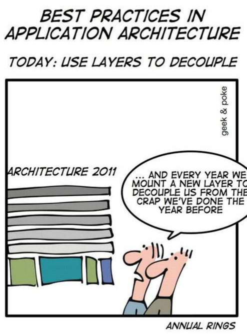


Software Design

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

M. Fowler. Patterns of Enterprise Application Architecture

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

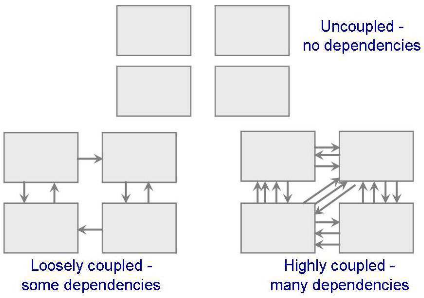
Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe component interactions.

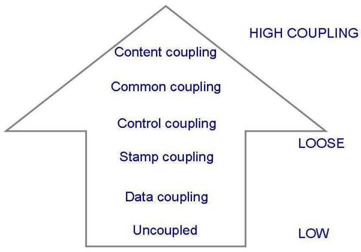
Usually the goal is to place different functionalities and responsibilities in different components.

What makes a good split?

Low coupling

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

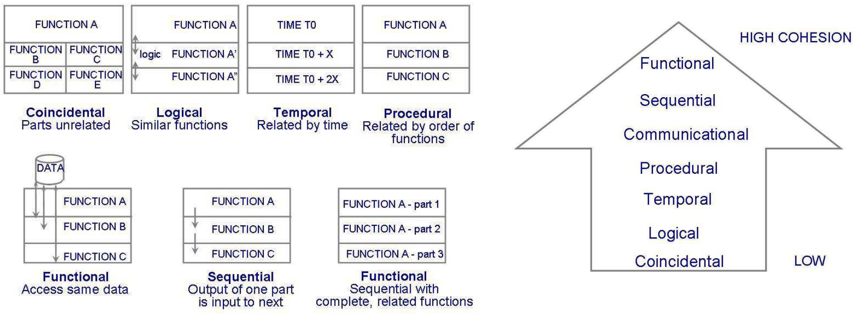




W. P. Stevens, G. J. Myers and L. L. Constantine, Structured design, *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.

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

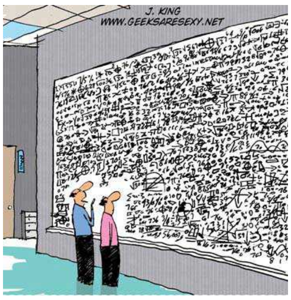
Abstraction is generally defined as a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information.

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.

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)

```
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;
    }
}
```

$m = 3, a = 4$

$ma_x = 2$

$ma_y = 2$

$ma_{width} = 3$

$ma_{height} = 3$

$LCOM2 = 1 – 10/12 = 1/6 = 0.166$

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 !!");
    }
    …………………………….
}
```

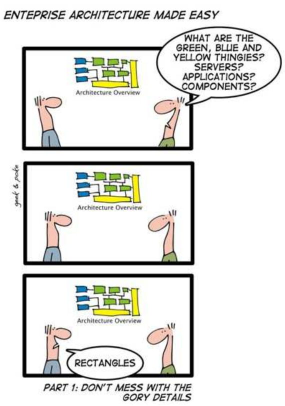
```
class Controller {
    private boolean alarm;
    private Bell b;
    private Light l;
    …………………………………….
    public void confirmAlarm (){
     if (alarm == true) {
         if(b != null) b.ring();
         if(l!=null) l.open();
         alarm = false;
     } else
         System.out.println("False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

McCabe version of WMC(Controller) = 10

Software Structure & Architecture

What do we mean by software architecture?

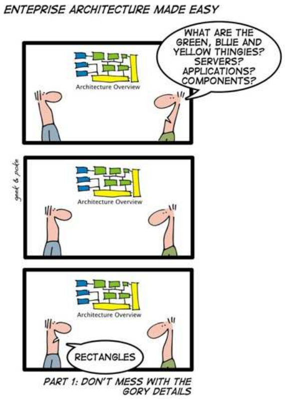
What do we mean by software architecture?



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.**

Design patterns

What do we mean by pattern?

What do we mean by design pattern?

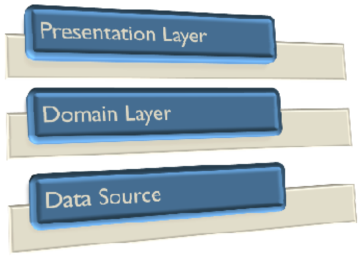
Christopher Alexander says,

"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"

Fowler's Enterprise Application Architecture (EAA) Patterns

How does a typical EAA look like?

How does a typical EAA look like?



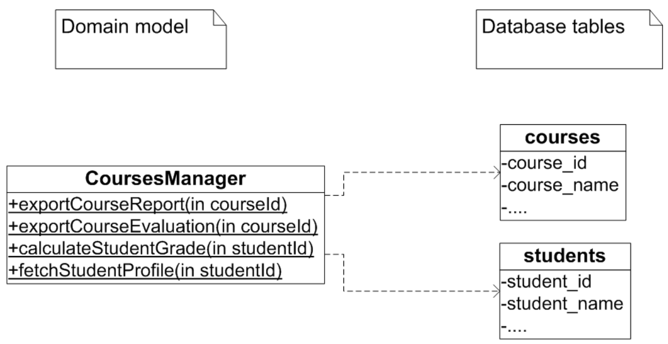
Layer	Responsibilities
Presentation	Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
Domain	Logic that is the real point of the system
Data Source	Communication with databases,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

Domain Logic Patterns

How do we organize Domain layer logic?

Transaction Script

***Organizes domain logic* by *procedures* where each procedure handles a *single request* from the *presentation*.**



Transaction Script

***Organizes domain logic** by **procedures** where each procedure handles a **single request** from the **presentation**.*

Most business applications can be thought of as a series of transactions. A transaction may view some information as organized in a particular way, another will make changes to it.

A **Transaction Script** organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script.

Suitable for cases where the business logic is simple.

But, transaction logic may get complex for complex apps

Code duplication in the case of more complex business logic.

Table Module

A single instance that handles the business logic for all rows in a database table or view.

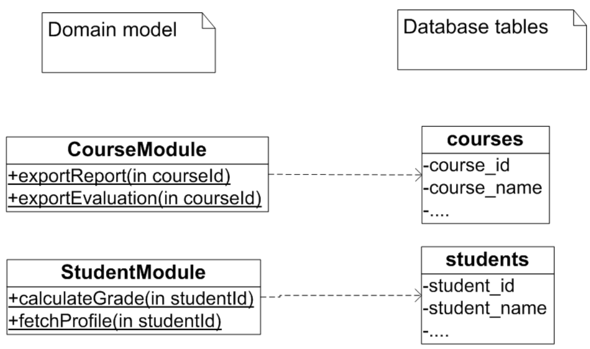


Table Module

A single instance that handles the business logic for all rows in a database table or view.

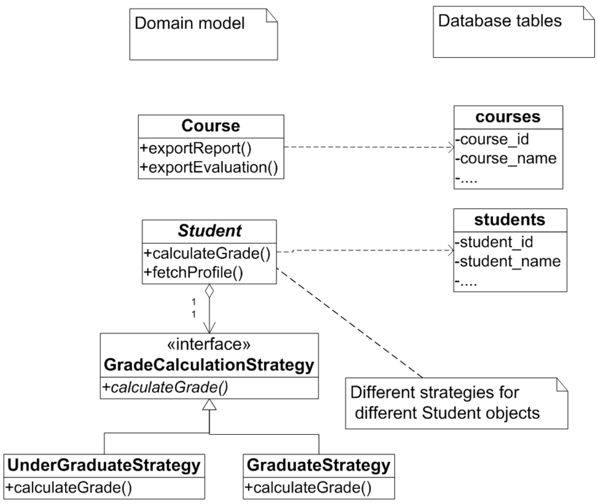
A Table Module organizes **domain logic** with **one class per table** in the database, and a single instance of a class contains the various procedures that will act on the data.

