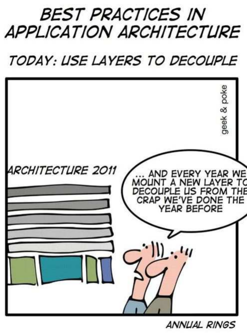


Software Design

www.cs.uoi.gr/~zarras/http://www.cs.uoi.gr/~zarras/se.htm

Slides material sources:

Software Engineering - Theory & Practice, S. L. Pfleeger

Introduction to Software Engineering, I. Sommerville

SWEBOK v3: IEEE Software Engineering Body of Knowledge

R.C. Martin, Agile Software Development, Principles, Patterns, and Practices, 2003

GoF, Design Patterns: Elements of Reusable OO Software, 1995

Design fundamentals

What is software design?

What is software design?

In the general sense, **design** can be viewed as a form of a problem solving process.

In the case of software the input of the design process is the requirements.

What are the basic steps of the design process?

What are the basic steps of the design process?

Architectural design (also referred to as high level design and top-level design) describes how software is organized into components.

Detailed design describes the desired behavior of these components.

What is the outcome of the design process?

What is the outcome of the design process?

The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision.

By recording the rationale, long-term maintainability of the software product is enhanced..

What makes a good design?

Modularity & Decomposition

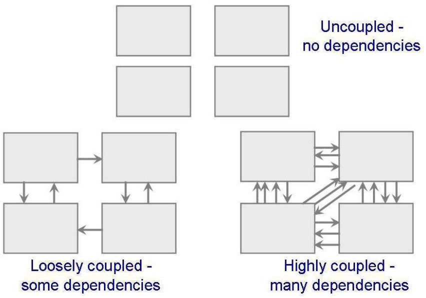
Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe component interactions.

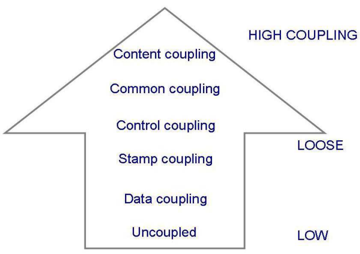
Usually the goal is to place different functionalities and responsibilities in different components.

What makes a good split?

Low coupling

Coupling is a measure of the interdependence among components in a computer program.

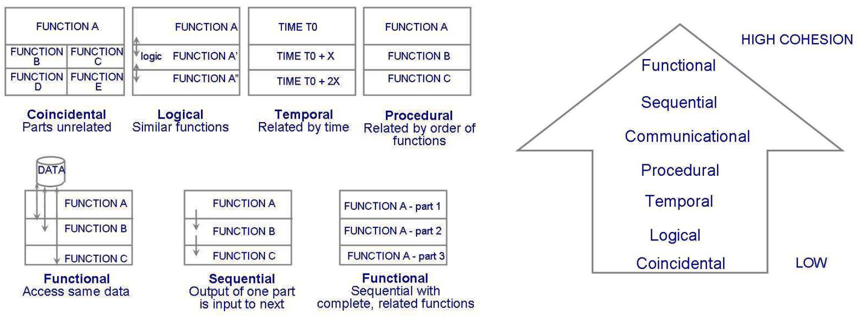




W. P. Stevens, G. J. Myers and L. L. Constantine, Structured design, *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.

High cohesion

Cohesion is a measure of the strength of association of the elements within a component.



W. P. Stevens, G. J. Myers and L. L. Constantine, Structured design, *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.

Abstraction, encapsulation & information hiding

Abstraction is generally defined as a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information.

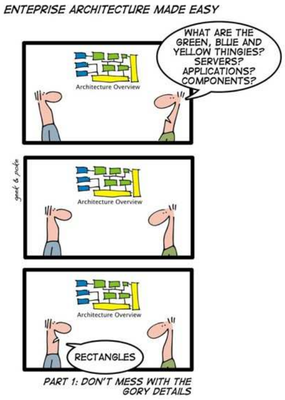
Encapsulation *and* ***information hiding*** *means* grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.

D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*. 15 (12): 1053–58, 1972.

Software Structure & Architecture

What do we mean by software architecture?

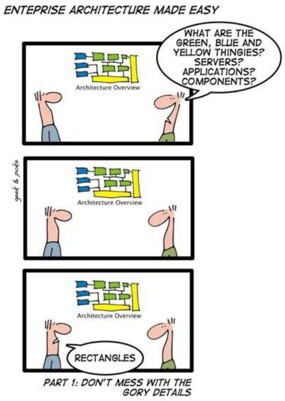
What do we mean by software architecture?



In its strict sense, a **software architecture** is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

P. Clements et al., Documenting Software Architectures: Views and Beyond, Pearson, 2010

What do we mean by software architecture?



During the mid-1990s, however, software architecture **emerged** as **a broader discipline** that involved the study of software structures and architectures in a more generic way.

This gave rise to a number of interesting concepts about software design at different levels of abstraction.

Some of these concepts can be useful during the architectural design (**architectural styles**) as well as during the detailed design (**design patterns**).

Interestingly, most of these **concepts** can be seen as attempts to **describe**, and thus **reuse**, **design knowledge.**

Architectural Styles

What do we mean by architectural style?

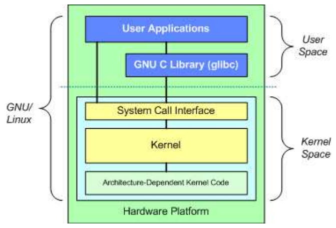
What do we mean by architectural style?

An architectural style determines the vocabulary of components (elements) and connectors (relations) that can be used in instances (architectures) of that style), together with a set of constraints on how they can be combined.

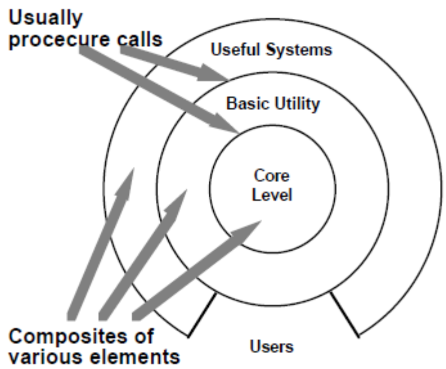
D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, 1993

Which are the major architectural styles?

Architectural styles

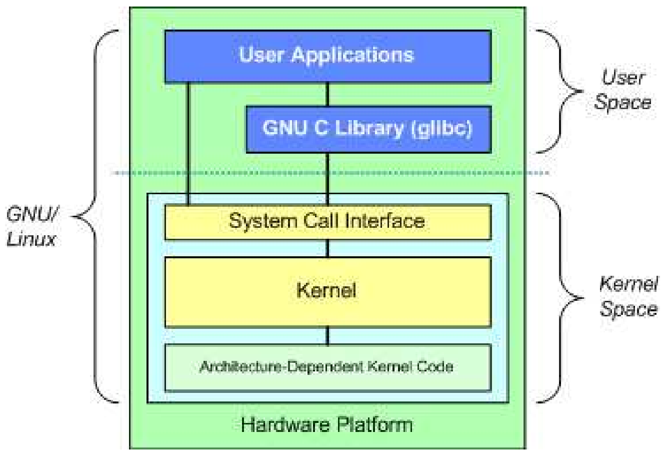


Layered architectures

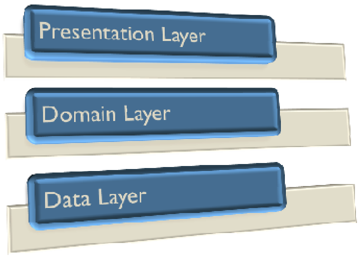


D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I , 1993

Architectural styles



Enterprise Application Architecture



Enterprise Application Architecture

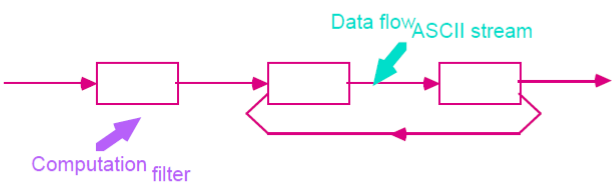
Layer	Responsibilities
Presentation	Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
Domain	Logic that is the real point of the system
Data Source	Communication with databases, messaging systems, transaction managers, other packages

