with 50 floors, 15 offices and 5 hallways on each floor. The office layout varies for each floor.

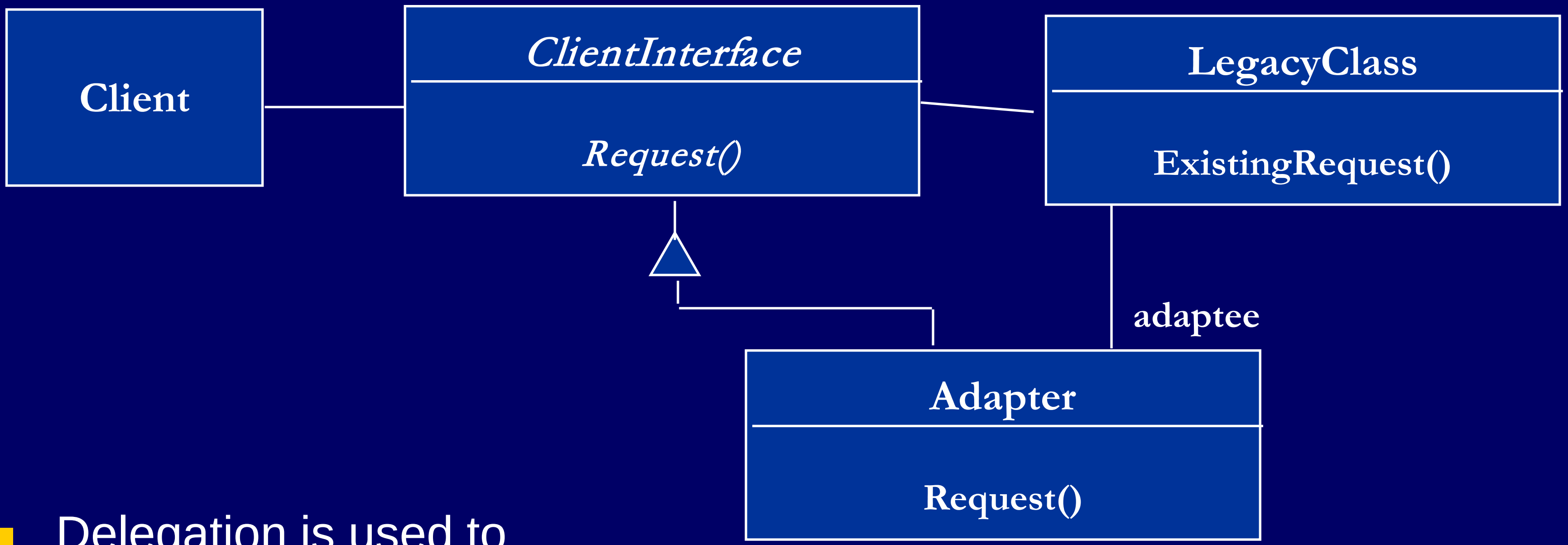
Comparison: Abstract Factory vs Builder

- Abstract Factory
  - Focuses on product family
    - The products can be simple (“light bulb”) or complex (“engine”)
  - Does not hide the creation process
    - The product is immediately returned

- Builder
  - The underlying product needs to be constructed as part of the system, but the creation is very complex
  - The construction of the complex product changes from time to time
  - The builder patterns hides the creation process from the user:
    - The product is returned after creation as a final step

Pattern: adapter

Adapter pattern



- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is use to specify the interface of the **Adapter** class.
- ***Target*** and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**
- **Target** may be realized as an interface in Java.