The primary **distinction** with Domain Model is that, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.

The strong point of Table Module is the **easy integration** with the Data Layer. However, you **do not have the full power of OO** that the Domain Model pattern gives you.

Domain Model

An object model of the domain that incorporates both behavior and data.



Domain Model

An object model of the domain that incorporates both behavior and data.

At its worst business logic can be very complex.

Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to deal with.

A **Domain Model** creates a web of interconnected objects, where each object represents some **meaningful concept**.

Good for real world complex business logic.
May be too much for very simple cases; better use Transaction scripts, instead.

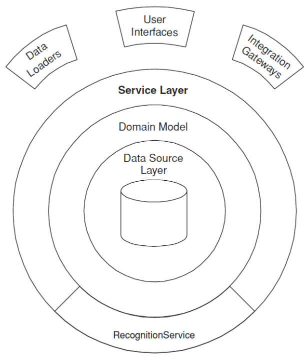
Service Layer

Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

Enterprise applications typically require different kinds of interfaces to the data: HTML, REST, others.

Despite their different purposes, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic.

Encoding the logic of the interactions separately in each interface causes a lot of duplication..



If an application has or will have to support different types of clients with different interfacing requirements.

Data Source Architectural Patterns

How do we transfer data from/to the Data layer to/from the Domain Layer ?

Table Data Gateway

An object that acts as a gateway to a database table. One instance handles all the rows in the table.

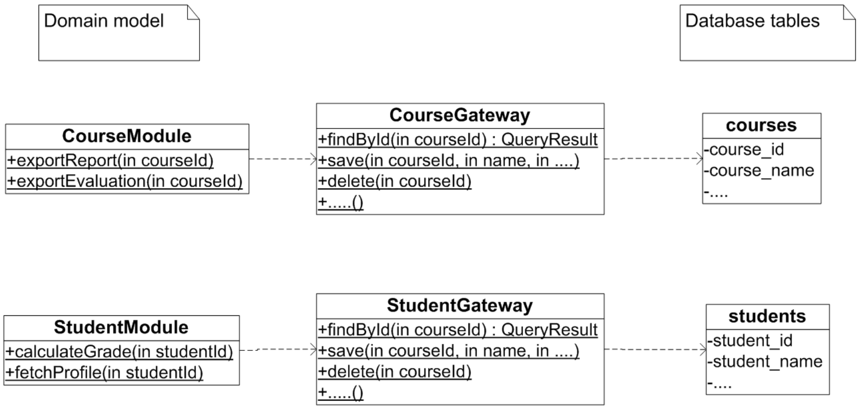


Table Data Gateway

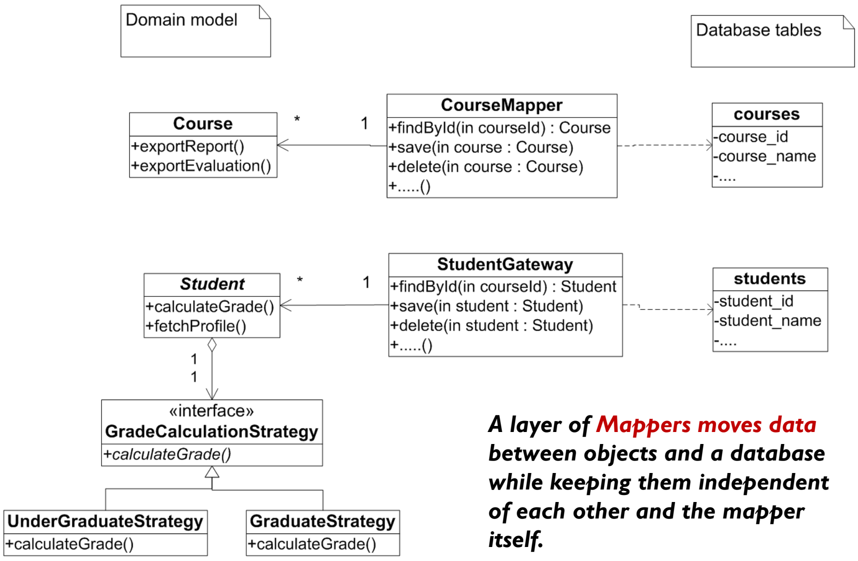
An object that acts as a gateway to a database table. One instance handles all the rows in the table.

Mixing SQL in application logic can cause several problems. Many developers aren't comfortable with SQL, and many who are comfortable may not write it well.

A Table Data Gateway holds all the SQL for accessing a single table or view. Other code calls its methods for all interaction with the database.

It is a simple way to transfer data from/to a table. Fits well with Table Module (better than with Domain Model).

Data Mapper



A layer of Mappers moves data between objects and a database while keeping them independent of each other and the mapper itself.

Data Mapper

A layer of Mappers moves data between objects and a database while keeping them independent of each other and the mapper itself.

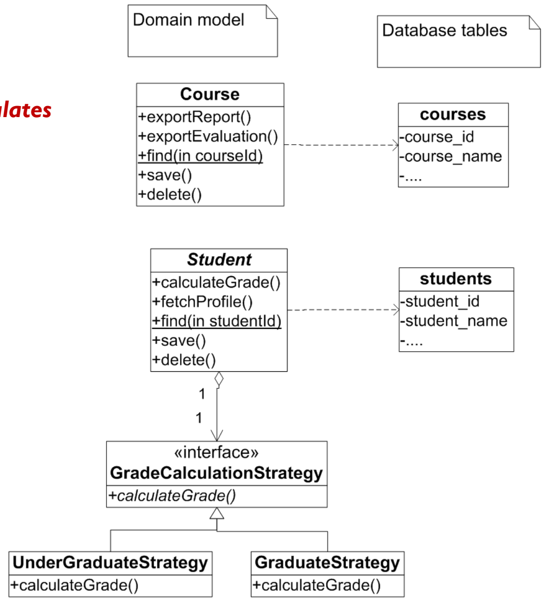
The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other.

With Data Mapper the in-memory objects needn't know even that there's a database present.

Fits well with Domain Model.

Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.



Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

Active Record uses the most **obvious** know approach, **putting data access logic in the domain object**. This way all people know hoe to read and write their data to and from the database.

Fits well with **SIMPLE** Domain Models. Adds responsibilities to domain classes

Object Relational Structural Patterns

How to map OO - Relational concepts ?

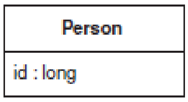
Identity Field

Saves a database ID field in an object to maintain identity between an in-memory object and a database row.

Intent

Reading data from a database is all very well, but in order to **write data back you need to tie the database to the in-memory object system**.

In essence, ***Identity Field*** *is mind-numbingly simple. All you do is store the primary key* of the relational database table in the object's fields.



Fits together with Domain Model

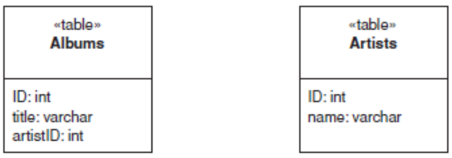
Foreign Key Mapping

Maps an association between objects to a foreign key reference between tables.

Objects can refer to each other directly by object references. To **save** these **objects** to a **database**, it's vital to **save these references**.

A ***Foreign Key Mapping maps*** *an* ***object reference*** *to a* ***foreign key*** *in the* database.

Single object reference →
Many to one / one to one



A *Foreign Key Mapping can be used for almost all associations between classes.* The most common case where it isn't possible is with many-to-many associations. Instead you need to use *Association Table Mapping*

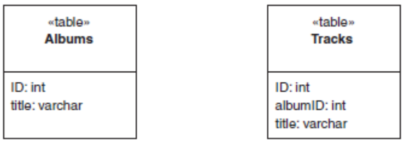
Foreign Key Mapping

***Maps** an **association** between **objects** to a **foreign key** reference **between tables**.*

Objects can refer to each other directly by object references.
To **save** these **objects** to a **database**, it's vital to **save these references**.

A ***Foreign Key Mapping maps*** *an **object reference** to a **foreign key** in the* database.