Architectural styles

```
$ls -l | grep "Aug"

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro
-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros $
```

```
$ls -l | grep "Aug" | sort +4n | more

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07
-rw-rw-r-- 1 john doc 14827 Aug 9 12:40 ch03 . . .
-rw-rw-rw- 1 john doc 16867 Aug 6 15:56 ch05

--More--(74%)
```

Pipes and filters



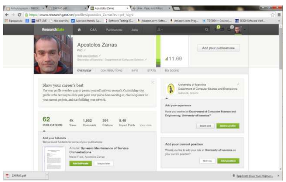
D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I , 1993

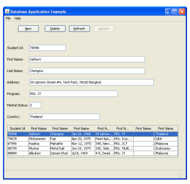
Architectural styles

```
$ls -l | grep "Aug"

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro
-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros $
```

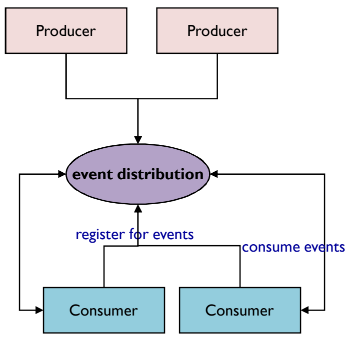
```
$ls -l | grep "Aug" | sort +4n | more

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07
-rw-rw-r-- 1 john doc 14827 Aug 9 12:40 ch03 . . .
-rw-rw-rw- 1 john doc 16867 Aug 6 15:56 ch05

--More--(74%)
```

Architectural styles



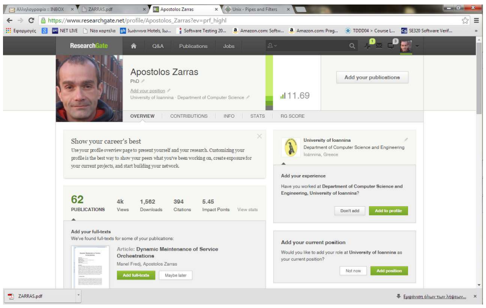


Implicit invocation / event based



D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I , 1993

Architectural styles



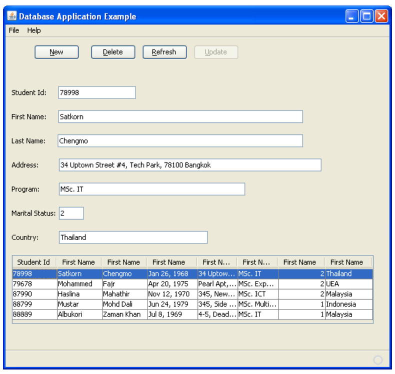
Social networking sites
like ResearchGate Linkedin for professionals and researchers to share papers, ask and answer questions, and find collaborators

People create their profile
People can follow other people

Anyone is an **event producer**
- publication updates
- profile updates
- questions raised

Followers are **event consumers**
- an update to someone you follow results in notifications sent to the followers

Architectural styles



GUI development toolkits

Widgets produce events.

Application objects handle/consume events.

What do these cases have in common ???

Architectural styles

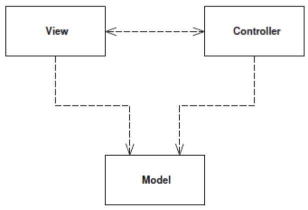
Model – View – Controller (MVC)

The **model** represents some information about the domain.

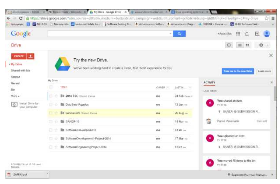
The **view** represents the display of the model in the UI.

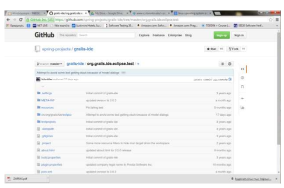
The **controller** takes user input, manipulates the model, and causes the view to update appropriately.

In this way UI is a combination of the view and the controller.

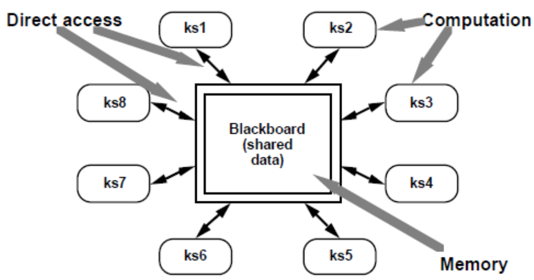


Architectural styles



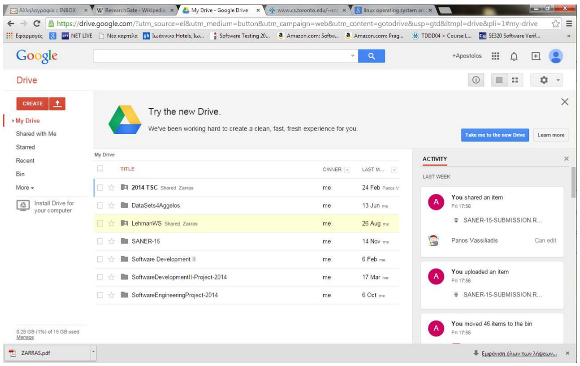


Blackboard/repository



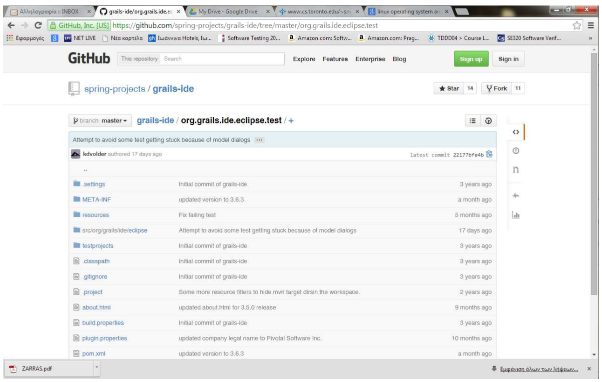
D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, 1993

Architectural styles



Google Drive is a file storage and synchronization service which enables user cloud storage, file sharing and collaborative editing.

Architectural styles



Software repositories like GitHub, SourceForge

Design patterns

What do we mean by pattern?

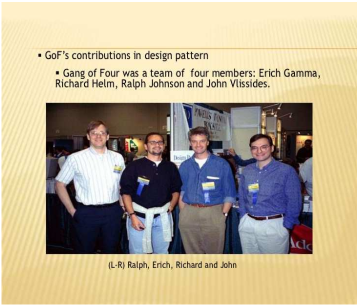
What do we mean by design pattern?

Christopher Alexander says,

"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"

Gang of Four (GoF) patterns

GoF Patterns



GoF say:

The **design patterns** are descriptions of communicating objects and classes that are customized to solve a **general design problem** in a particular context.