Adapter Pattern

- “Convert the interface of a class into another interface clients expect.”
- The adapter pattern lets classes work together that couldn’t otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper

Adapter Pattern

- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent. We will only cover object adapters (and call them therefore simply adapters)

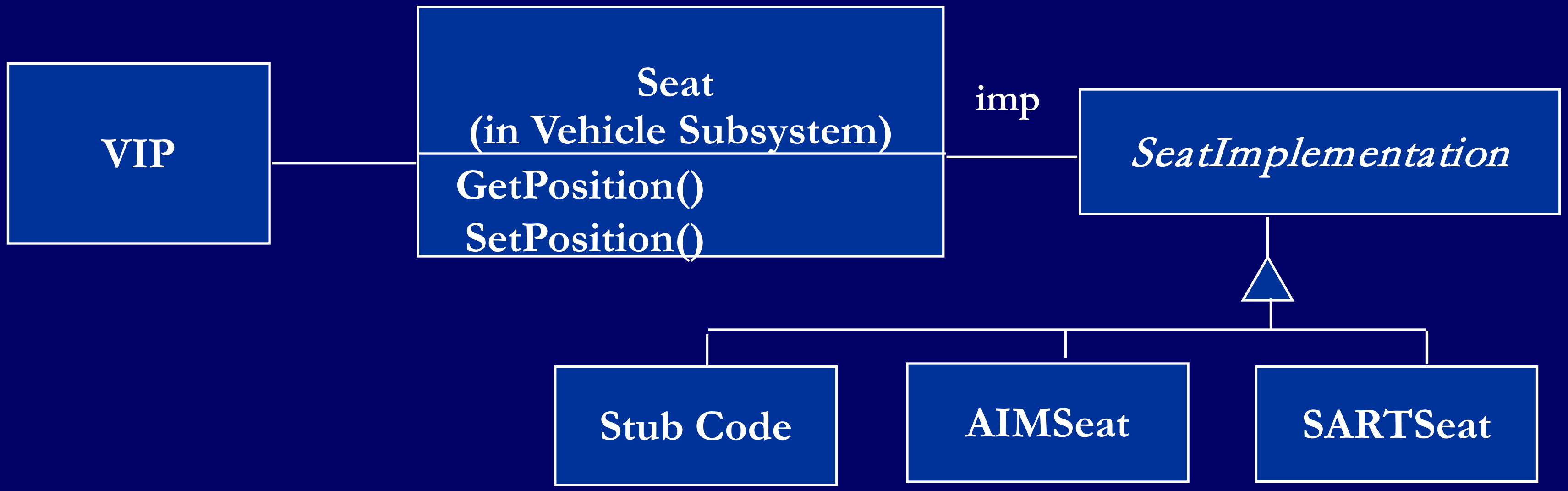
Pattern: bridge

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”.
- Also know as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

Using a Bridge

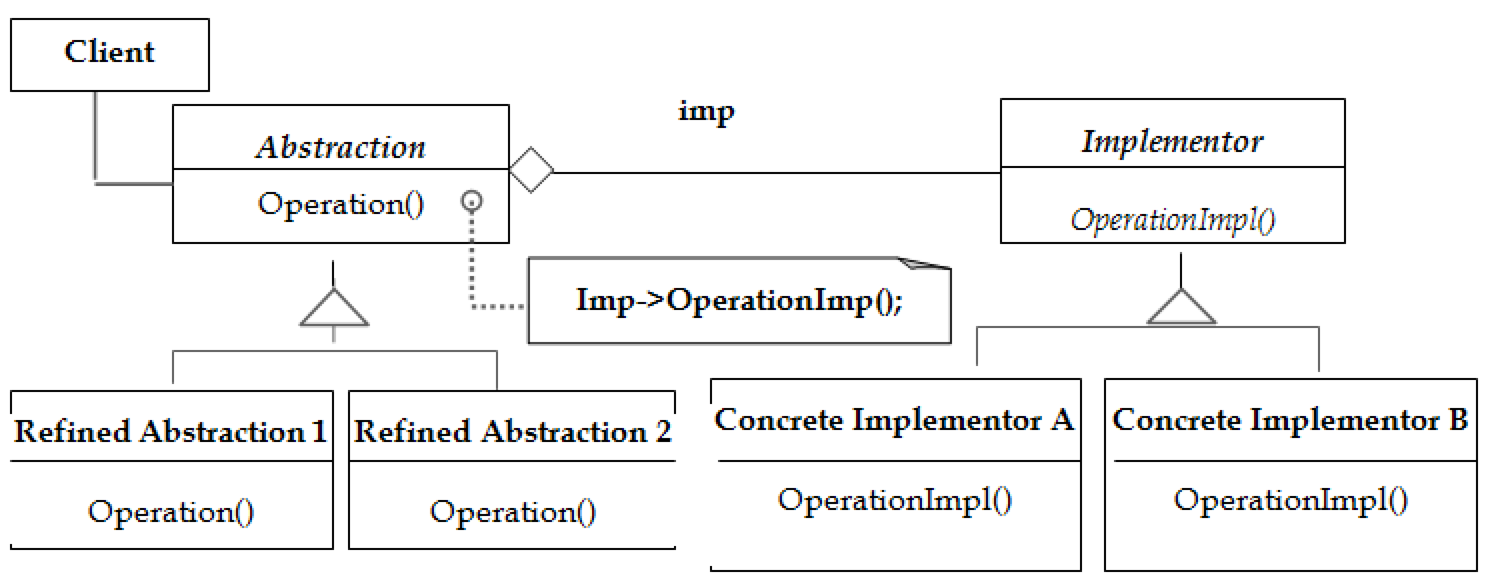
- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project: if seat data is required to be read, but the seat is not yet implemented, known, or only available by a simulation, provide a bridge:



Seat Implementation

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class Stubcode implements SeatImplementation {
  public int GetPosition() {
    // stub code for GetPosition
  }
  …
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  …
}
```

Bridge Pattern



Adapter vs Bridge

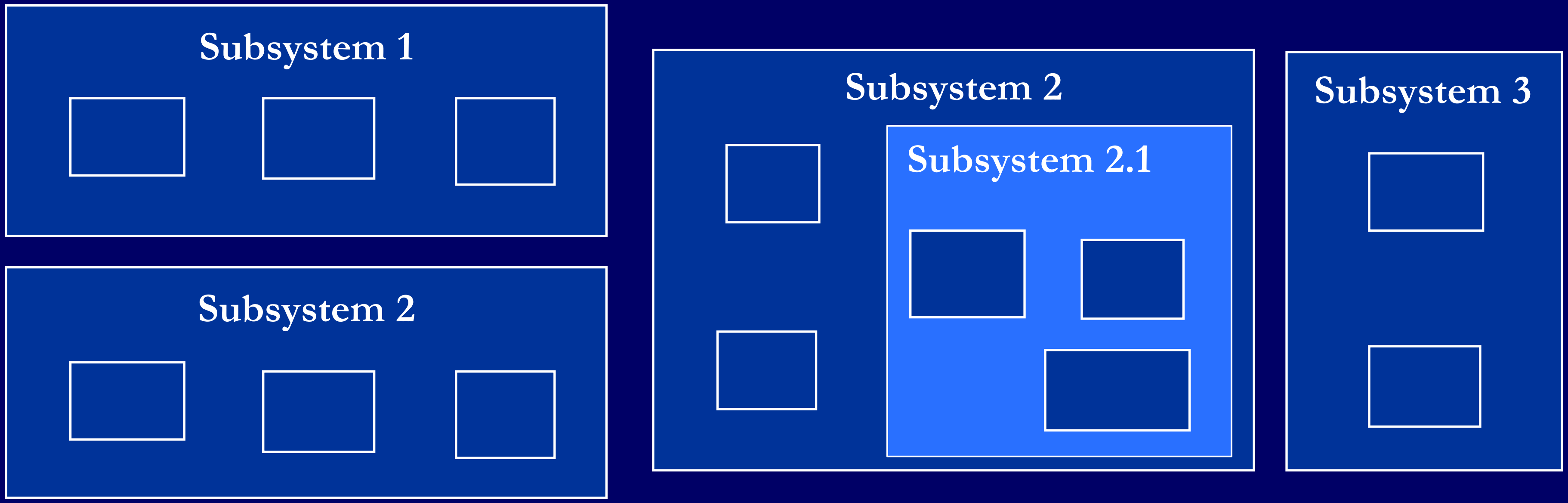
- Similarities:
  - Both are used to hide the details of the underlying implementation.

- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they’re designed (reengineering, interface engineering).
