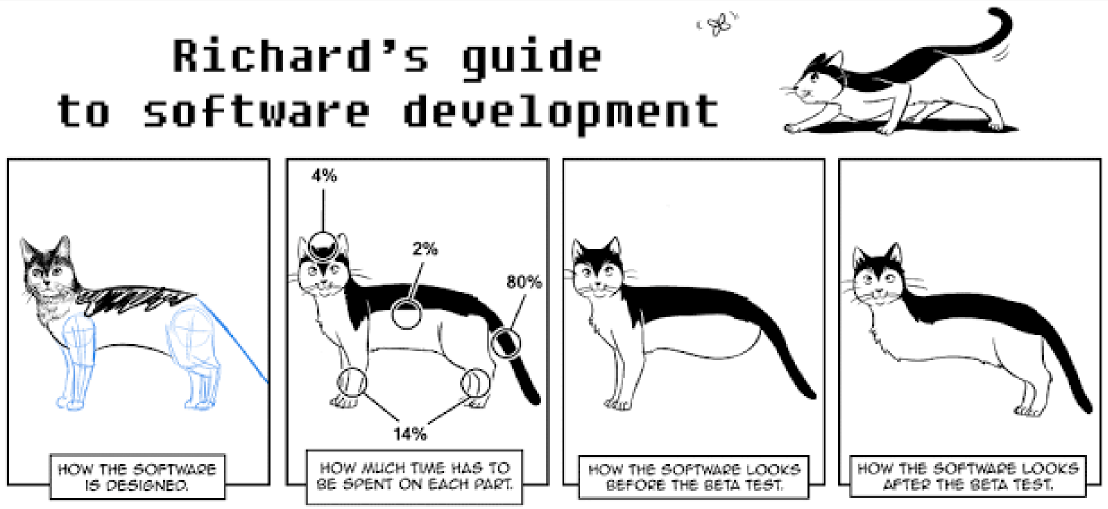
CSEN603 – Software Engineering

Lecture 3: Software Design I

Mervat Abuelkheir
Ammar Yasser
Mohamed Agamia
Nada Hisham

- Definition and Concepts
- Design Models
- Design Patterns
- UI


Richard's guide to software development
HOW THE SOFTWARE IS DESIGNED.
4%
2%
80%
14%
HOW MUCH TIME HAS TO BE SPENT ON EACH PART.
HOW THE SOFTWARE LOOKS BEFORE THE BETA TEST.
HOW THE SOFTWARE LOOKS AFTER THE BETA TEST.

Software Engineering

Architecture

Requirements Engineering

Design and Design Patterns

Implementation

Verification and Validation

Quality and Maintenance

Scale and Evolution

Economics

Process, Models, Methods

Requirements and Design

- In principle, **requirements should state what the system should do** and the **design should describe how it does this**
- In practice, **requirements and design are inseparable**
  - Your system may inter-operate with other systems that generate design requirements
  - Use of a specific architecture to satisfy nonfunctional requirements may be a domain requirement
  - Design options may be the consequence of a regulatory requirement

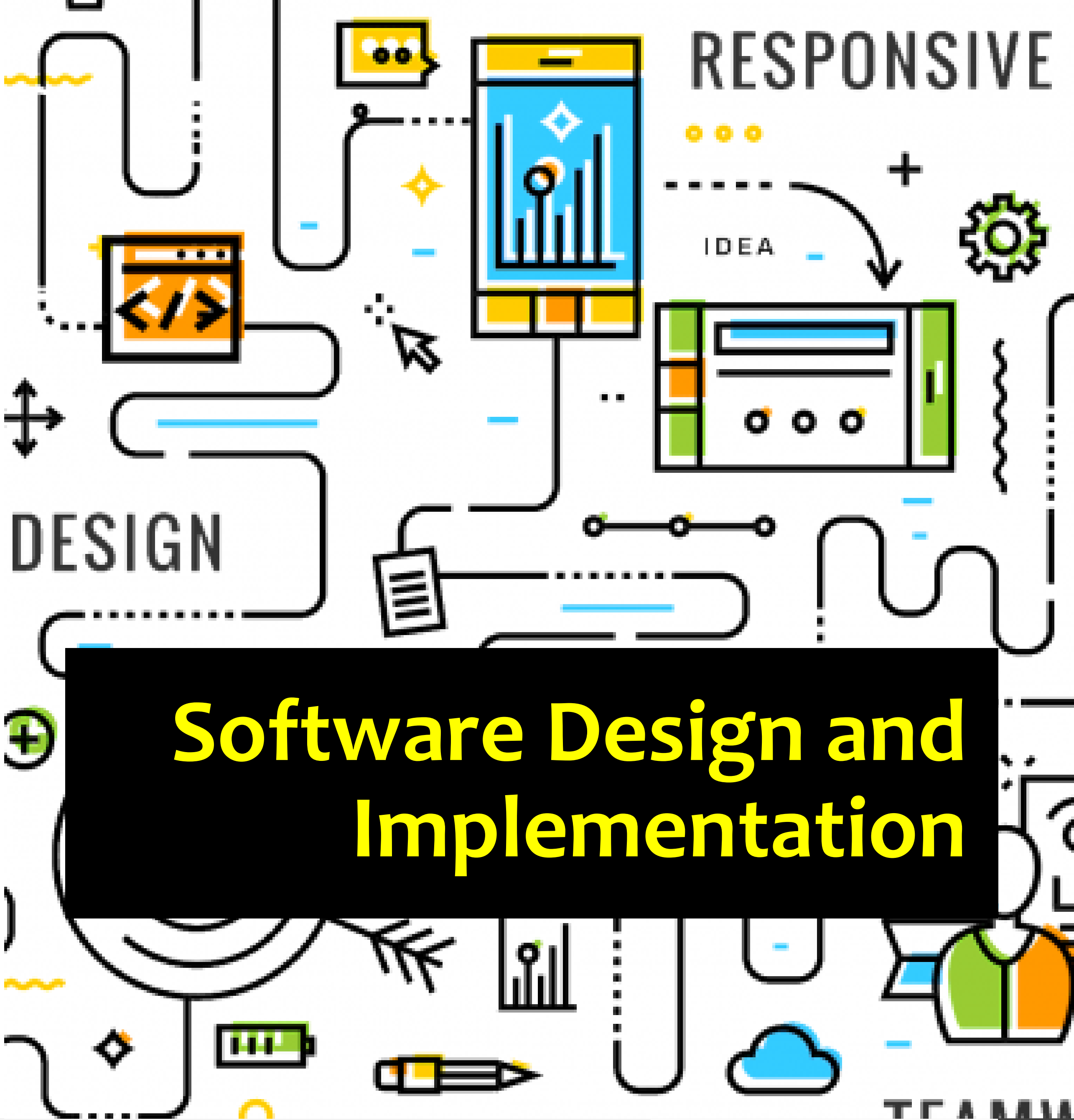
Requirements and Design

- You do not always build a **new system** from scratch, so analyze requirements accordingly
  - Clients often have an **old system** that is so familiar that they do not realize it has functions not needed in a new system
  - A **replacement system** is when a system is built to **replace** an existing system
  - A **legacy system** is an existing system that is not being replaced, but is being **extended** or must interface to a new system

- In requirements analysis, it is important to distinguish:
  - features of the old system that are needed in the new system
  - features of the old system that are not needed in the new system
  - proposed new features
- **In a nutshell, software developers:**
  - **Develop** (new software)
  - **Understand** (existing software)
  - **Maintain** (fix bugs)
  - **Upgrade** (add new features)

Software Design and Implementation

- Software **design** and **implementation** is the stage in the software engineering process at which an executable software system is developed from requirements
- Software design and implementation activities are interleaved
  - Software **design is a creative activity** in which you **identify software components and their relationships**, based on customer’s requirements
  - **Implementation is** the process of **realizing the design as a program**
- Focus on concepts of **modularization**, **cohesion**, **coupling**, **managing complexity**, **abstraction**



Design Concepts

Modularity

- Dividing software into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements

Cohesion

- The degree to which the elements of a module belong together
- A cohesive module performs a single task requiring little interaction with other modules