Design Patterns: Elements of Reusable Object Oriented Software," Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995

Classification of GoF patterns

Creational	Structural	Behavioral
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Common creational patterns

Factory Method
(& Parameterized Factory Variant)

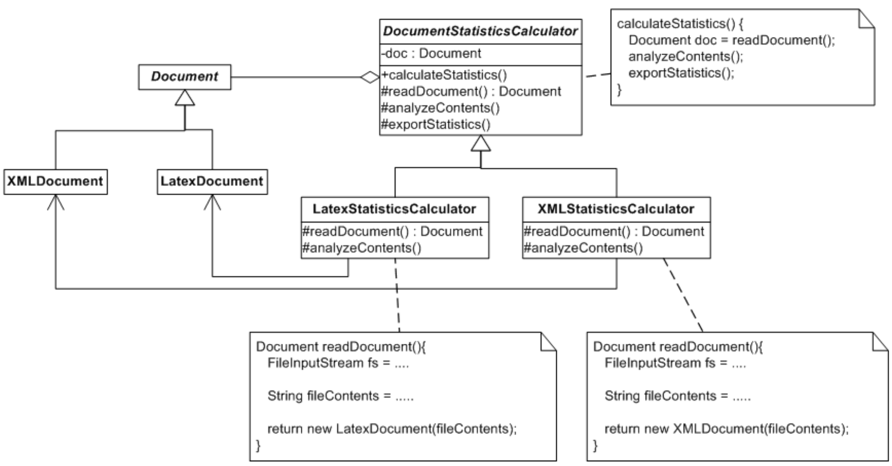
Factory Method

Intent

Factory Method lets a class **defer instantiation of the objects it needs to its subclasses**.

Factory Method

Motivation



Factory Method

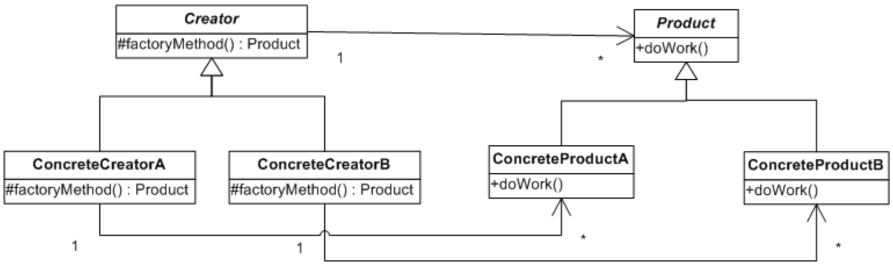
Structure

Product (Document) defines the interface of objects the factory method creates.

ConcreteProduct (LatexDocument) implements the Product interface.

Creator (DocumentStatisticsCalculator) declares the factory method, which returns an object of type Product; may call the factory method to create a Product object.

ConcreteCreator (LatexStatisticsCalculator) overrides the factory method to return an instance of a ConcreteProduct.



Factory Method

Benefits

- We **avoid tight coupling** between the **Creator** and the **concrete Product objects**.
- We keep the **Product object creation code** in **one place.**
- We can **introduce new types of products** into the program without changing **Creator** code.

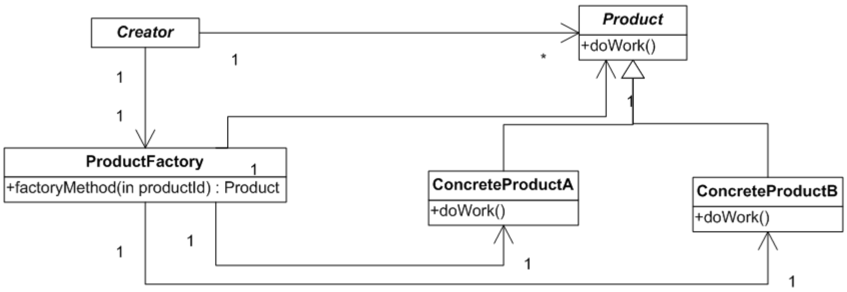
Liabilities

- We need to add **several small subclasses** to implement the pattern.

Parameterized Factory Variant

Structure

Parameterized factory. A variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create.



Prototype

Prototype

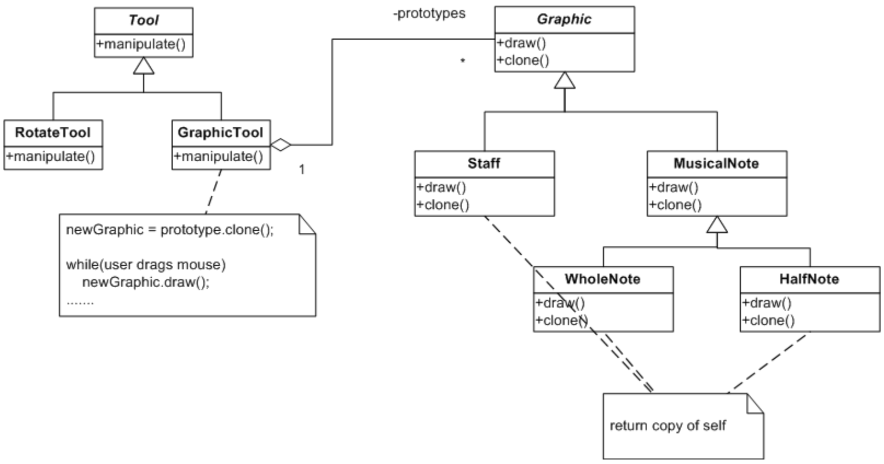
Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Prototype

Motivation To realize the tools…

To realize the graphic elements…



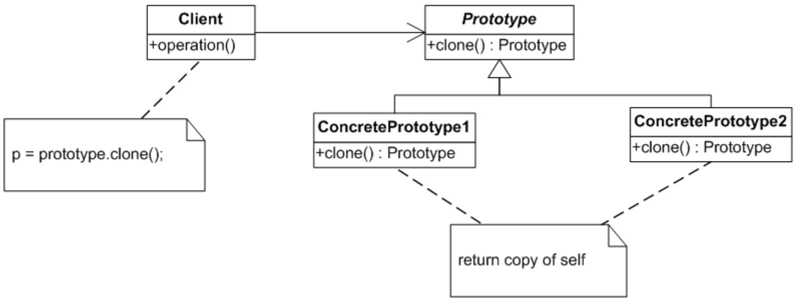
Prototype

Structure

Prototype (Graphic) declares an interface for cloning itself.

ConcretePrototype (Staff, WholeNote, HalfNote) implements an operation for cloning itself.

Client (GraphicTool) creates a new object by asking a prototype to clone itself.



Prototype

Benefits

- We can **create objects without coupling** to their **concrete classes**. So, we can easily **parameterize** the **creating objects** with different classes of **prototypical objects**.
- We **can get rid of repeated initialization code** in favor of cloning pre-built prototypes and produce complex objects more conveniently.

Liabilities

- Cloning complex objects can be **tricky**. **Shallow** vs **deep** copies, circular dependencies, ….

Singleton

Singleton

Intent
Ensure a class only has **one instance**, and provide a **global point of access** to it.

Singleton

Motivation

It's important for some classes to have exactly one instance.

Although there can be many printers in a system, there should be only one printer spooler.

There should be only one file system and one window manager.

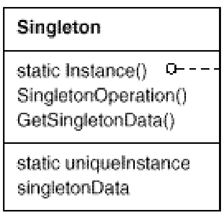
An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible?

A nice solution is to make the class itself responsible for keeping track of its sole instance.

Singleton

Structure




```
if(uniqueInstance == null){
  uniqueInstance = new Singleton();
}

return uniqueInstance;
```

Singleton

Benefits

- **Guarantees** that a class has only a **single object**.
- Provides a **global access** point to that object.

Liabilities

- Mixing **object management** with **object behavior** in **one class**.
- A **global unique object** can make **debugging** and **unit testing difficult**; singleton objects cannot be easily **mocked**.