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
    - New “beasts” can be added to the “object zoo”, even if these are not known at analysis or system design time.

Pattern: facade

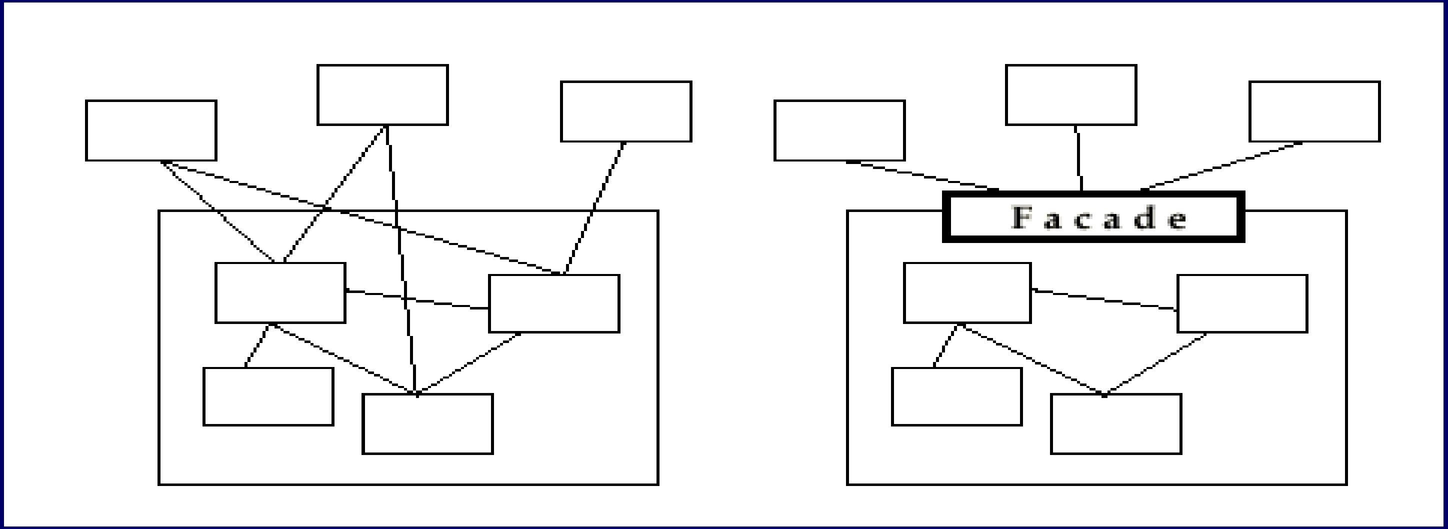
First: what are subsystems ?

- A collection of related classes are grouped together in a conceptual entity we call subsystem
- A software consists of subsystems where every subsystem consist of classes.



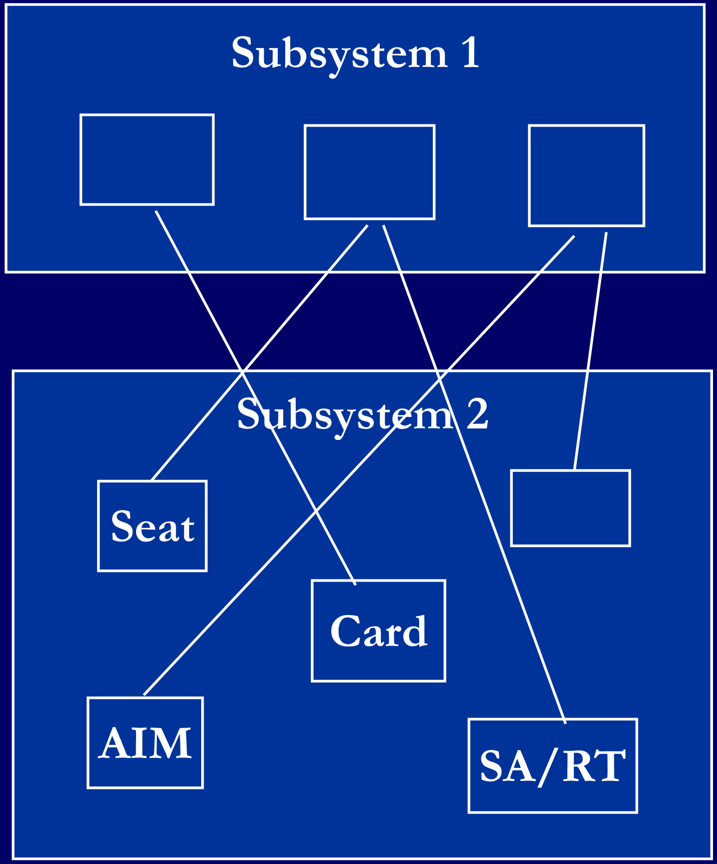
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