List of references → One to many



A *Foreign Key Mapping can be used for almost all associations between classes*. The most common case where it isn't possible is with many-to-many associations. Instead you need to use *Association Table Mapping*

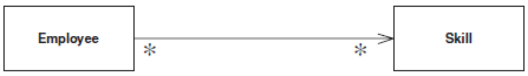
Association Table Mapping

Saves an association as a table with foreign keys to the tables that are linked by the association.

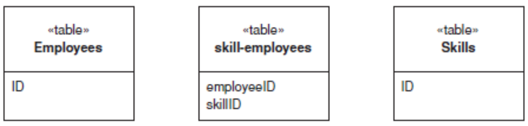
When you're mapping a one-to-many association you can handle this using ***Foreign Key Mapping*** *essentially using a foreign key to the single-valued* end of the association.

But a many-to-many association can't do this because there is no single-valued end to hold the foreign key.

The answer is the classic resolution that's been used by relational data people for decades: create an extra table to record the relationship.



List of references → Many to many



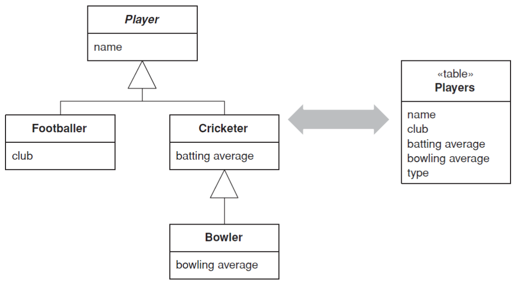
The canonical case for *Association Table Mapping is a many-to-many association*, since there are really no any alternatives for that situation.

Single Table Inheritance

Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.

Relational databases don't support inheritance, so when mapping from objects to databases we have to consider how to represent our nice inheritance structures in relational tables.

Single Table Inheritance *maps all fields of all classes* of an inheritance structure into a single table.

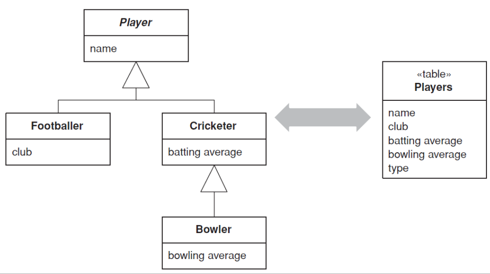


Single Table Inheritance

Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.

There's only a single table to worry about on the database. There are no joins in retrieving data. Any refactoring that pushes fields up or down the hierarchy doesn't change the DB.

Fields are sometimes relevant and sometimes not, which can be confusing. Columns used only by some subclasses lead to wasted space in the database. The single table may end up being too large.



Class Table Inheritance

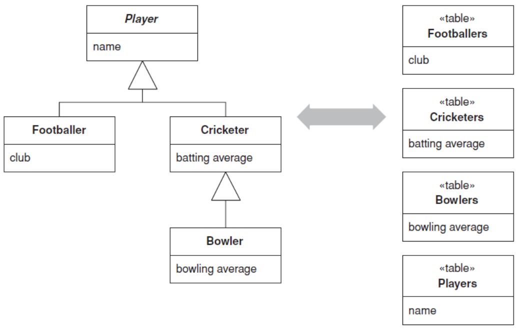
Represents an inheritance hierarchy of classes with one table for each class.

Relational databases don't support inheritance, so we have to consider how to represent our nice inheritance structures in relational tables.

Class Table Inheritance *supports this by using one database* ***table per class*** *in* the inheritance structure.

The **linking** between the tables can be done by assuming a **equality in the tables key values**.

Another way is to have **foreign keys** between the tables



Class Table Inheritance

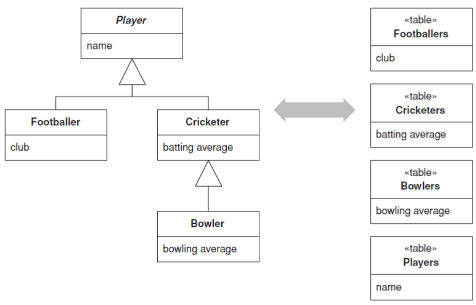
Represents an inheritance hierarchy of classes with one table for each class.

All columns are relevant for every row so tables are easier to understand and don't waste space.

The relationship between the domain model and the database is straightforward.

Need to touch multiple tables to load an object, which means a join or multiple queries and sewing in memory.

Any refactoring of fields up or down the hierarchy causes database changes.

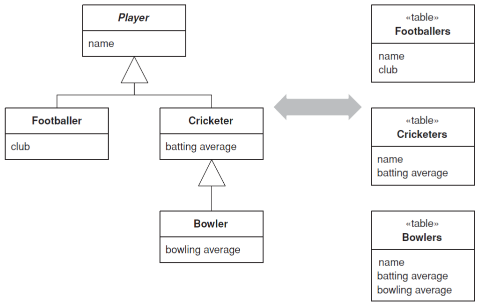


Concrete Table Inheritance

Represents an inheritance hierarchy of classes with one table per <u>leaf</u> class in the hierarchy with all fields in the table row.

Relational databases don't support inheritance, so when mapping from objects to databases we have to consider how to represent our nice inheritance structures in relational tables.

Concrete Table Inheritance *supports this by using one database* ***table per leaf class*** *in the* inheritance structure, *with* ***all fields*** *in the* ***table row***.



Concrete Table Inheritance

Represents an inheritance hierarchy of classes with one table per <u>leaf</u> class in the hierarchy with all fields in the table row.

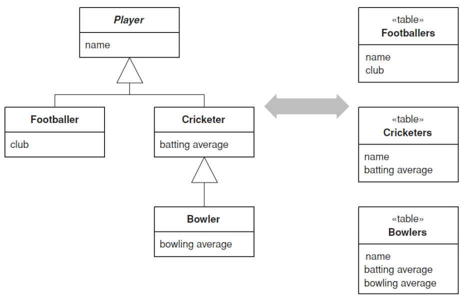
Each table is self-contained and has no irrelevant fields.

There are no joins to do when reading the data from the concrete mappers.

Each table is accessed only when that class is accessed, which can spread the access load.

Redundancy in the data

With fields are pushed up or down the hierarchy, you don't have to alter the table definitions. If a superclass field changes, you need to change each table.



Web presentation patterns

How to deal with the UI ?

Model View Controller

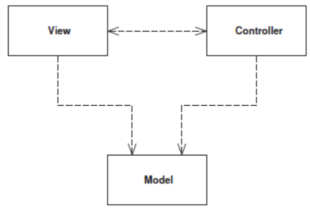
Splits user interface interaction into three distinct roles.

The **model** is an object that 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.



The separation of presentation and model is one of the most important design principles in software, and the only time we shouldn't follow it is in very simple systems where the model has no real behavior in it anyway.

Page Controller

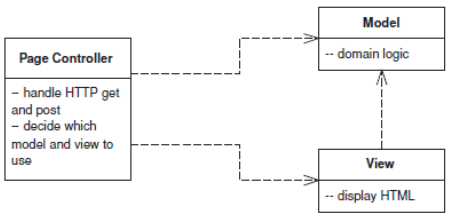
An object that handles a request for a specific page or action on a Web site.

The basic idea behind a **Page Controller** is to have one module on the Web server act as the controller for each page on the Web site.

In practice, it doesn't work out to exactly one module per page, since you may get a different page depending on dynamic information.

More strictly, the **controllers** tie in to **each action**, which may be clicking a link or a button.

Page Controller works particularly well in a site where most of the controller navigational logic is pretty simple.

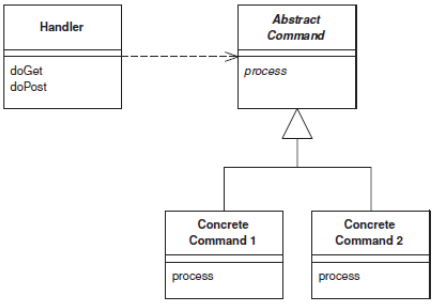


Front Controller

A controller that handles all requests for a Web site.

A **Front Controller** handles all calls for a Web site, and is usually structured in two parts: A Web handler and a command hierarchy.