Common structural patterns

Adapter

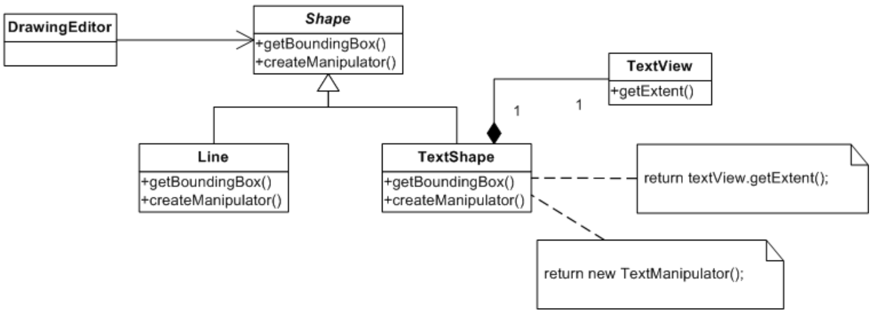
Adapter

Intent

Convert the **interface** of a **class** into **another interface clients expect**. Adapter **lets classes work together** that couldn't otherwise because of **incompatible interfaces**.

Adapter

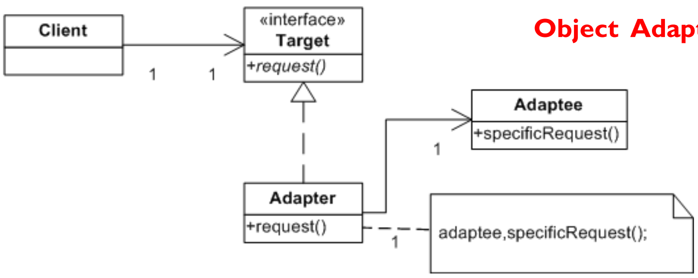
Motivation



Adapter

Structure

Object Adapter



Target (Shape) defines the domain-specific interface that Client uses.

Client (DrawingEditor) collaborates with objects conforming to the Target interface.

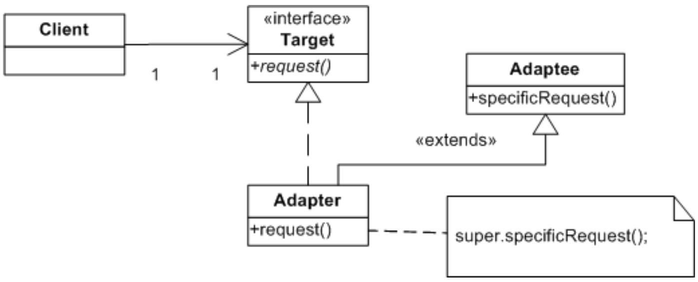
Adaptee (TextView) defines an existing interface that needs adapting.

Adapter (TextShape) adapts the interface of Adaptee to the Target interface.

Adapter

Structure

Class Adapter



Target (Shape) defines the domain-specific interface that Client uses.

Client (DrawingEditor) collaborates with objects conforming to the Target interface.

Adaptee (TextView) defines an existing interface that needs adapting.

Adapter (TextShape) adapts the interface of Adaptee to the Target interface.

Adapter

Benefits

- **Adaptees** added/used **without changes** to **existing code**.

Liabilities

- The overall **complexity** of the code **increases** because we add new classes and a level of indirection.

Composite

Composite

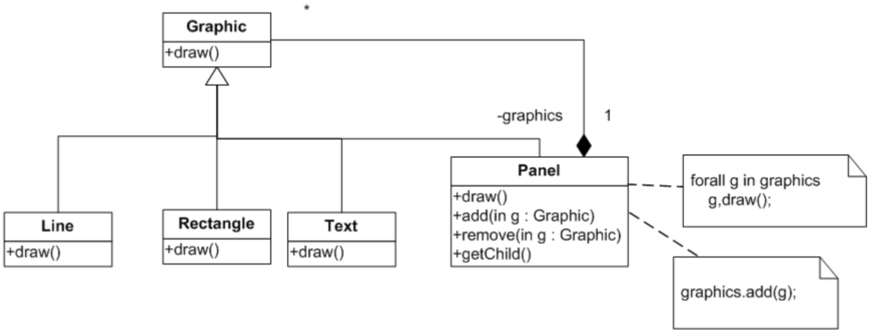
Intent

Compose objects into **structures** to represent **part-whole hierarchies**. Composite lets **clients treat** individual **objects** and **compositions** of objects **uniformly**.

Composite

Motivation

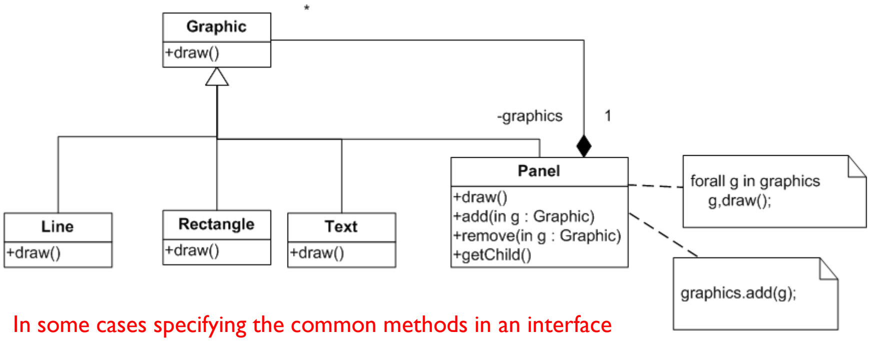
To realize the graphics….



Composite

Motivation

To realize the graphics….



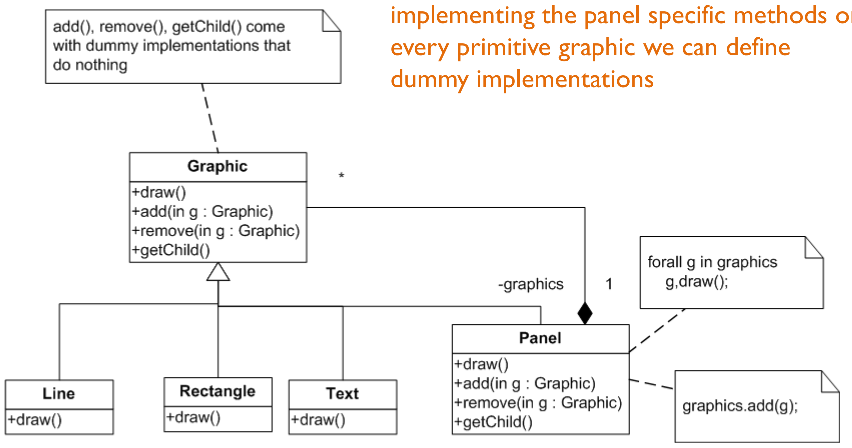
In some cases specifying the common methods in an interface may not be enough.

E.g. a client class has a list of Graphic and wants to add(graphic) in all the panels of this list … In this case the client has to check every list element to see if it is a composite panel ….

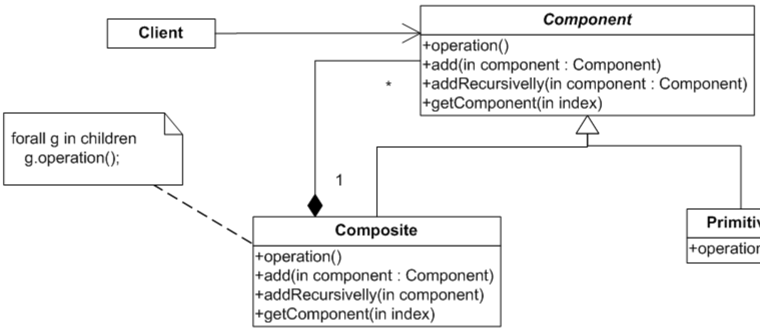
Composite

Motivation

To solve this problem we can maximize the common interface of primitive and composite objects…. Make it the union of the provided methods … to avoid implementing the panel specific methods on every primitive graphic we can define dummy implementations



Composite



Structure

Component (Graphic) declares the interface for objects in the composition; implements default behavior for the interface common to all classes.

Primitive (Rectangle, Line,Text, etc.) represents leaf objects in the composition.

Composite (Picture) defines behavior for components having children.

Client manipulates objects in the composition through the Component interface.