Coupling

- The degree of interdependence between modules

Information Hiding

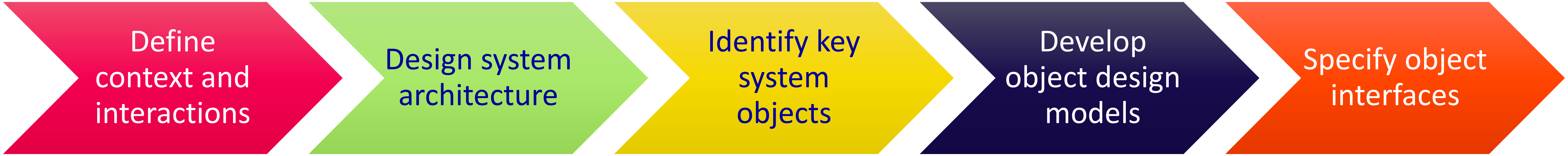
- Not exposing internal information of a module unless necessary
- e.g. private fields, getter & setter methods

Abstraction

- Managing the complexity of software
- Anticipating detail variations and future changes

An Object-Oriented Design Process

- Structured OO design processes involve developing different system design models
- Design models require a lot of effort for development and maintenance
- for small systems → **not cost-effective**
- for large systems → **important**, as large systems are typically developed by different groups
- **Common activities in OO design processes are:**



Define context and interactions

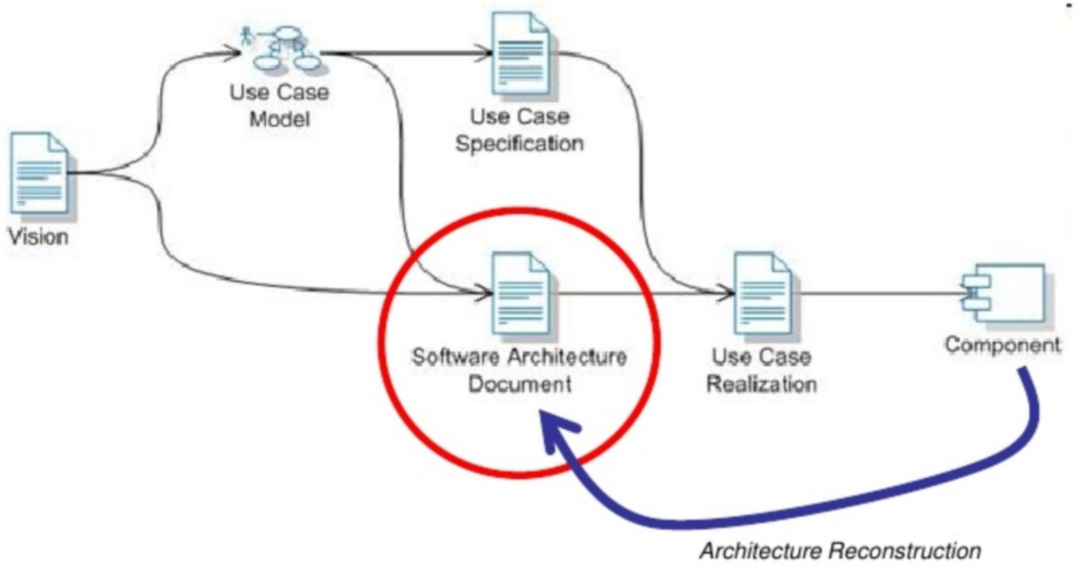
Design system architecture

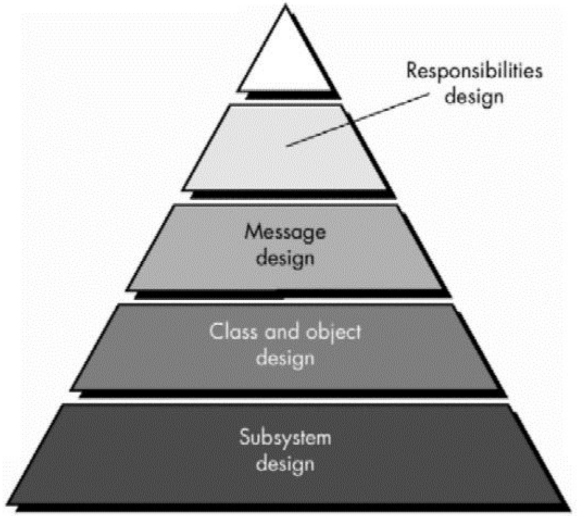
Identify key system objects

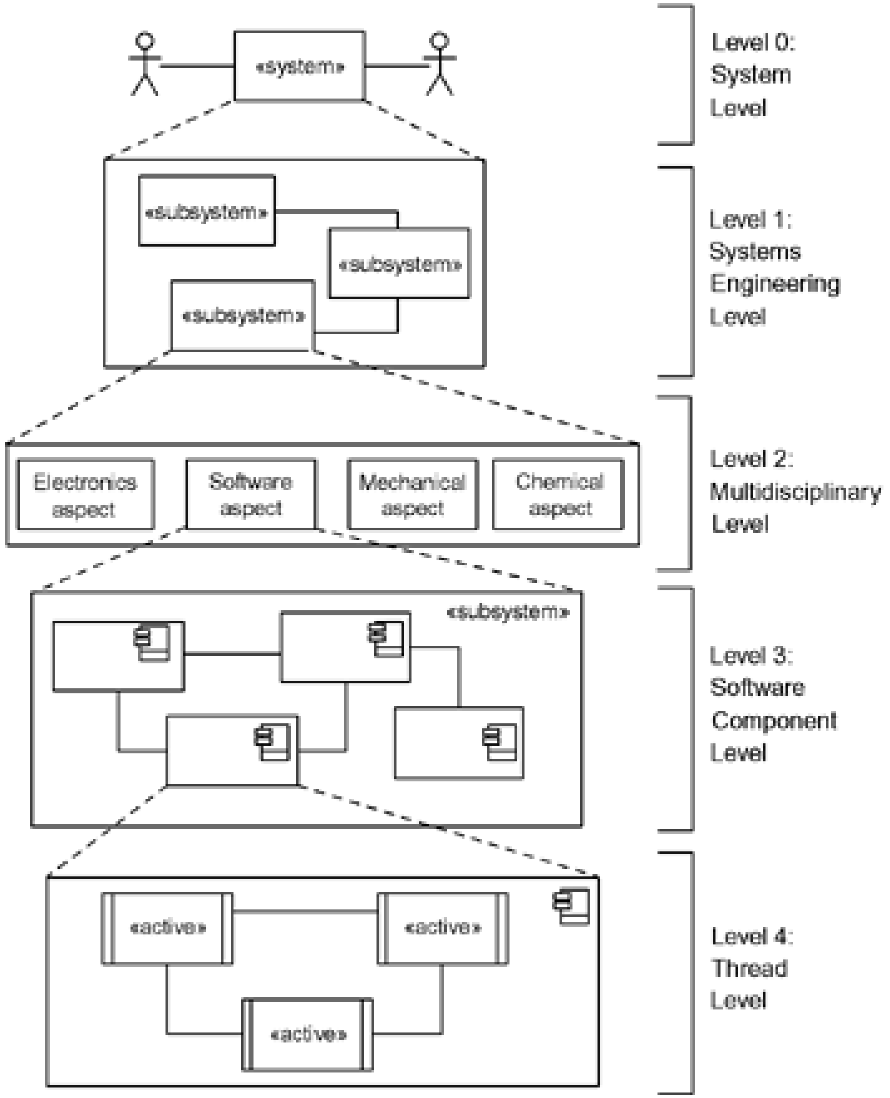
Develop object design models

Specify object interfaces

Big Design Picture







System Modeling

Context Models

- Model the system boundaries and environment
- Context diagrams, Activity diagrams

Interaction Models

- Model the interactions between system and environment or system components
- Use-case models, Sequence diagrams

Structural Models

- Model the organization of a system or the structure of the data
- Class diagrams

Behavioral Models

- Model the behavior of the system and how it responds to events
- Activity diagrams, Sequence diagrams, State diagrams