The Web handler is the object that actually receives post or get requests from the Web server. It pulls just enough information from the URL and the request to decide what kind of action to initiate and then delegates to a command to carry out the action



Front Controller works particularly well in a site where the controller navigational logic is more complex.

Template View

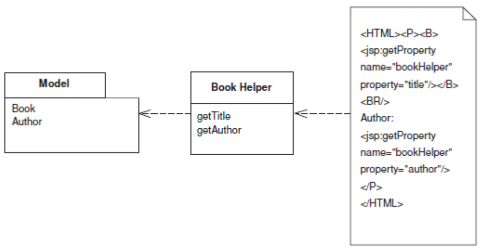
Renders information into HTML by embedding markers in an HTML page.

The basic idea of Template View is to embed markers into a static HTML page when it's written.

When the page is used to service a request, the markers are replaced by the results of some computation, such as a database query.

The strength of Template View is that it allows you to compose the content of the page by looking at the page structure.

But they are not easy to use with complex generation logic and they are not easy to test.

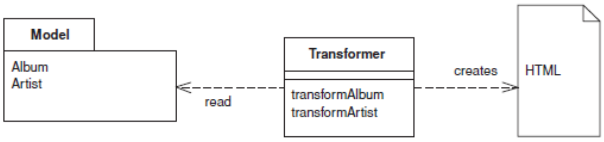


Transform View

A module that processes domain data element by element and transforms it into HTML.

The basic notion of Transform View is writing a program that looks at domain oriented data and converts it to HTML.

The program walks the structure of the domain data and, as it recognizes each form of domain data, it writes out the particular piece of HTML for it.

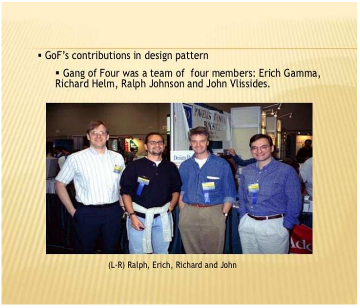


Transform View does not mix HTML with view generation logic and is easier to test.

However, several people may prefer to have everything embedded in one module.

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

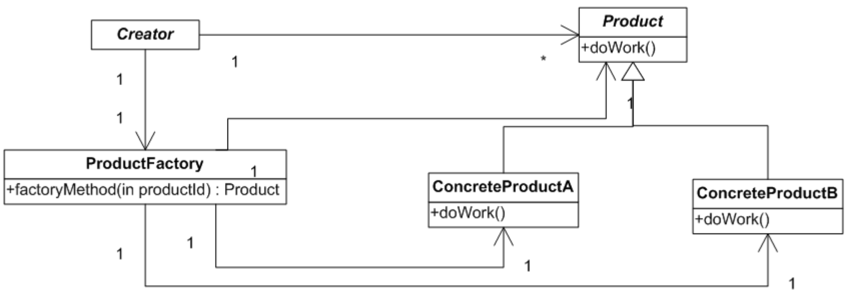
Creational	Structural	Behavioral
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

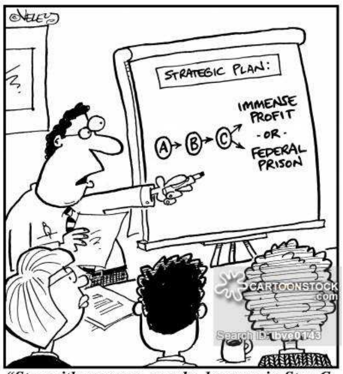
Parameterized Factory

Parameterized Factory

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.





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

Strategy

Strategy

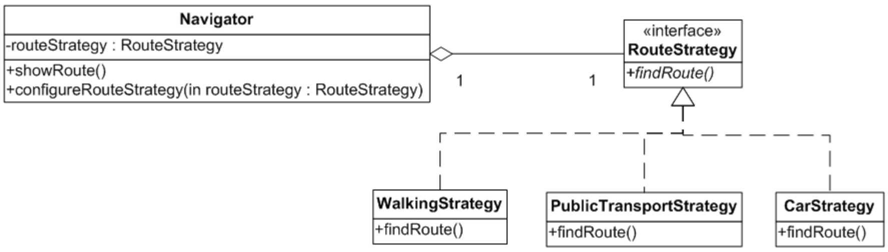
Intent

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

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

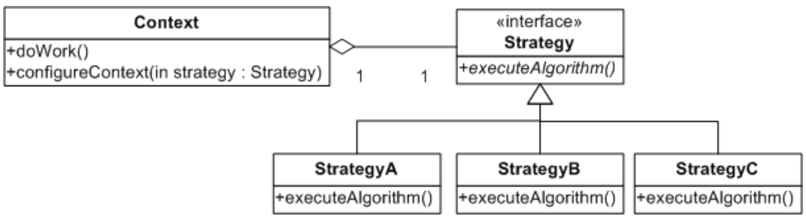
Strategy

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.

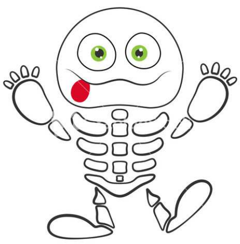
Strategy

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

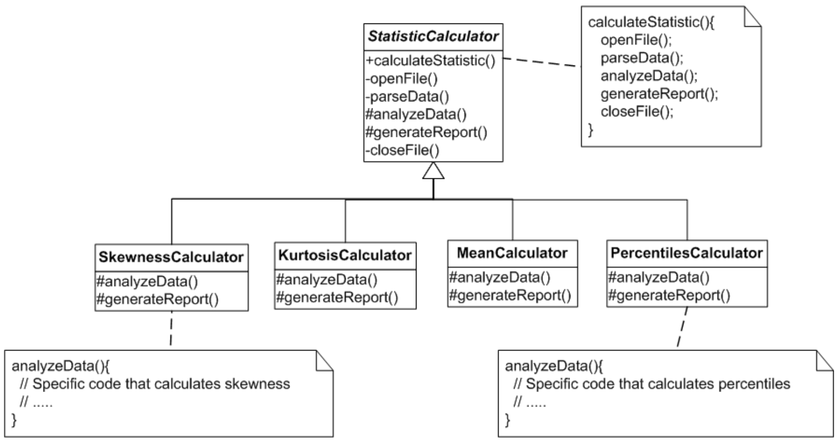
Template Method

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**.

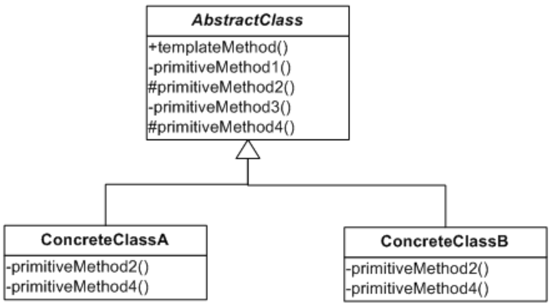
Template Method

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.

Adapter

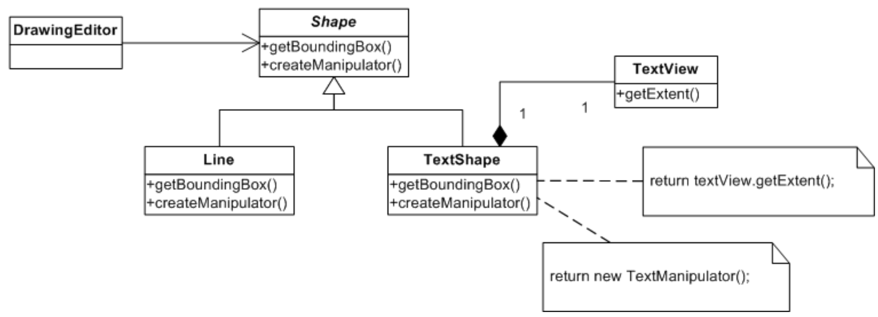
Adapter

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**.