Composite

Benefits

- Makes the **clients simple**. Clients can treat composite structures and individual objects uniformly.
- Makes it **easier** to **add new kinds of components**. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.

Liabilities

- Can make your **design too general**. The disadvantage of making it easy to add new components is that it makes it **harder to restrict the components** of a **composite.**

Decorator

Decorator

Intent

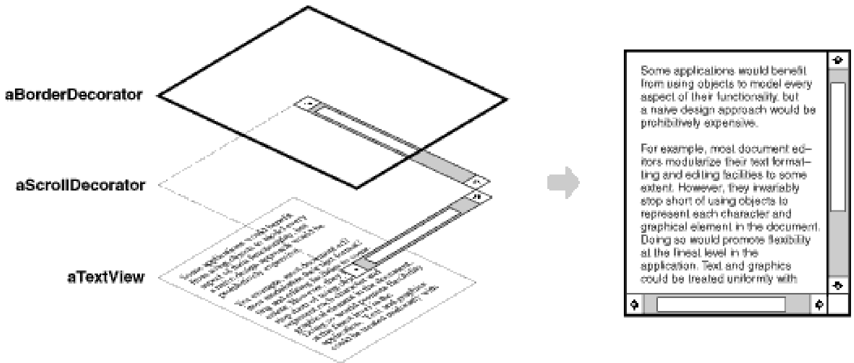
Decorators provide a flexible **alternative to subclassing** for extending functionality.

… lets us **attach new behaviors/features** to objects by placing these objects **inside special wrapper objects** that **contain the new behaviors/features**.

Decorator

Motivation

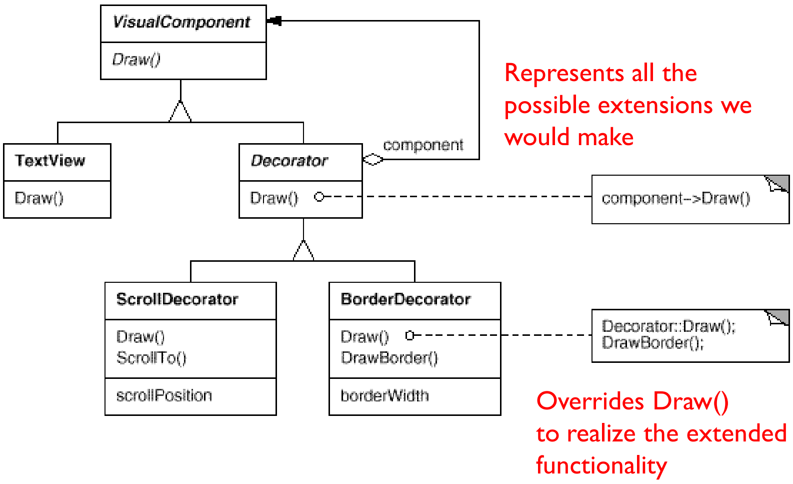
Inheritance is static: from the moment you create an object you cannot add/remove Functionalities/features



With inheritance cannot add functionalities dynamically – after the object creation

Decorator

Motivation



Decorator

Intent

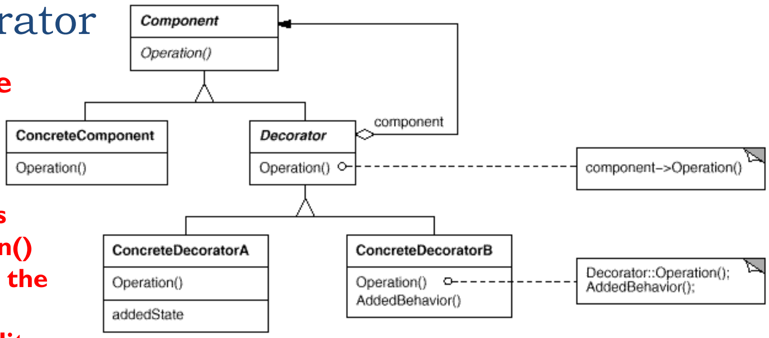
Decorators provide a flexible **alternative to subclassing** for extending functionality.

Applicability

- to add responsibilities to individual objects dynamically.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
- When a class definition may be hidden or otherwise unavailable for subclassing.

Decorator

Structure



Overrides Operation() to realize the extended functionality

Component (VisualComponent) defines the interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent (TextView) defines a class of objects to which additional responsibilities can be attached.

Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface.

ConcreteDecorator (BorderDecorator, ScrollDecorator) adds responsibilities to the component.

Decorator

Benefits

- More **flexibility** than **static inheritance**. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance.
  - With decorators, **responsibilities** can be added and removed at **run-time** simply by **placing objects inside decorators**.
  - In contrast, **inheritance** requires **creating a new class** for **each additional responsibility** (and **combinations** of responcibilities). This gives rise to many classes and increases the complexity of a system.
- Also avoids **feature-laden classes** high up in the hierarchy.
  - **Instead of trying to support all foreseeable features** in a **complex, customizable class**, we can define a simple class and add functionality incrementally with Decorator objects.

Liabilities

- **Lots of small objects**. Although these systems are easy to customize by those who understand them, they can be **hard to learn and debug**.

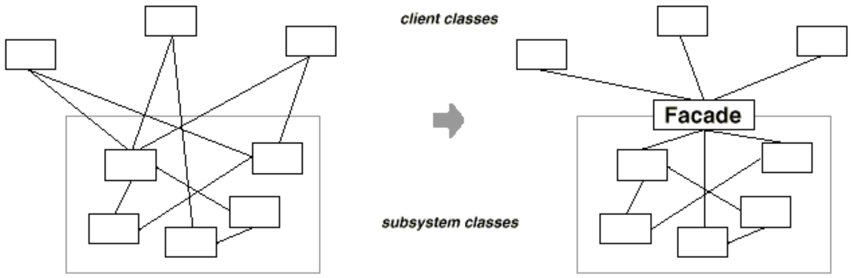
Façade

Façade

Intent
Provide **a unified interface** to a set of interfaces in a **subsystem**. Facade defines a **higher-level interface** that makes the subsystem **easier to use**.

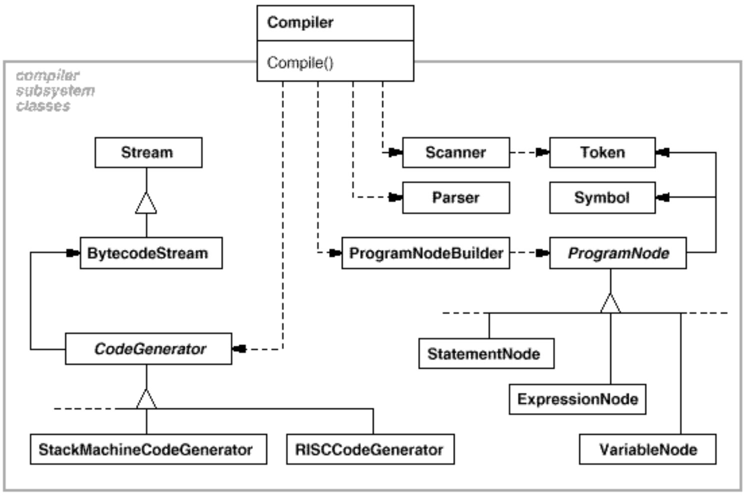
Façade

Motivation



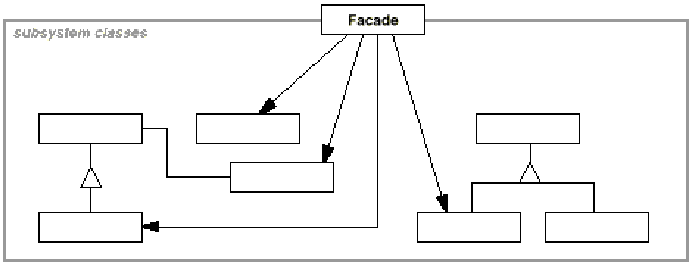
Façade

Motivation



Façade

Structure



Facade (Compiler) knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.