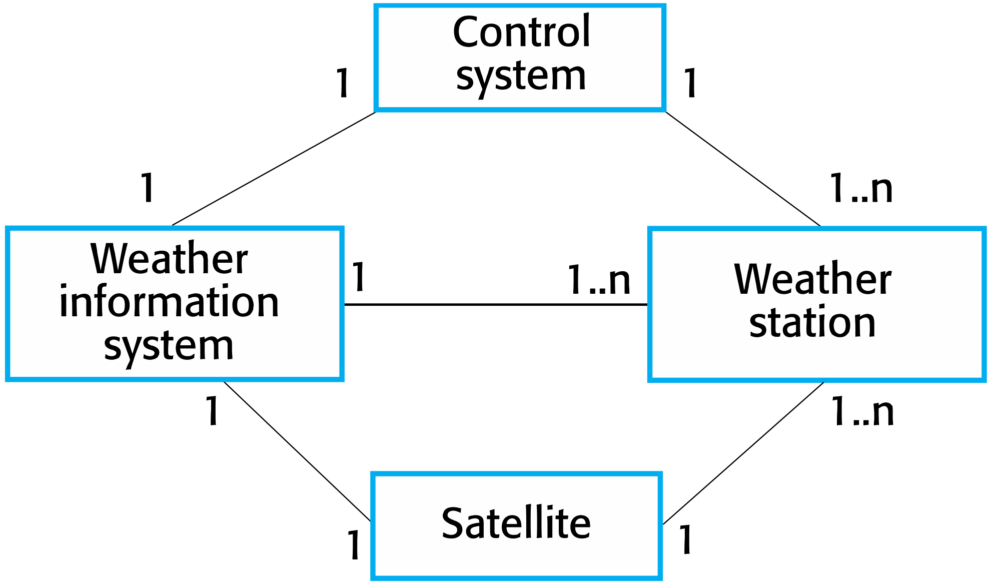
Which are static and which are dynamic?

1- System Context and Interactions

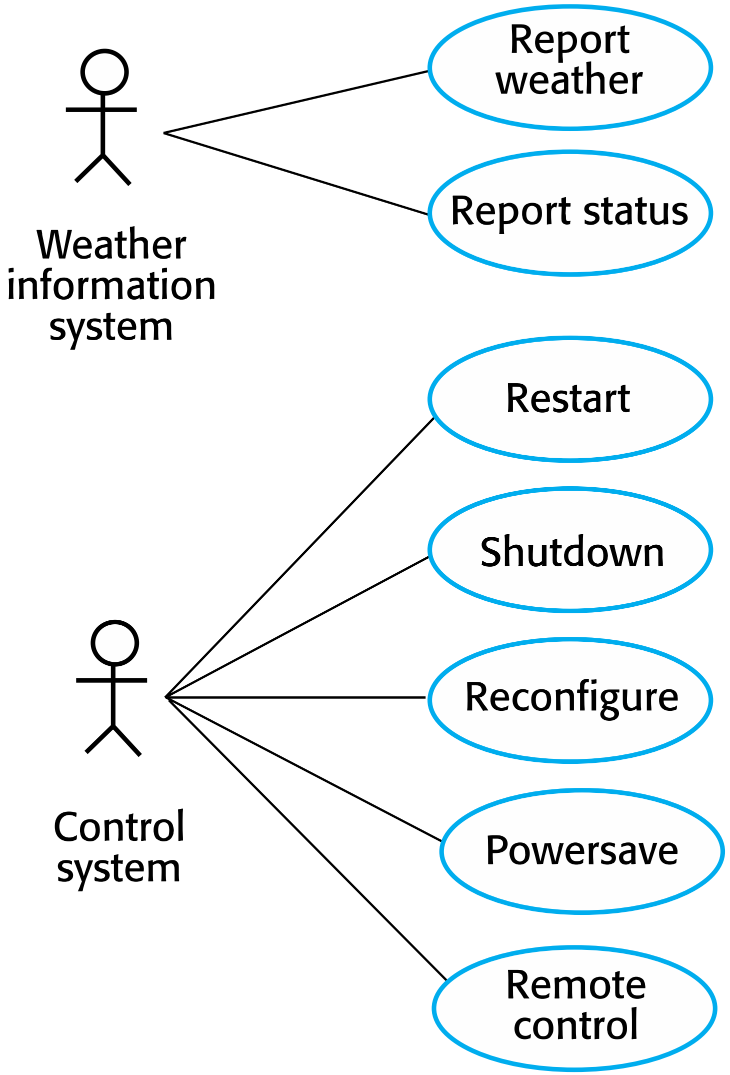
- Understanding the **context** allows you to establish system boundaries:
  - What **features** are implemented in the system
  - What features are in other **associated systems**
  - Context diagrams and process diagrams
- Understanding the **relationships** between software being designed and its external environment helps decide:
  - How to provide the required system **functionality**
  - How to structure system to **communicate** with environment
  - Use case diagrams and sequence diagrams

- A system **context model** is a **structural** model that demonstrates the other systems in the environment
- An **interaction model** is a **dynamic** model that shows how the system interacts with its environment as it is used

Let's design a wilderness weather station software



1- System Context and Interactions

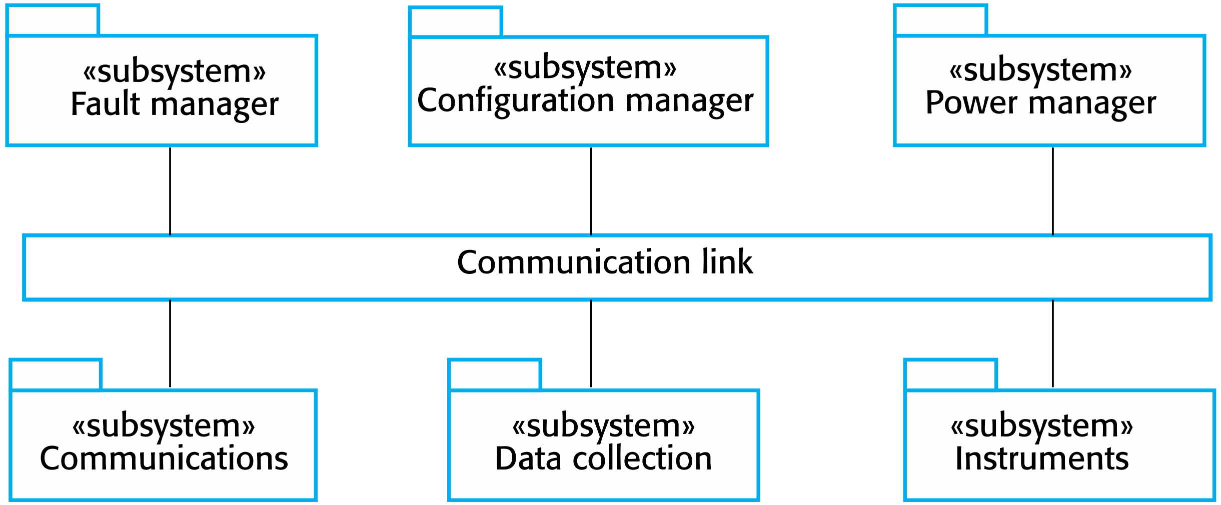


System	Weather Station
Use Case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future

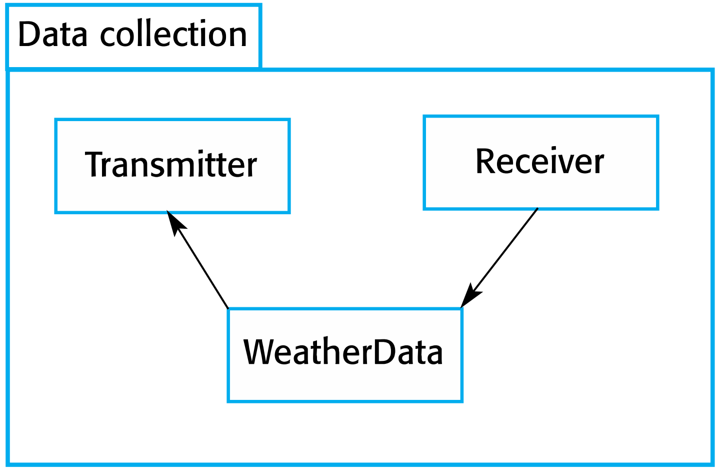
2- Architectural Design

- **Interactions** between the system and its environment **are used to design the system architecture**
- Identify the major components that make up the system and their interactions
- Organize the components using an architectural pattern (recall lecture 1)
    - Pipes and filters?
    - Client-server?
    - Repository?