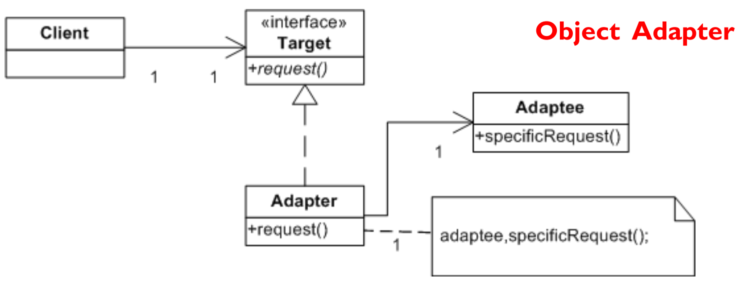
Adapter

Motivation



Adapter

Structure



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

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

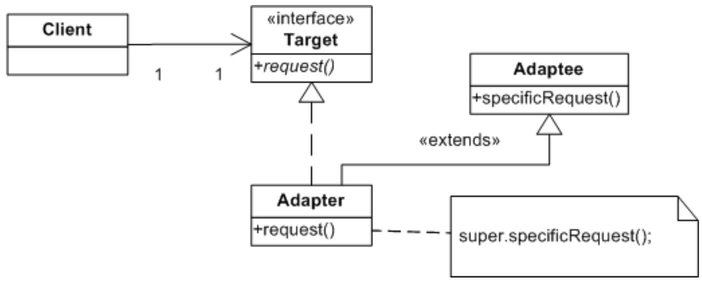
Adaptee (TextView) defines an existing interface that needs adapting.

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

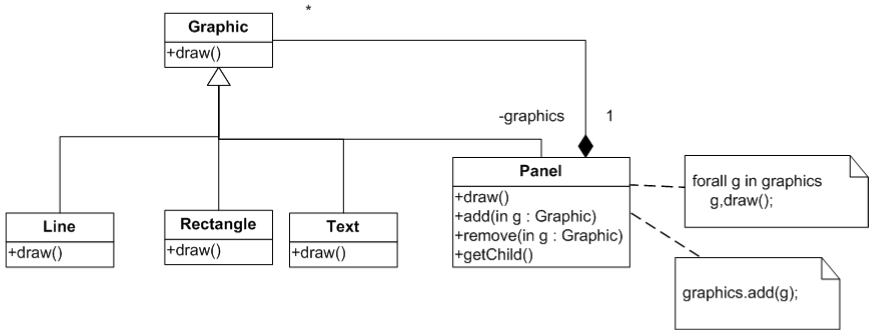
Intent

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

Composite

Motivation

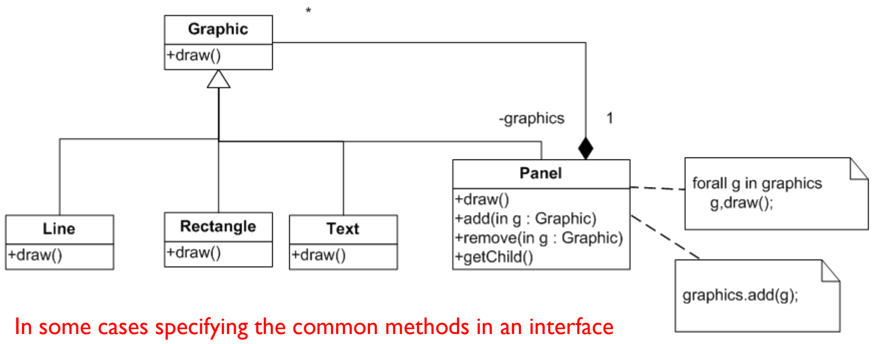
To realize the graphics….



Composite

Motivation

To realize the graphics….



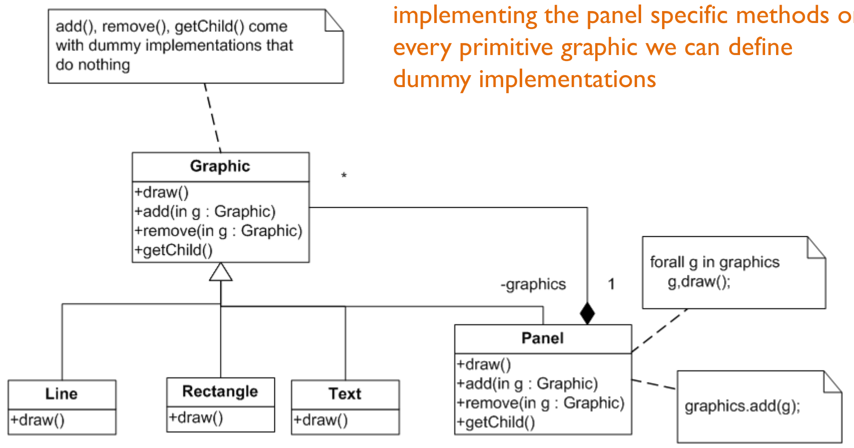
In some cases specifying the common methods in an interface may not be enough.

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 ….

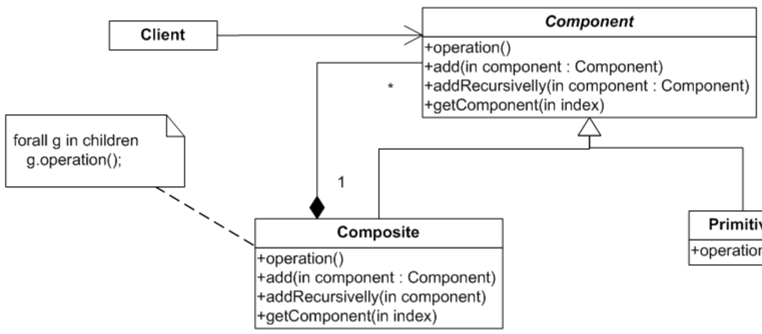
Composite

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.**

"Well, I'll be—we just spent all weekend planning to rob a façade."

Façade

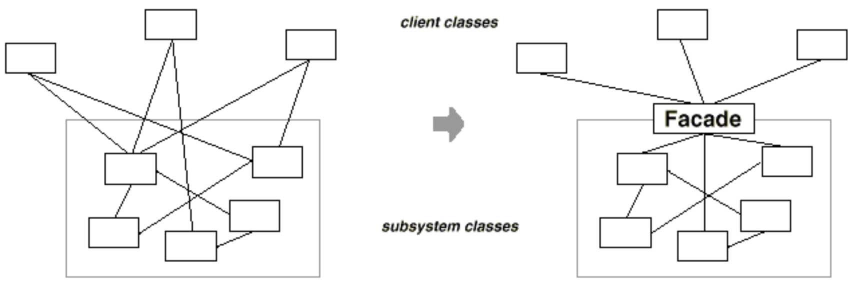
Façade

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**.

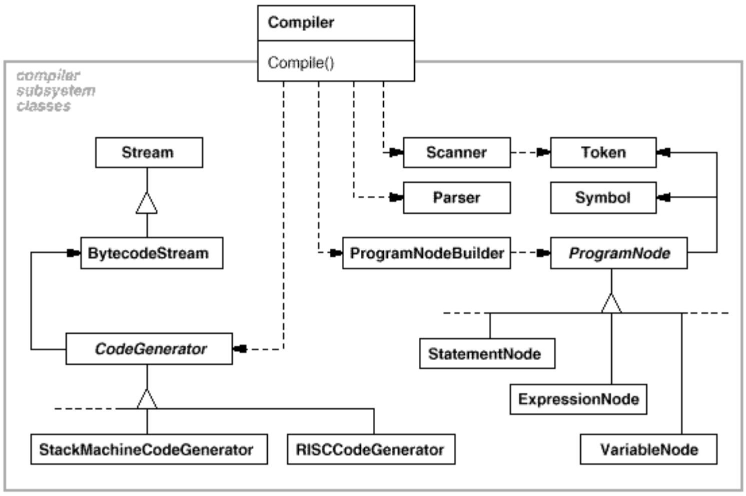
Façade

Motivation



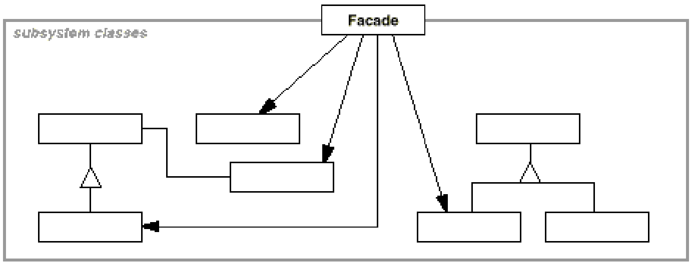
Façade

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.**

More topics on Software Design

……..