Subsystem classes (Scanner, Parser, ProgramNode, etc.)
implement subsystem functionality.
handle work assigned by the Facade object.
have no knowledge of the facade; that is, they keep no references to it.

Facade

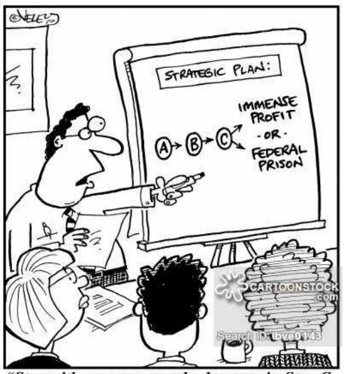
Benefits

- Façade shields clients from subsystem components, thereby reducing the number of objects that clients deal with and **making the subsystem easier to use**.
- Façade **doesn't prevent** applications from **using subsystem classes** if they need to.

Liabilities

- Façade can be become a **God (complex) class.**

Common behavioral patterns



"Stay with me now, people, because in Step C, things get a bit delicate."

Strategy

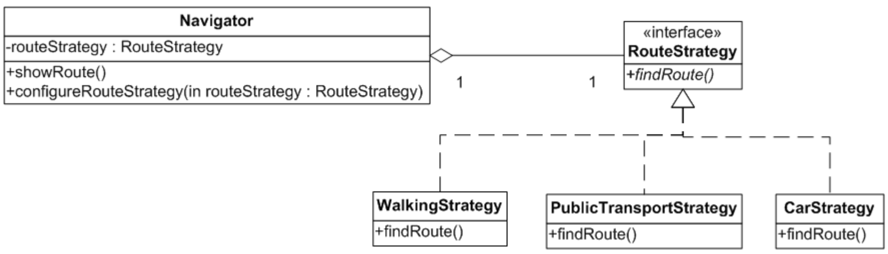
Strategy

Intent
Define a **family of algorithms**, **encapsulate each one**, and make them **interchangeable**.

Strategy **lets the algorithm vary independently** from **clients** that use it.

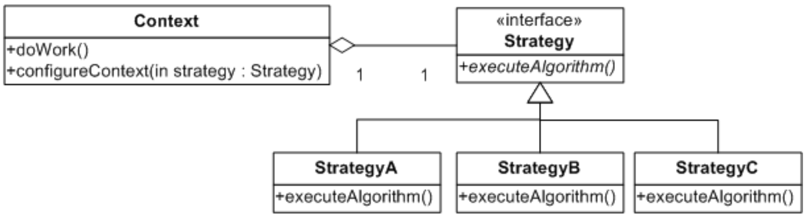
Strategy

Motivation



Strategy

Structure



Strategy (RouteStrategy) declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

StrategyA, StrategyB, …. (WalkingStrategy, CarStrategy, …) implements the algorithm using the Strategy interface.

Context (Navigator) is **configured** with a ConcreteStrategy object. Maintains a reference to a Strategy object. May define an interface that lets Strategy access its data.

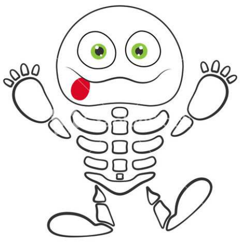
Strategy

Benefits

- We can **isolate Context from the implementation details** of an **algorithm**.
- We can **add new strategies without having to change Context**.
- We can **avoid complex conditionals** that realize the alternative algorithms in **Context**.
- We can **swap algorithms** used inside a **Context object** at **runtime**.

Liabilities

- Specifying a **common interface** for different algorithms **may not be easy**.
- Strategies increase the **number of objects** in an application.



Template Method

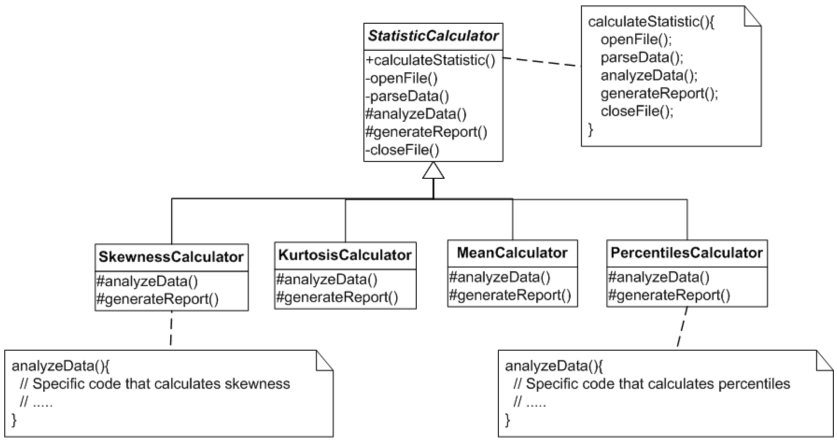
Template Method

Intent

Define the **skeleton** of an **algorithm** in an **operation**, **deferring some steps** to **subclasses**. Template Method lets **subclasses redefine certain steps** of an algorithm **without changing the algorithm's structure**.

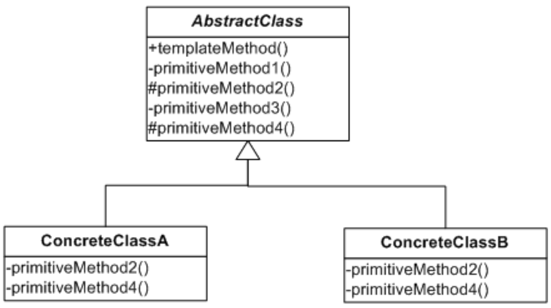
Template Method

Motivation



Template Method

Structure



AbstractClass (StatisticCalculator) defines **abstract primitive operations that concrete subclasses** define to implement steps of an algorithm. Implements a **template method** defining the **skeleton** of an algorithm.

The **template method** **calls primitive operations** as well as **abstract operations** defined in AbstractClass or those of other objects.

ConcreteClass (KurtosisCalculator, …) implements the primitive operations to carry out subclass-specific steps of the algorithm.

Template Method

Benefits

- **Template methods** are a **fundamental** technique for **code reuse**. They are particularly important in class libraries, because they are the means for **factoring out common behavior** and reduce **duplication**.

Liabilities

- To reuse an abstract class effectively, **subclass writers** must **understand** which **operations** are designed for **overriding** and **how** this should be done.

Command

Command

Intent

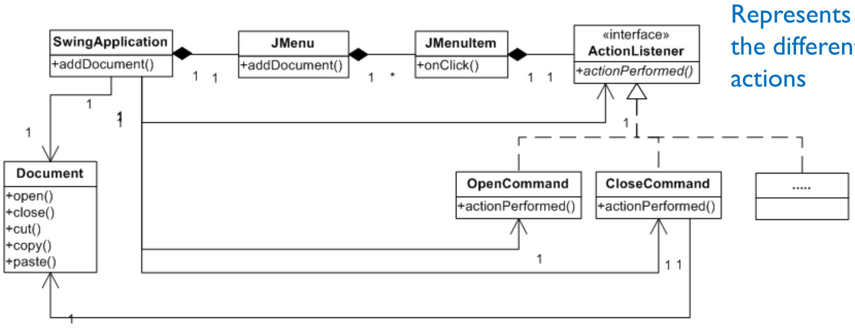
Encapsulate an **action** as an **object**, thereby letting you **parameterize clients** with **different actions**, **queue** or **log** actions, and support **undoable/redoable actions.**

Command

Motivation

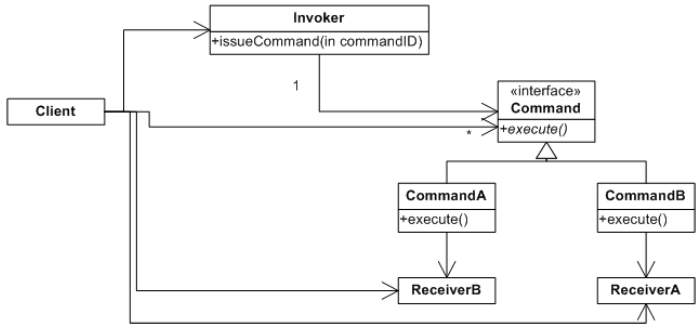
Every menu item should do a diff. action

Represents all the different actions



Command

Structure



Command declares an interface for executing an action.