The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure



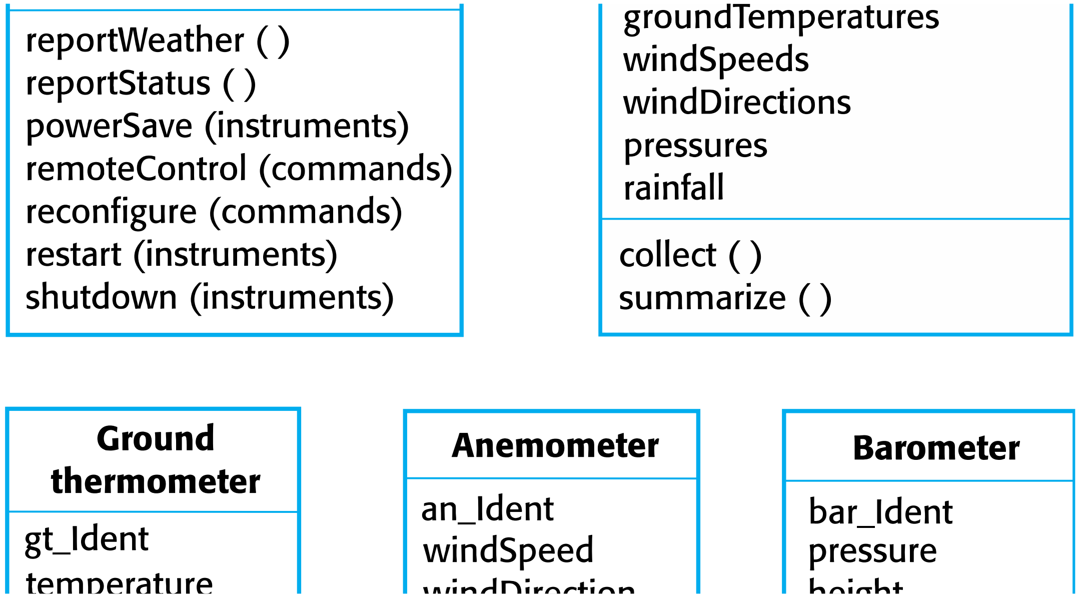
High-level architecture of the weather station



Architecture of data collection subsystem

3- Object Class Identification

- **Identifying object classes is difficult**
- No “magic formula” – this task requires skill, experience, and domain knowledge
- **An iterative process** – you are unlikely to get it right first time
- You can:
  - **Identify objects based on tangible things** in the application domain
  - **Identify objects based on** what participates in what **behavior**
  - **Identify objects**, attributes, and methods **using scenario-based analysis**



reportWeather ( )
reportStatus ( )
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ( )
summarize ( )

Ground thermometer
gt_Ident
temperature

Anemometer
an_Ident
windSpeed
windDirection

Barometer
bar_Ident
pressure
height

For the weather station system, use system hardware and data for object identification:

- **Ground thermometer, Anemometer, Barometer**
  ‘hardware’ objects related to system instruments
- **Weather station**
  Basic interface of weather station to environment
  Reflects interactions in the use-case model
- **Weather data**
  Encapsulates summarized data from the instruments

4- Object Design Models

- Show the **objects and object classes and relationships between these entities**
- **Two broad types of design models:**
  - Structural models describe the static structure of the system in terms of object classes and relationships
  - Dynamic models describe the dynamic interactions between objects

Examples of Design Models

- **Subsystem models** → show **logical groupings of objects** into coherent subsystems
- **Sequence models** → show the **sequence of object interactions**
- **State machine models** → show **how objects change their state** in response to events
- Other models → use-case models, aggregation models, etc.

4- Object Design Models

Subsystem Models

- Show how the **design is organized into logically related groups of objects**
- Shown using **packages** – an encapsulation construct
- Logical models – the actual organization of objects in the system may be different

Examples of Design Models

- **Subsystem models** → show **logical groupings of objects** into coherent subsystems
- **Sequence models** → show the **sequence of object interactions**
- **State machine models** → show **how objects change their state** in response to events
- Other models → use-case models, aggregation models, etc.

4- Object Design Models

Sequence Models

- Show the **sequence of object interactions**
  - Objects are arranged horizontally across the top
  - Time is represented vertically so models are read top to bottom
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system

Examples of Design Models

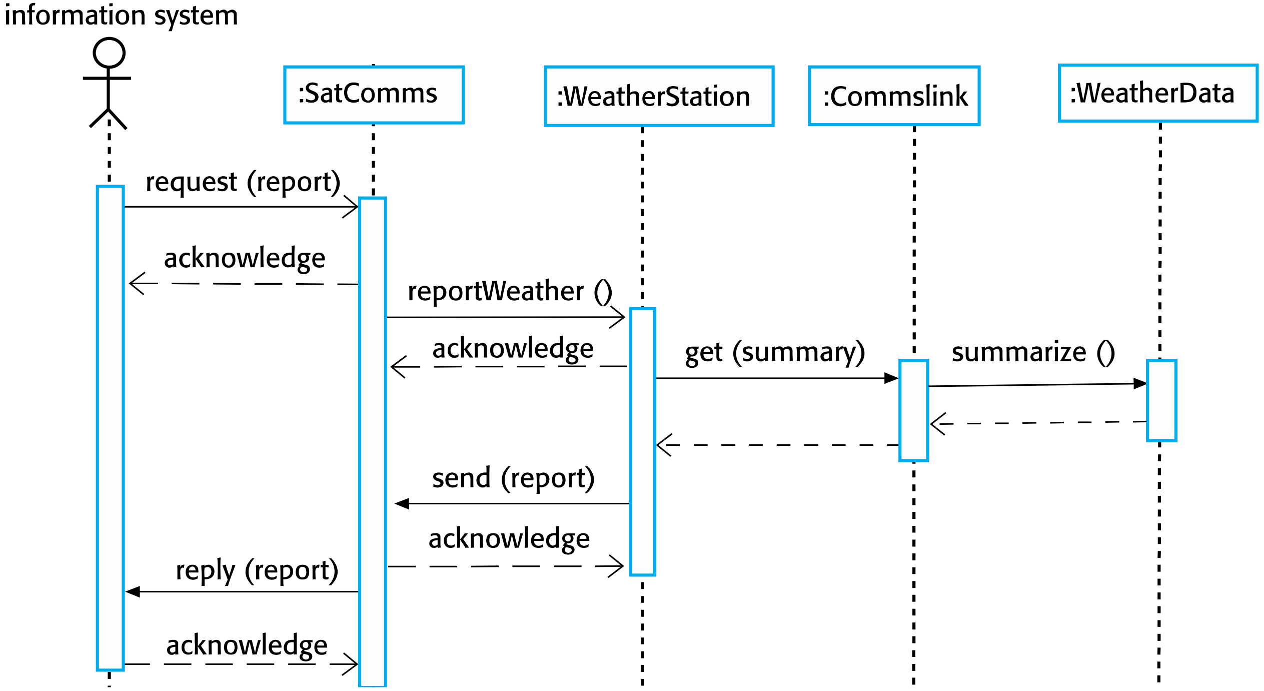
- **Subsystem models** → show **logical groupings of objects** into coherent subsystems
- **Sequence models** → show the **sequence of object interactions**
- **State machine models** → show **how objects change their state** in response to events
- Other models → use-case models, aggregation models, etc.