More metrics on software design quality

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

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]**

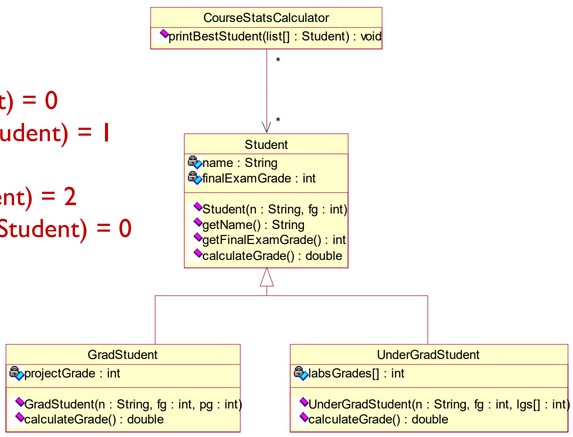
DIT(A) = depth of A in the tree

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



More on Architectural Styles …

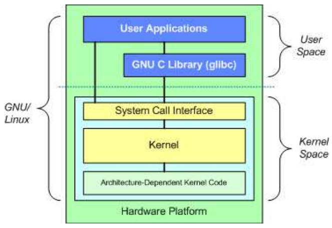
What do we mean by architectural style?

What do we mean by architectural style?

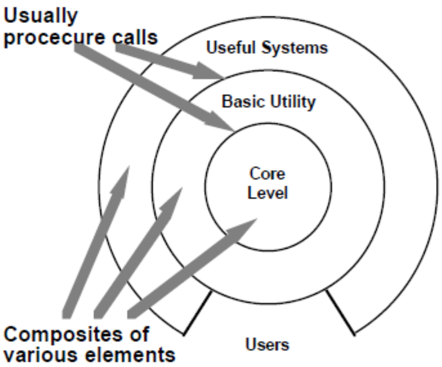
An architectural style determines the vocabulary of components (elements) and connectors (relations) that can be used in instances (architectures) of that style), together with a set of constraints on how they can be combined.

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

Architectural styles

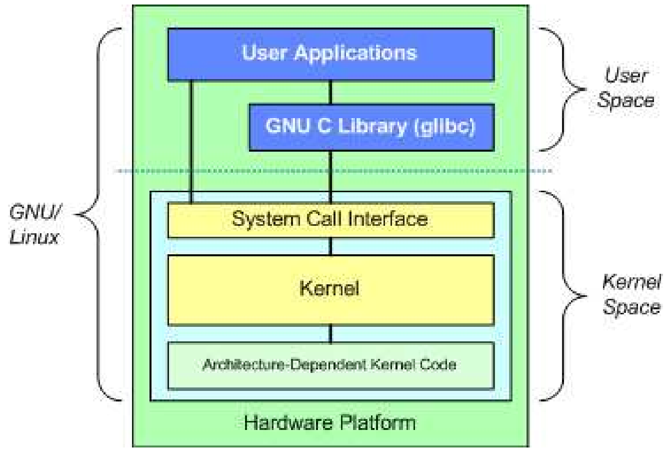


Layered architectures

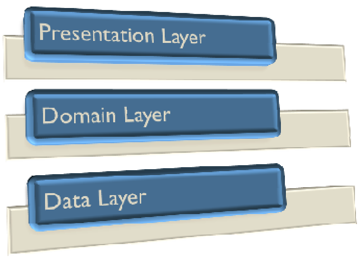


D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I , 1993

Architectural styles



Enterprise Application Architecture



Enterprise Application Architecture

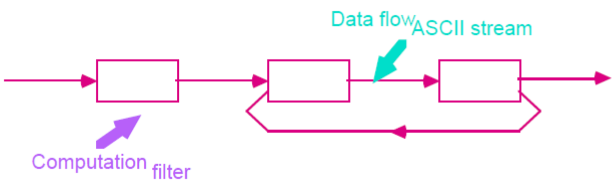
Layer	Responsibilities
Presentation	Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
Domain	Logic that is the real point of the system
Data Source	Communication with databases,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



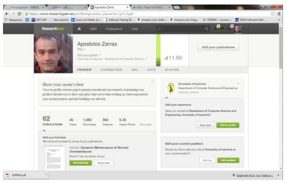
D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, 1993

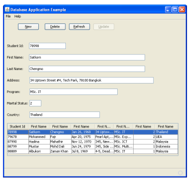
Architectural styles

```
$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 $
```