ConcreteCommand (PasteCommand, OpenCommand) defines a binding between a Receiver object and an action. implements Execute by invoking the corresponding operation(s) on Receiver.

Client (Application) creates a ConcreteCommand object and sets its receiver.

Invoker (JMenuItem) asks the command to carry out the request.

Receiver (Document, Application) knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Command

Benefits

- Command **decouples** the object that invokes the operation from the one that knows how to perform it.
- We can **assemble** commands into a **composite command**, **queue** or **log** commands
- It's easy to **add new Commands** because you **don't have to change existing classes**.

Liabilities

- The code may become more complicated since you're introducing a **new layer** between **invokers** and **receivers**.
- Specifying a **common interface** for **different commands** may not be easy.

Observer

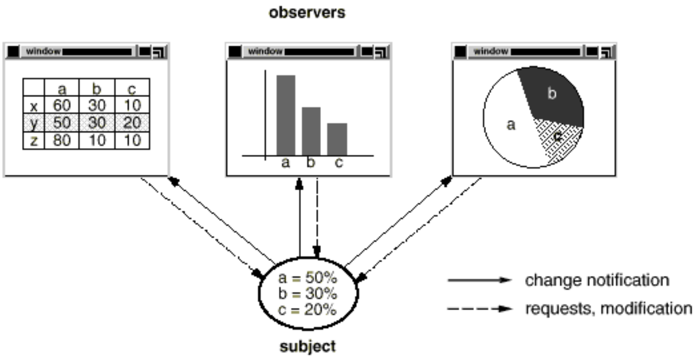
Observer

Intent

Define a **one-to-many association** between objects so that when one object **changes state**, all its **dependents** are **notified** and updated automatically.

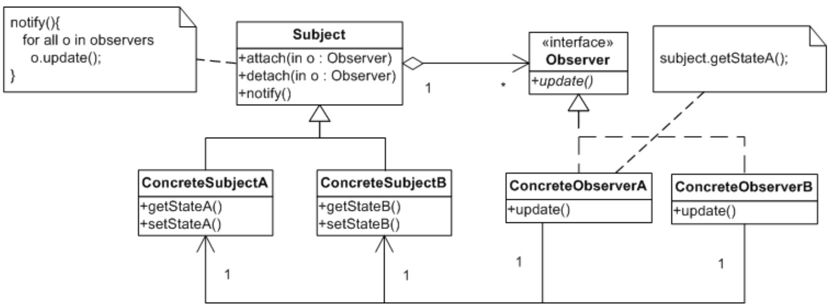
Observer

Motivation



Observer

Structure



Subject knows its observers. **Any number of Observer objects** may observe a Subject object. Subject Provides an **interface** for **attaching** and **detaching** Observer objects.
Observer defines an **updating interface** for objects that should be notified of changes in a subject.
ConcreteSubject stores state of interest to ConcreteObserver objects. Sends a notification to its observers when its state changes.
ConcreteObserver maintains a reference to a ConcreteSubject object. Stores state that should stay consistent with the subject's. Implements the Observer updating interface to keep its state consistent with the subject's.

Observer

Benefits

- The **coupling** between **subjects** and **observers** is abstract and minimal.
- We can introduce **new Observer classes without** having to **change** the **Subject class**.
- We can establish **relations** between objects at **runtime**.

Liabilities

- **Unexpected/unwanted notifications.**

Software design quality

How do we assess the quality of a software design?

Software design quality

"...And that, in simple terms, is what's wrong with your software design."

Software design **reviews**, informal or formal techniques, to determine the quality of design artifacts.

Software **metrics** to quantify the assessment.

The CK metrics suite is a well known set of metrics for OO software.

Coupling

Coupling Between Object classes (CBO) **[Chidamber & Kemerer]**

CBO(A) = number of classes used by A (inheritance is typically not counted)

Coupling Factor (COF) **[Abreu, Esteves, Goulao]**

S = {c1, c2, ….cN}

COF(S) = Sum (isClient(ci, cj))/ (N*(N-1))

isClient : S*S → {0, 1}

isClient(ci, cj)) = 1 if ci uses cj else isClient(ci, cj)) = 0

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;

    public Controller(Bell ab, Light al){
     alarm = false;
     b = ab;
     l = al;
    }
    public void alarmSignal(){
     alarm = true;
    }
    public void cancelAlarm(){
     alarm = false;
     System.out.println("False alarm !!");
    }
    …………………………….
}
```

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;
    ……………………………….
    public void confirmAlarm (){
     if (alarm == true) {
         if(b != null) b.ring();
         if(l!=null) l.open();
         alarm = false;
     } else
         System.out.println("False alarm !!");
    }
    public void stopAlarm(){
      if(b != null) b.stop();
      if(l!=null) l.close();
    }
}
```

CBO(Controller) = 2
COF(Controller, Bell, Light) = (2+0+0)/(3*2) = 0.33

Cohesion

Lack of Cohesion of Methods (LCOM)

LCOM **[Chidamber & Kemerer]**

Q set contains the pairs of class methods that use common attributes

P set contains the pairs of class methods that don't use common attributes

LCOM(x) = |P| – |Q| αν |P| > |Q|,

LCOM(x) = 0 αν |P| <= |Q|

```
public class Rectangle {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double ax, double ay,
                     double aWidth, double aHeight){
     x = ax;
     y = ay;
     width = aWidth;
     height = aHeight;
    }

    ……………………………………
}
```

```
public class Rectangle {
    ……………………………………..

    public void draw(){
     System.out.println("Drawing Rectangle at : (" +
        x + "," + y +
        ") with width = " + width + " height = " + height);
     // drawing code ....
    }

    public double calculateArea(){
     return width * height;
    }
}
```

Q = 3, P = 0 => LCOM = 0

Cohesion

LCOM2(x) = 1 – Sum(ma_i)/(m*a) **[Henderson-Sellers, Constantine, Graham]**

m = number of methods

a = number of attributes

ma_i = number of methods that use a_i, i=1, …, a

maximum of ma_i is m

LCOM2 the smaller the better…..

LCOM2 = 1 ?

LCOM2 = 0 ?

```
public class Rectangle {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double ax, double ay,
                     double aWidth, double aHeight){
     x = ax;
     y = ay;
     width = aWidth;
     height = aHeight;
    }

    ……………………………………
}
```

```
public class Rectangle {
    ……………………………………..

    public void draw(){
     System.out.println("Drawing Rectangle at : (" +
        x + "," + y +
        ") with width = " + width + " height = " + height);
     // drawing code ....
    }

    public double calculateArea(){
     return width * height;
    }
}
```

$m = 3, a = 4$

$ma_x = 2$

$ma_y = 2$

$ma_{width} = 3$

$ma_{height} = 3$

LCOM2 = 1 – 10/12 = 1/6 = 0.166

Cohesion

LCOM3 = (m - Sum(ma_i)/a) / (m – 1) [Henderson-Sellers, Constantine, Graham]

the smaller the better…..