4- Object Design Models

Sequence Models

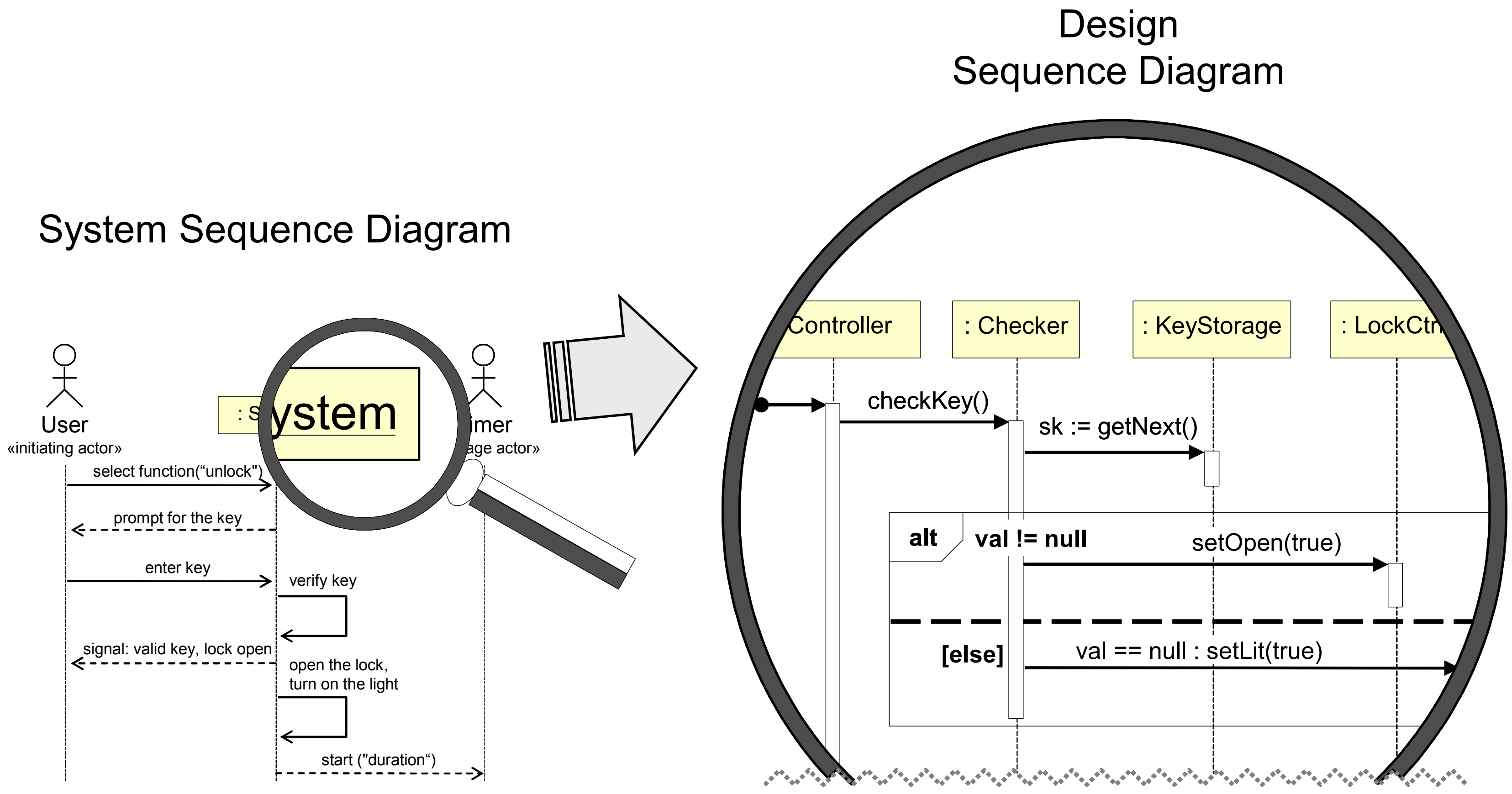
- Show the **sequence of object interactions**
  - Objects are arranged horizontally across the top
  - Time is represented vertically so models are read top to bottom
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system

Sequence diagram describing data collection subsystem in weather station system



System vs. Object Sequence Models

- **System Sequence Diagrams** represent interactions of external actors
- **Object Sequence Diagrams** represent interactions of objects inside the system

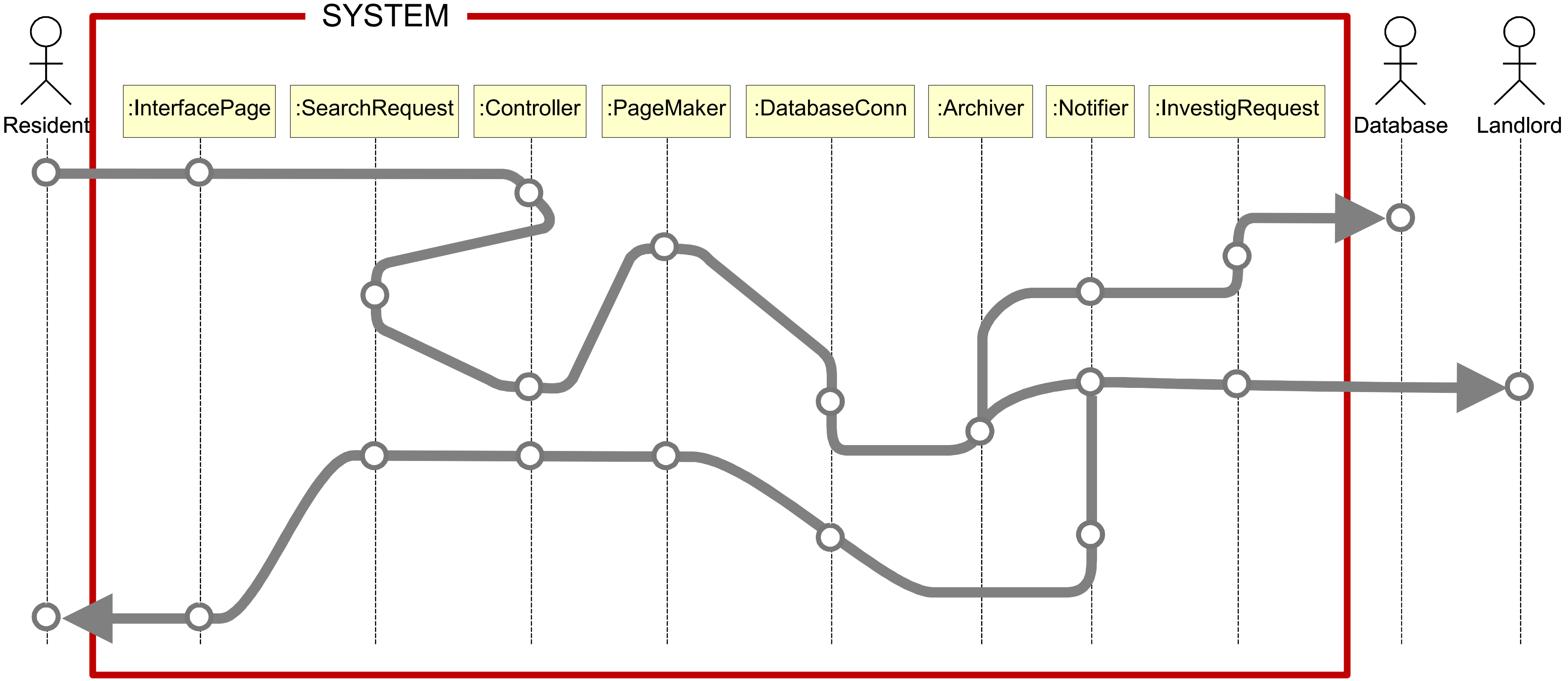


Design Sequence Diagram

System Sequence Diagram

System vs. Object Sequence Models

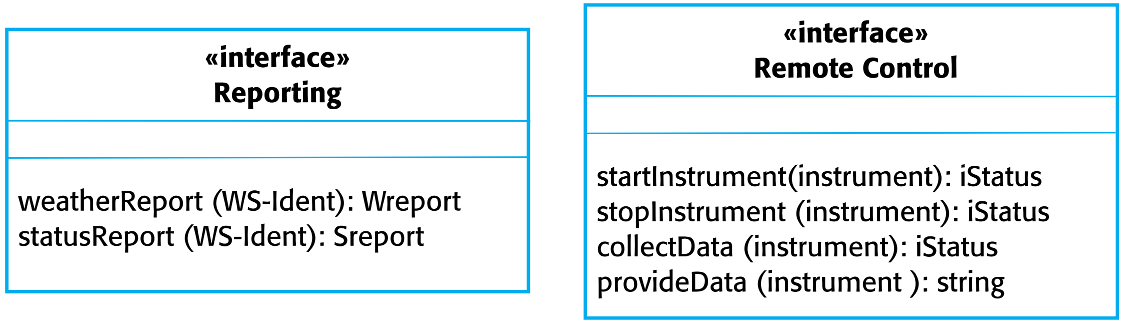
- We start System Sequence Diagrams (show only actors and the system as a “black box”)
- Then design the internal behavior using conceptual objects and modify or introduce new objects, as needed to make the system function work