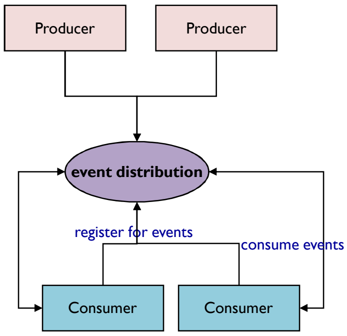
```
$ls -l | grep "Aug" | sort +4n | more

-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



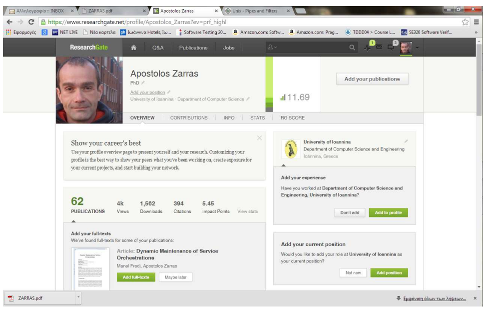


Implicit invocation / event based



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

Architectural styles



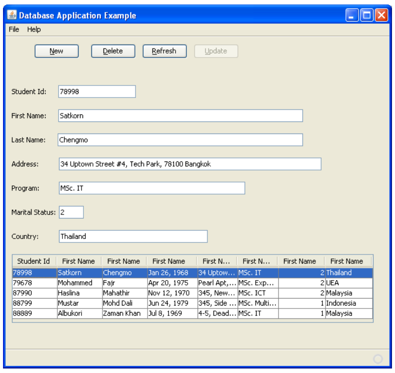
Social networking sites
like ResearchGate Linkedin for professionals and researchers to share papers, ask and answer questions, and find collaborators

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

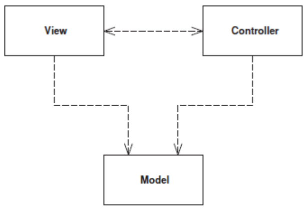
Model – View – Controller (MVC)

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

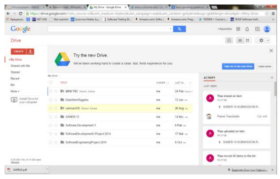
The **view** represents the display of the model in the UI.

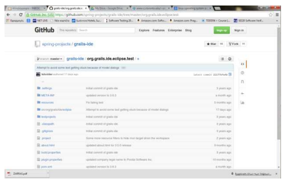
The **controller** takes user input, manipulates the model, and causes the view to update appropriately.

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

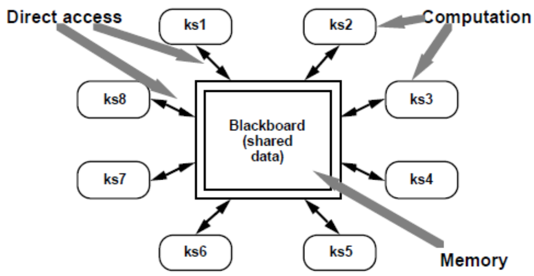


Architectural styles



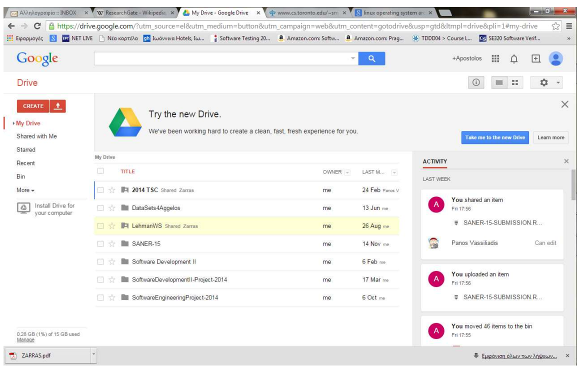


Blackboard/repository



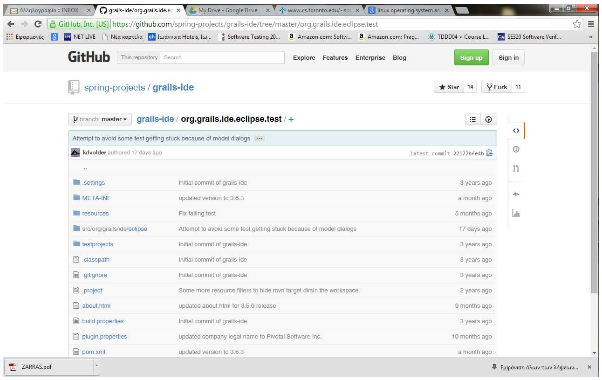
D. Garlan and M. Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, 1993

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

More GoF patterns …

Factory Method (& Parameterized Factory Variant)

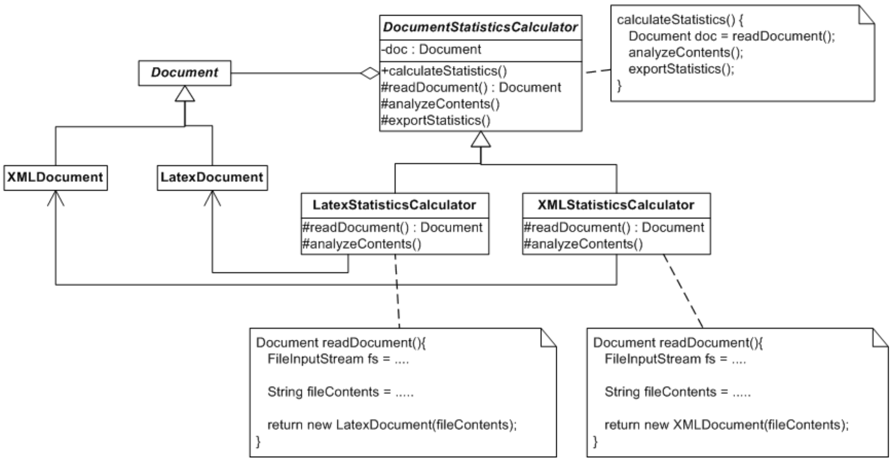
Factory Method

Intent

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

Factory Method

Motivation



Factory Method

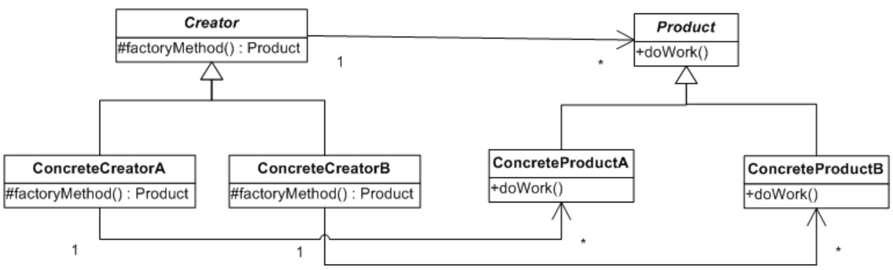
Structure

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.

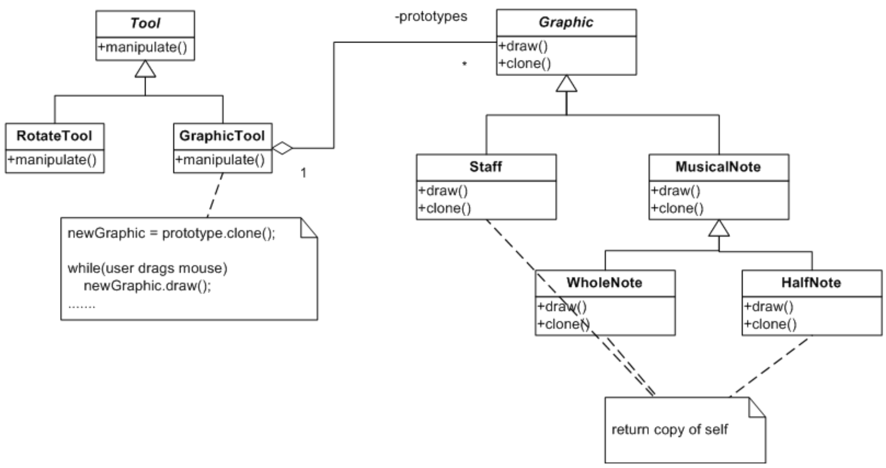
Prototype

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…



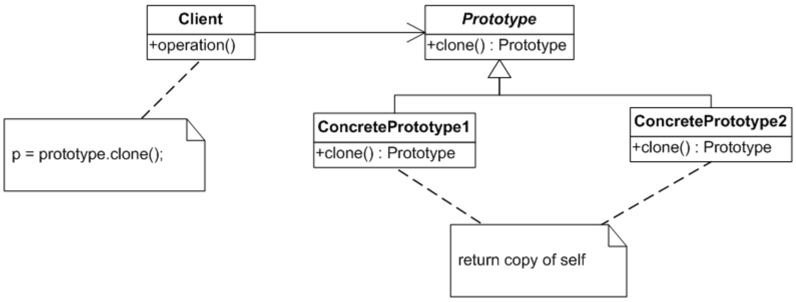
Prototype

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.

z8

Slide 148

z8 ok this means that there may be subclasses which put their instances in the same superclass static member variable

zarras, 20/04/2018

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.

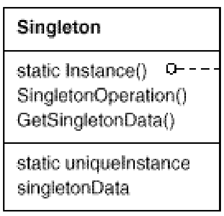
An accounting system will be dedicated to serving one company.

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**.

Decorator

Decorator

Intent

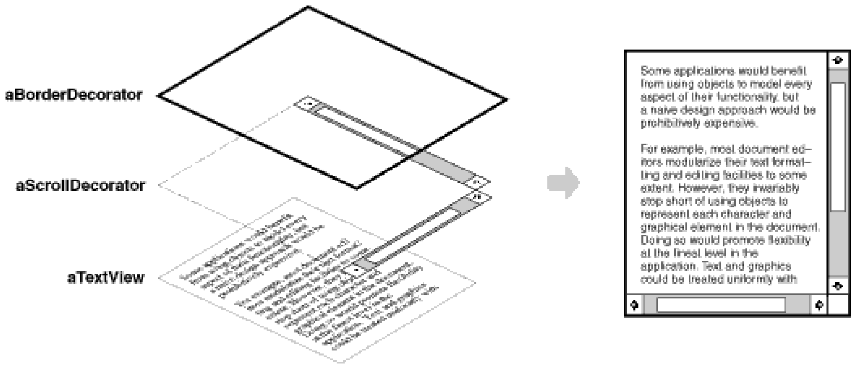