= 0 perfect

= 1 bad

> 1 dead attributes

LCOM3 = 0 ??

why dead attributes if LCOM3 > 1 ?

```
public class Rectangle {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double ax, double ay,
                       double aWidth, double aHeight){
     x = ax;
     y = ay;
     width = aWidth;
     height = aHeight;
    }

    .......................................
}
```

```
public class Rectangle {
    ………………………………………..

    public void draw(){
     System.out.println("Drawing Rectangle at : (" +
        x + "," + y +
        ") with width = " + width + " height = " + height);
     // drawing code ....
    }

    public double calculateArea(){
     return width * height;
    }
}
```

$m = 3, a = 4$

$ma_x = 2$

$ma_y = 2$

$ma_{width} = 3$

$ma_{height} = 3$

LCOM3 = (3-10/4)/(3-1) = 0.25

Class complexity

Weighted Methods per Class (WMC) [Chidamber & Kemerer]

WMC(A) = Sum(Ci), i=1, .. N methods

Ci = method i complexity

➔ Ci can be measured in various ways:
- Lines of Code (LOC)
- McCabe (number of conditions +1)
  - if, for, while → 1 condition
  - Switch ➔ transform to if conditions first because it depends on how switch is implemented…

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;

    public Controller(Bell ab, Light al){
     alarm = false;
     b = ab;
     l = al;
    }
    public void alarmSignal(){
     alarm = true;
    }
    public void cancelAlarm(){
     alarm = false;
     System.out.println("False alarm !!");
    }
    …………………………….
}
```

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;
    …………………………………….
    public void confirmAlarm (){
     if (alarm == true) {
         if(b != null) b.ring();
         if(l!=null) l.open();
         alarm = false;
     } else
         System.out.println("False alarm !!");
    }
    public void stopAlarm(){
      if(b != null) b.stop();
      if(l!=null) l.close();
    }
}
```

McCabe version of WMC(Controller) = 10

Class complexity

Request For a Class (RFC) **[Chidamber & Kemerer]**

RFC(A) = M + R

M = number of class methods

R = number of methods called by the class methods (with each method counts once if called multiple times

➔ Usually we only count methods of the same project – we do not consider standard API calls and so on.

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;

    public Controller(Bell ab, Light al){
     alarm = false;
     b = ab;
     l = al;
    }
    public void alarmSignal(){
     alarm = true;
    }
    public void cancelAlarm(){
     alarm = false;
     System.out.println("False alarm !!");
    }
    …………………………….
}
```

```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;
    …………………………………….
    public void confirmAlarm (){
     if (alarm == true) {
         if(b != null) b.ring();
         if(l!=null) l.open();
         alarm = false;
     } else
         System.out.println("False alarm !!");
    }
    public void stopAlarm(){
      if(b != null) b.stop();
      if(l!=null) l.close();
    }
}
```

RFC(Controller) = 5 + 4

Reuse vs complexity

Depth of Inheritance Tree (DIT) **[Chidamber & Kemerer]**

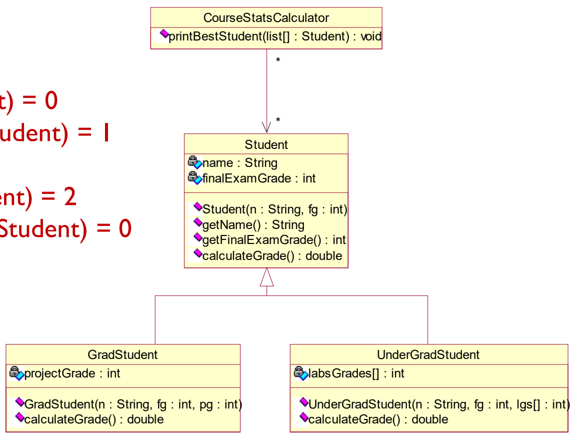
DIT(A) = depth of A in the tree

Number of Children (NOC) **[Chidamber & Kemerer]**

NOC(A) = number of subclasses A has

DIT(Student) = 0
DIT(GradStudent) = 1

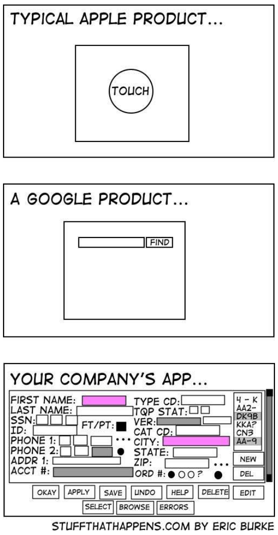
NOC(Student) = 2
NOC(GradStudent) = 0



User interface design

What are the fundamental UI design principles?

UI design principles



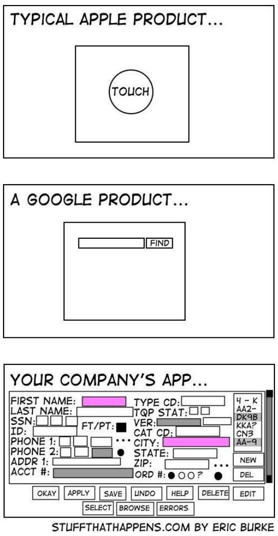
Learnability.
The software should be easy to learn so that the user can rapidly start working with the software.

User familiarity.
The interface should use terms and concepts drawn from the experiences of the people who will use the software.

Consistency.
The interface should be consistent so that comparable operations are activated in the same way.

Minimal surprise.
The behavior of software should not surprise users.

UI modalities



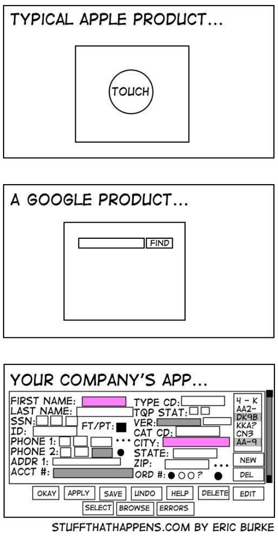
Recoverability.
The interface should provide mechanisms allowing users to recover from errors.

User guidance.
The interface should give meaningful feedback when errors occur and provide context-related help to users.

User diversity.
The interface should provide appropriate interaction mechanisms for diverse types of users and for users with different capabilities (blind, poor eyesight, deaf, colorblind, etc.).

How should the user interact with the software?

UI modalities

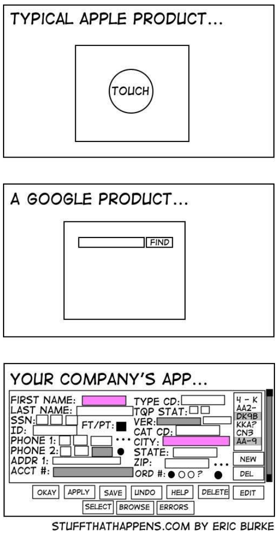


Question-answer.
The interaction is essentially restricted to a single question-answer exchange between the user and the software.

Direct manipulation.
Users interact with objects on the computer screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or a finger on touch screens) that manipulates an object and invokes actions that specify what is to be done with that object.

Menu selection.
The user selects a command from a menu list of commands.

UI modalities



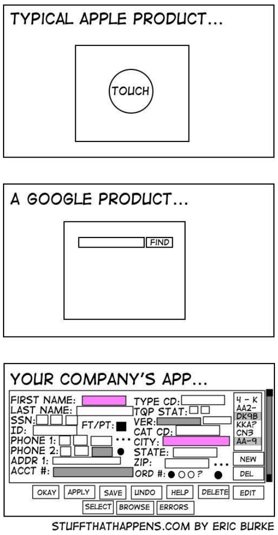
Form fill-in.
The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.

Command language.
The user issues a command and provides related parameters to direct the software what to do.

Natural language.
The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated into software commands.

How should information from the software be presented to the user?

UI information presentation



Limit the number of colors used.

Use color change to show the change of software status.

Use color-coding to support the user's task.

Use color-coding in a thoughtful and consistent way.

Use colors to facilitate access for people with color blindness or color deficiency (e.g., use the change of color saturation and color brightness, try to avoid green and red combinations).

Don't depend on color alone to convey important information to users with different capabilities.