5- Interface Specification

- Object interfaces have to be specified so that objects and components can be designed in parallel
- Objects may have several interfaces which are viewpoints on the methods provided
- Class diagrams are used for interface specification
- More on that next lecture

Example object interfaces for weather station object class



«interface»
Reporting

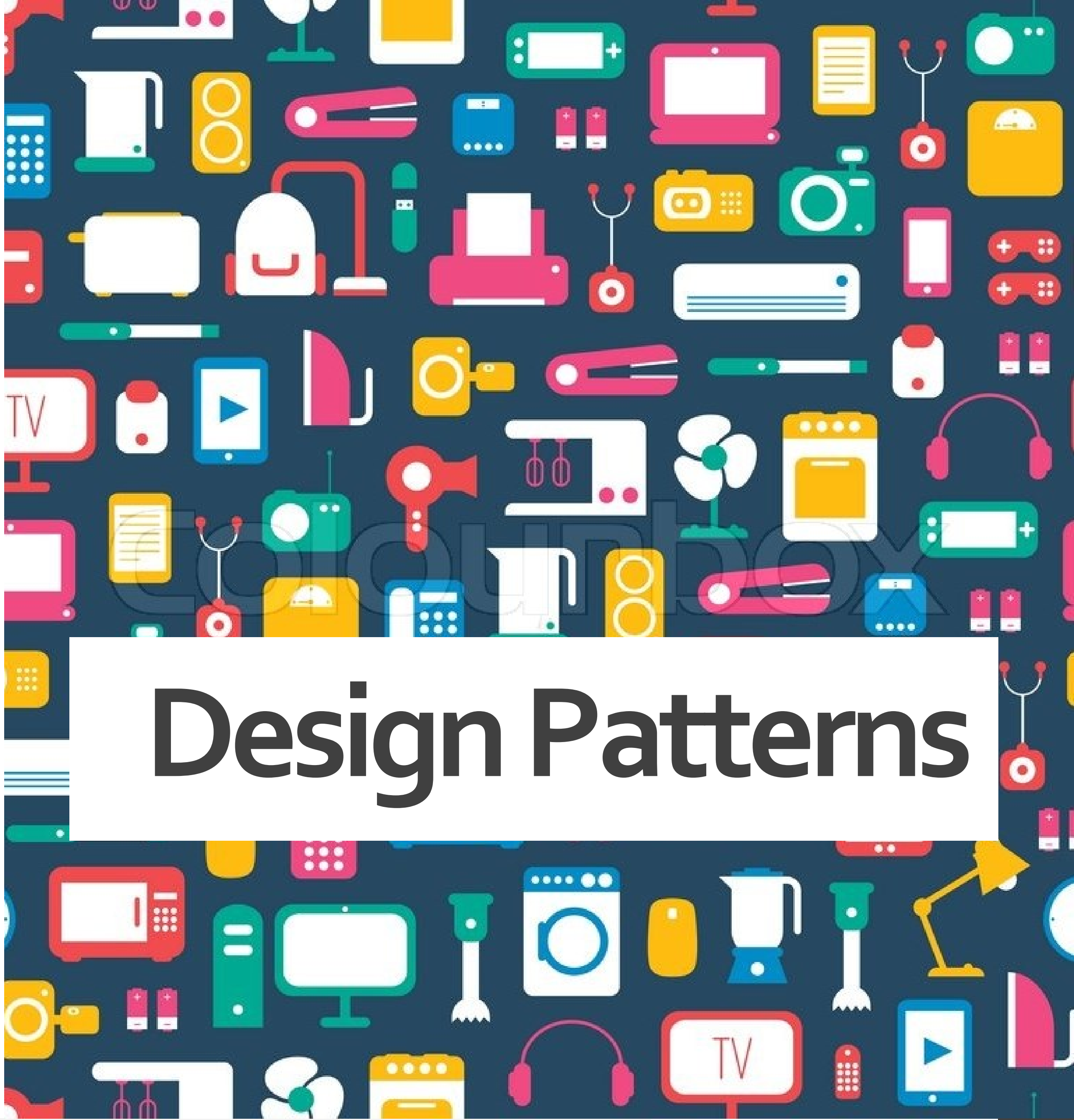
weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

«interface»
Remote Control

startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Design Patterns

- A **design pattern is a description of the problem and the <u>essence</u> of its solution**
- A way of **reusing** abstract knowledge about a problem and its solution
- Describe **best practices**, **good designs**, and **capture experience**
- Should be **sufficiently abstract** to be reused in different settings
- Pattern descriptions make use of OO characteristics such as inheritance and polymorphism



Object Responsibilities (toward other objects)

- **Knowing** something (memorization of data or object attributes)
- **Doing** something on its own (computation programmed in a “method”)
  - e.g. business rules for implementing business policies and procedures
- **Calling** methods on dependent objects (communication by sending messages)
  - e.g. calling constructor methods

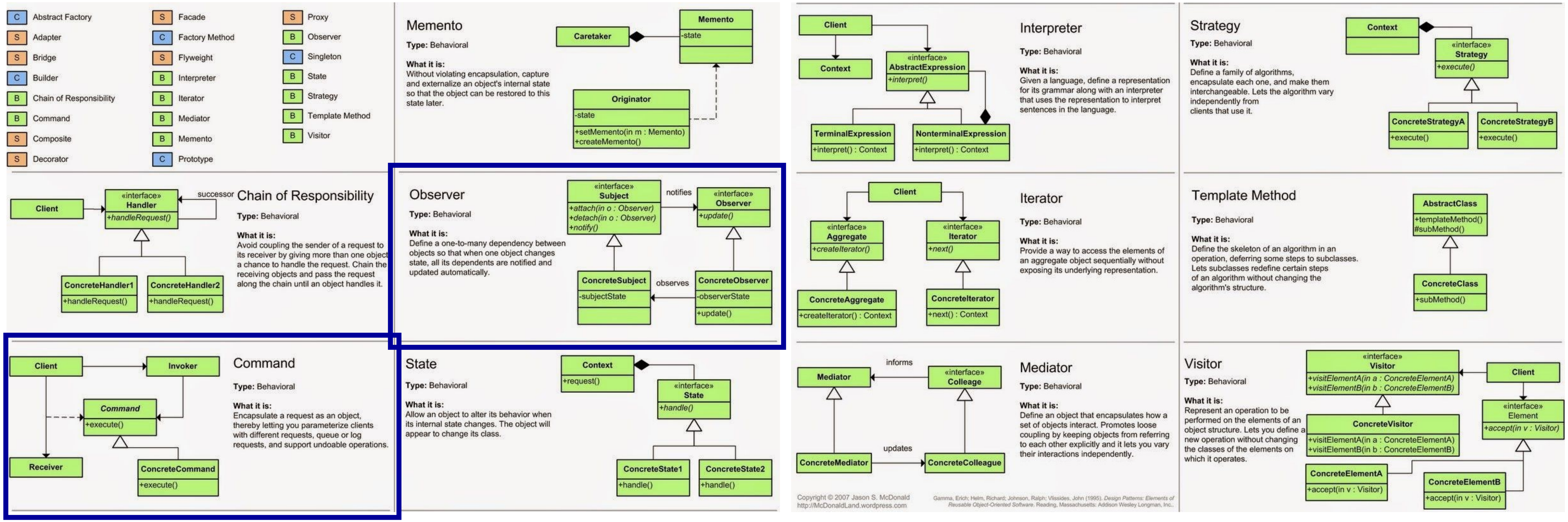
- **Design patterns provide systematic, tried-and-tested, heuristics for subdividing and refining object responsibilities**, instead of arbitrary, ad-hoc solutions

Design Patterns

Creational

Structural

Behavioral



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

Memento

Type: Behavioral

What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Chain of Responsibility

Type: Behavioral

What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Command

Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Interpreter

Type: Behavioral

What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator

Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Copyright © 2007 Jason S. McDonald
http://McDonaldLand.wordpress.com