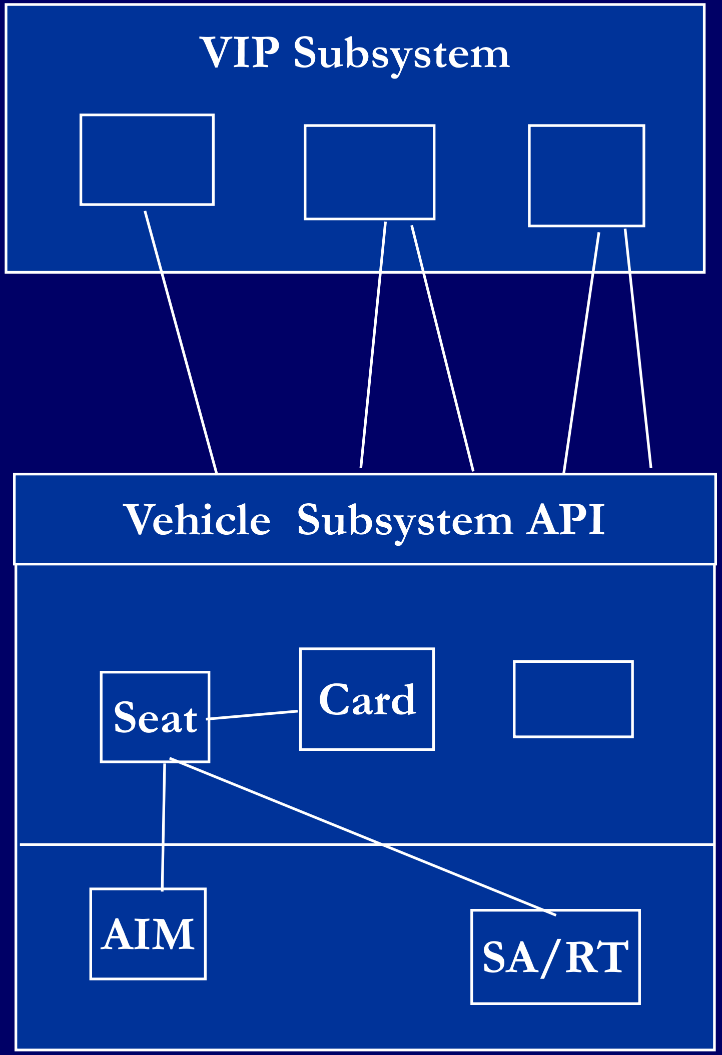
Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code



Realizing an Opaque Architecture with a Facade

- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used the subsystem can be used in an early integration test
  - We need to write only a driver



Design Patterns: discussion

- Not just about object-oriented design
  - User interface patterns
  - Business patterns
  - Anti-patterns (things to avoid)
- Be careful, not every coding problem is a design pattern