Decorators provide a flexible **alternative to subclassing** for extending functionality.

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

Decorator

Motivation

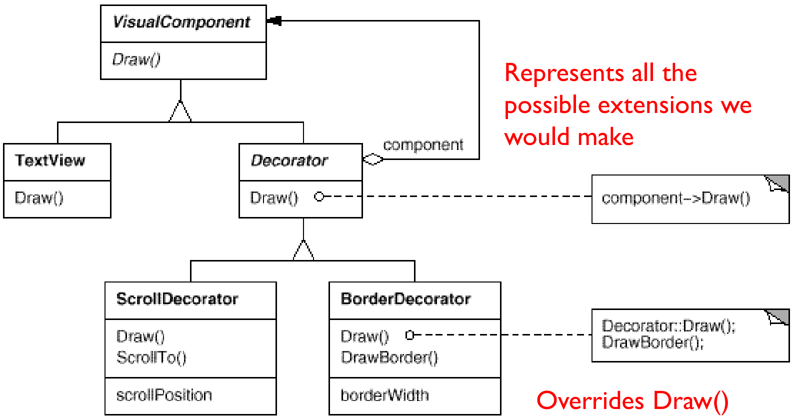
Inheritance is static: from the moment you create an object you cannot add/remove Functionalities/features



With inheritance cannot add functionalities dynamically – after the object creation

Decorator

Motivation



Represents all the possible extensions we would make

Overrides Draw() to realize the extended functionality

Decorator

Intent

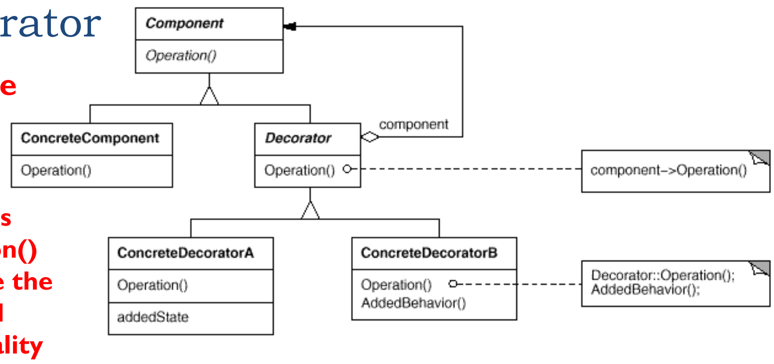
Decorators provide a flexible **alternative to subclassing** for extending functionality.

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**.

Command

Command

Intent

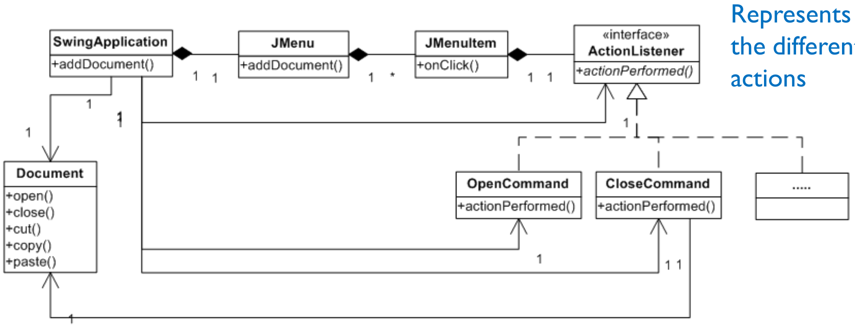
Encapsulate an **action** as an **object**, thereby letting you **parameterize clients** with **different actions**, **queue** or **log** actions, and support **undoable/redoable actions**.

Command

Motivation

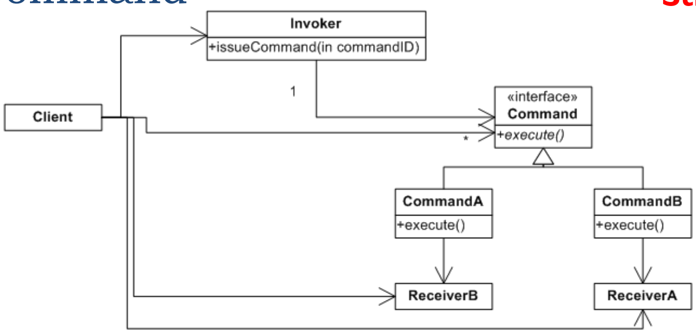
Every menu item should do a diff. action

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

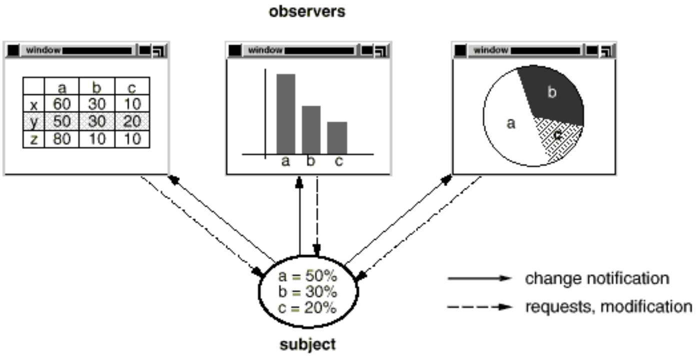
Observer

Intent

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

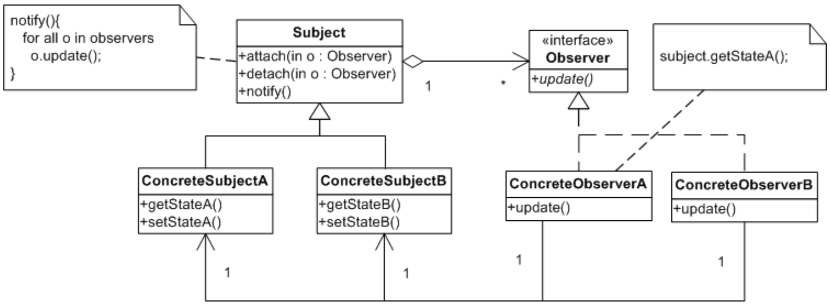
Observer

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**.

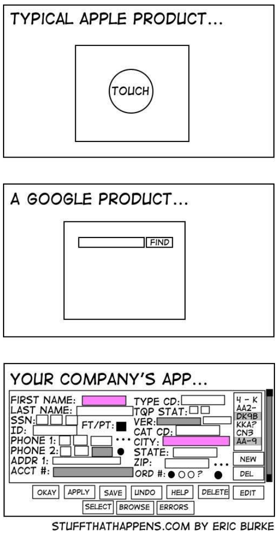
Liabilities

- **Unexpected/unwanted notifications.**

User interface design

What are the fundamental UI design principles?

UI design principles



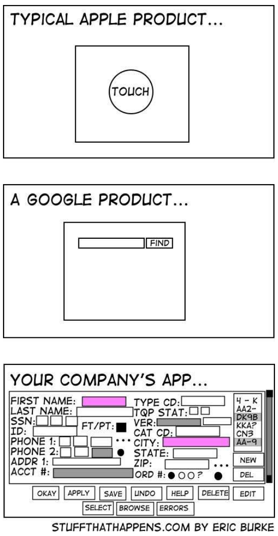
Learnability.
The software should be easy to learn so that the user can rapidly start working with the software.

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



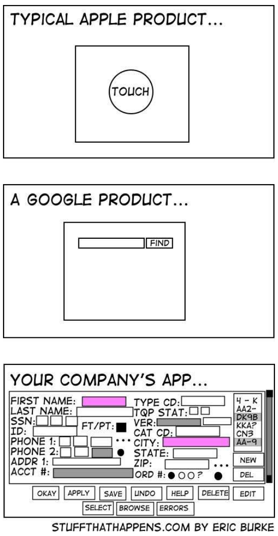
Recoverability.
The interface should provide mechanisms allowing users to recover from errors.

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

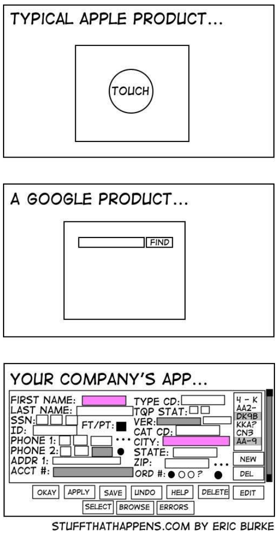


Question-answer.
The interaction is essentially restricted to a single question-answer exchange between the user and the software.

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



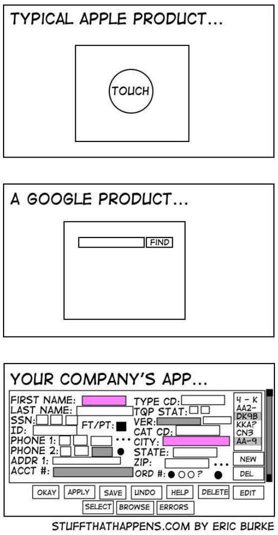
Form fill-in.
The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.

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.