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

Strategy

Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

Template Method

Type: Behavioral

What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor

Type: Behavioral

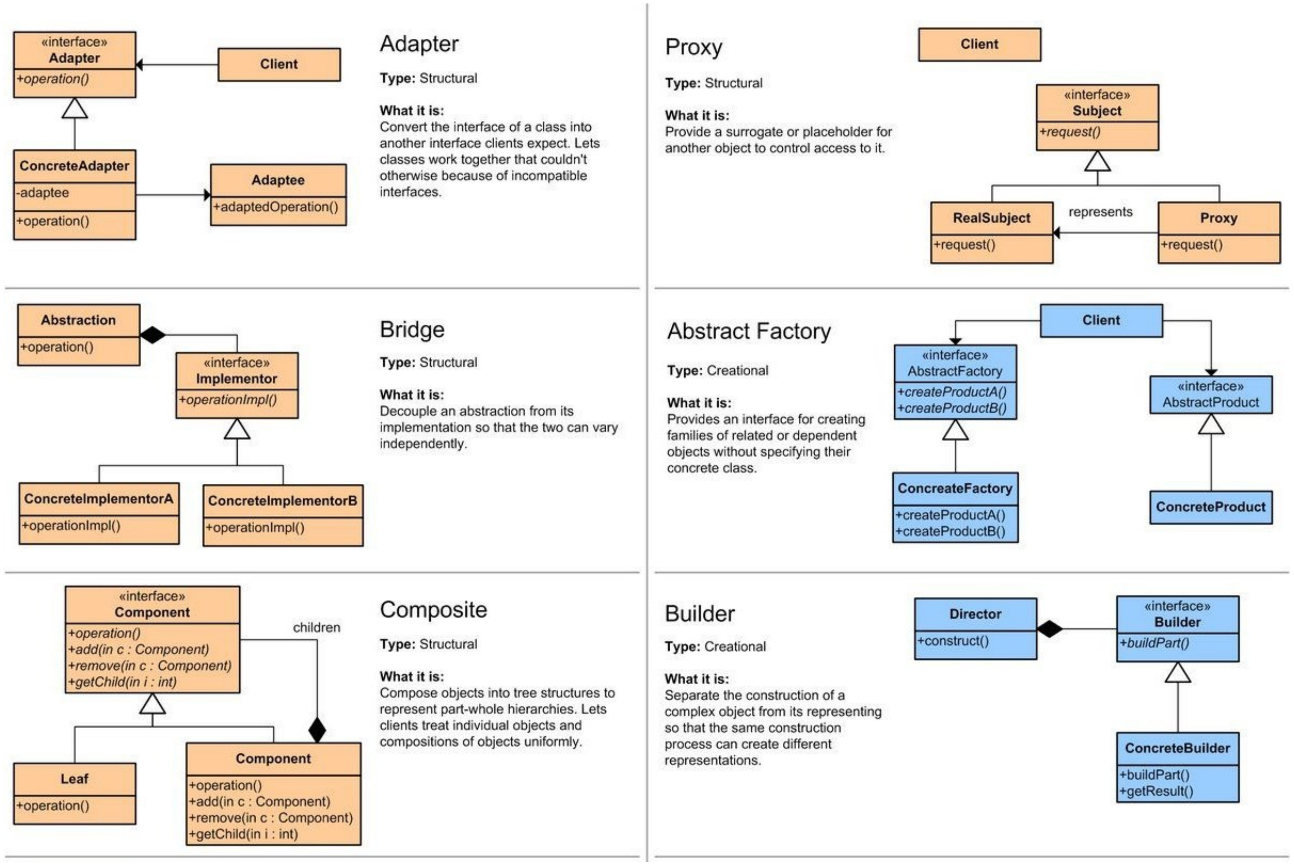
What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

https://github.com/kamranahmedse/design-patterns-for-humans

https://github.com/DovAmir/awesome-design-patterns

Design Patterns

Creational Structural Behavioral



Adapter

Type: Structural

What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Proxy

Type: Structural

What it is:
Provide a surrogate or placeholder for another object to control access to it.

Abstract Factory

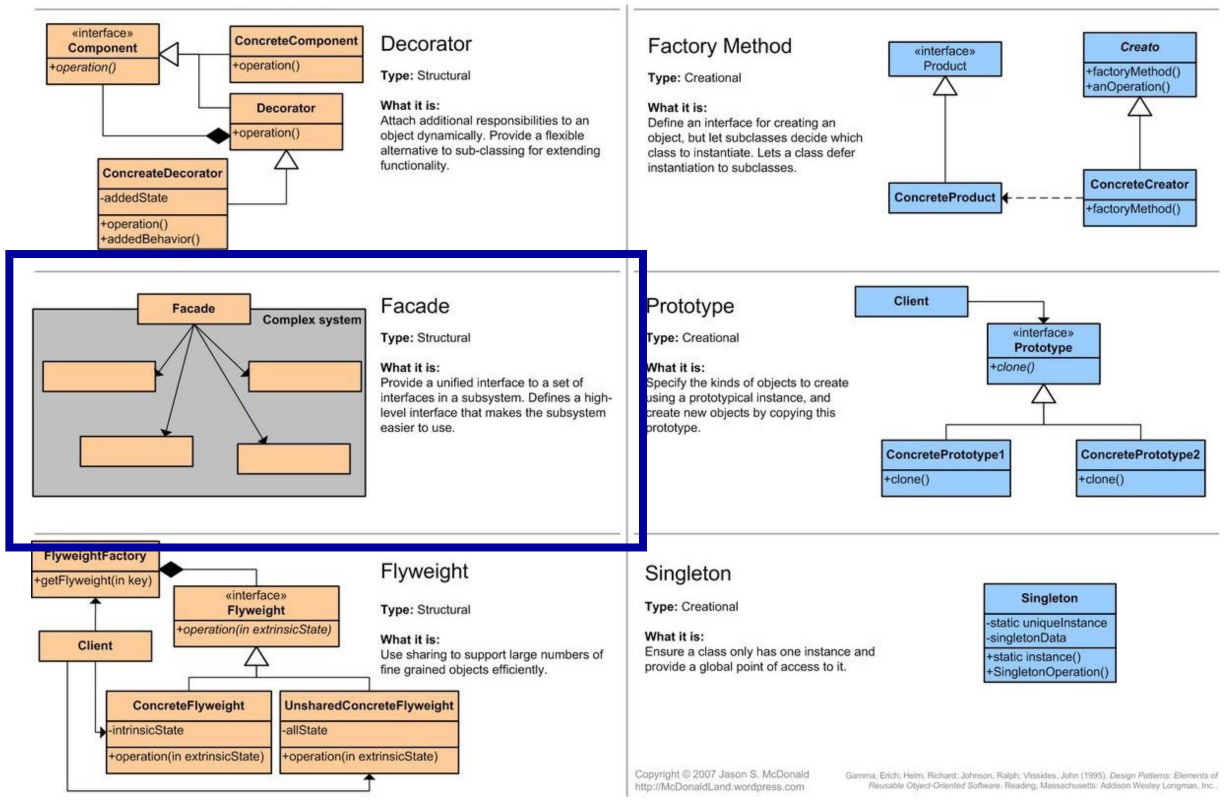
Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.

Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Decorator

Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.

Flyweight

Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Factory Method

Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

Prototype

Type: Creational

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

Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..

https://github.com/kamranahmedse/design-patterns-for-humans

https://github.com/DovAmir/awesome-design-patterns

Pattern Elements

- **Name**
  - A meaningful pattern identifier
- **Description**
- **Problem description**
- **Solution description**
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- **Consequences**
  - Results and trade-offs of applying the pattern

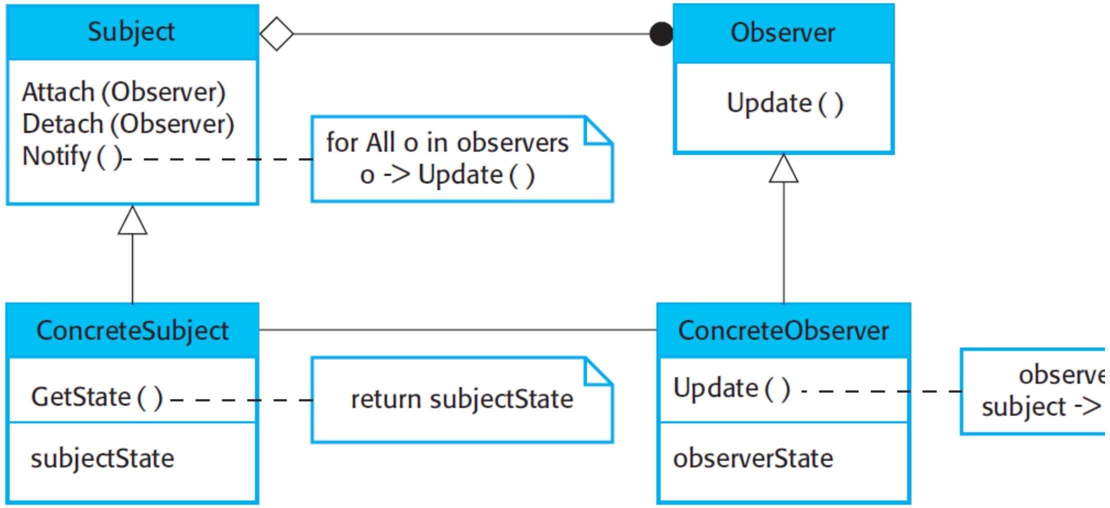
Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

The Observer Pattern

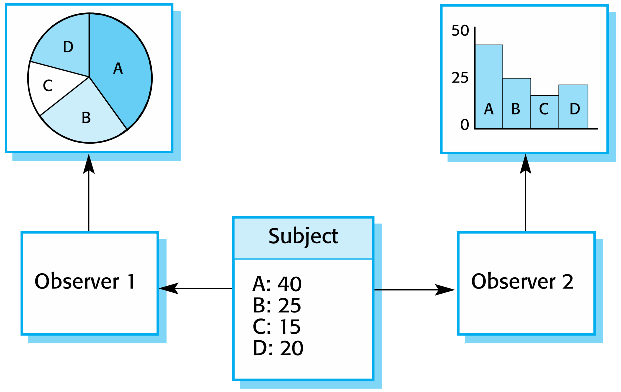
- **Name**
  - **Observer (a.k.a Publish-Subscribe)**
- **Description**
  - Separate the display of object state from the object itself
- **Problem description**
  - When you need multiple displays of a single state
- **Solution description**
  - Define `Subject` and `Observer` objects so that when a subject changes state, all registered observers are notified and updated
- **Consequences**
  - Allow addition of new cases
  - Changes to the subject may cause updates to observers

What should Observer Know? Do? What should Subject Know? Do?

Class diagram with the Observer pattern

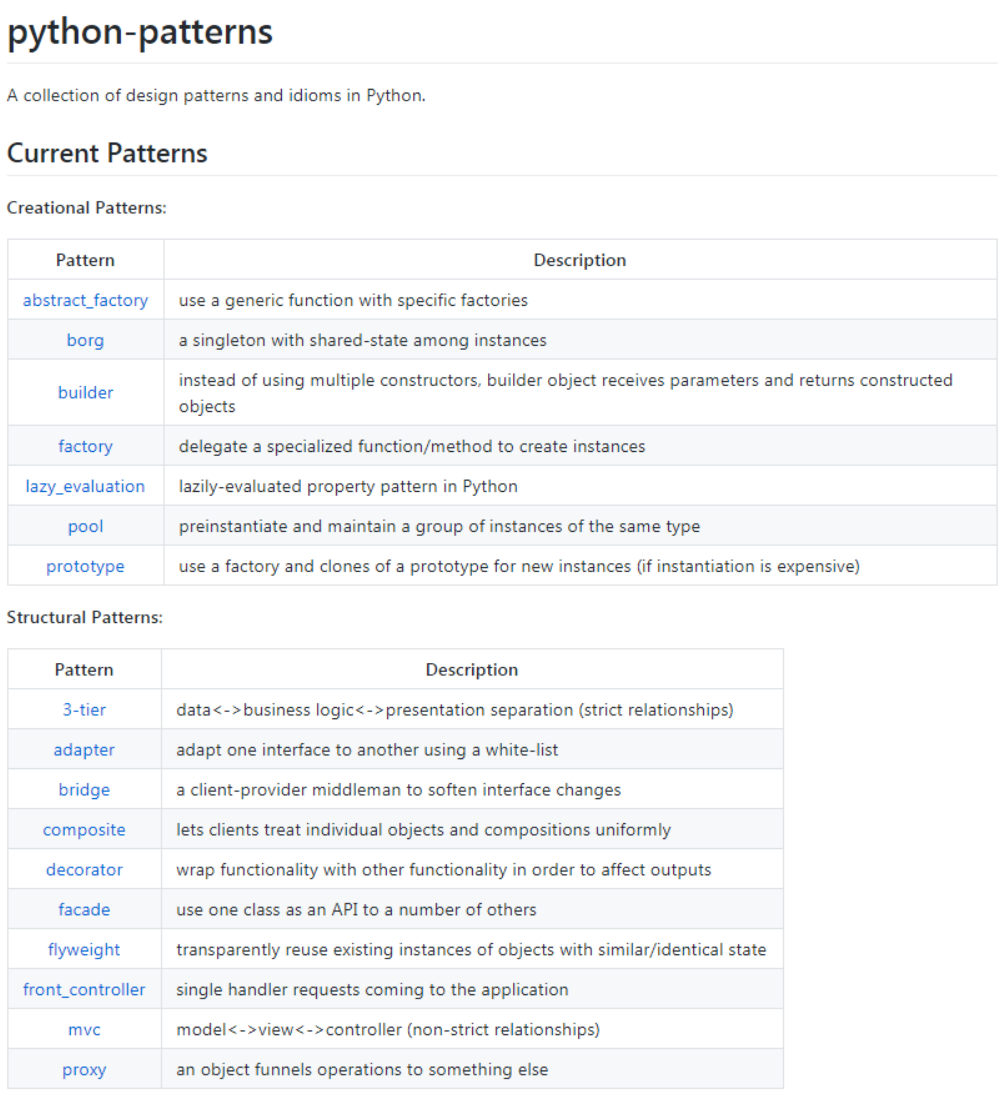


Multiple displays using the Observer pattern



Design Patterns Across Programming Languages

https://github.com/DovAmir/awesome-design-patterns



python-patterns

A collection of design patterns and idioms in Python.

Current Patterns

Creational Patterns:

Pattern	Description
abstract_factory	use a generic function with specific factories
borg	a singleton with shared-state among instances
builder	instead of using multiple constructors, builder object receives parameters and returns constructed objects
factory	delegate a specialized function/method to create instances
lazy_evaluation	lazily-evaluated property pattern in Python
pool	preinstantiate and maintain a group of instances of the same type
prototype	use a factory and clones of a prototype for new instances (if instantiation is expensive)

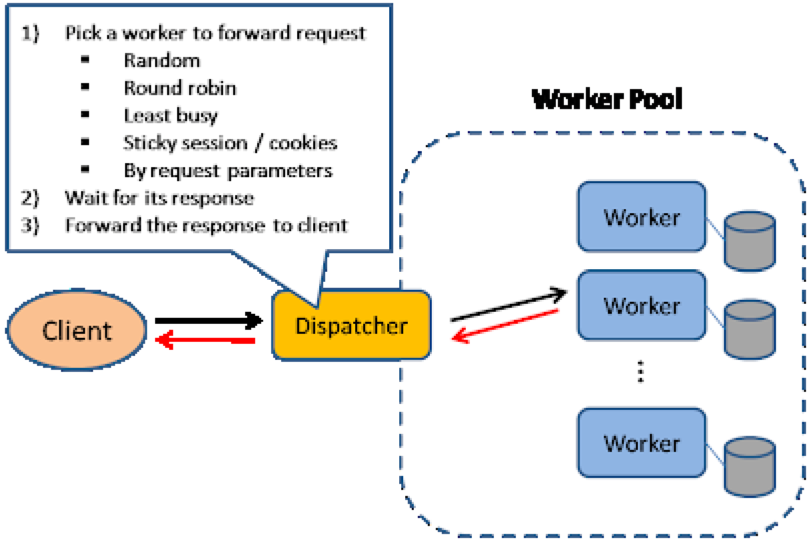
Structural Patterns:

Pattern	Description
3-tier	data<->business logic<->presentation separation (strict relationships)
adapter	adapt one interface to another using a white-list
bridge	a client-provider middleman to soften interface changes
composite	lets clients treat individual objects and compositions uniformly
decorator	wrap functionality with other functionality in order to affect outputs
facade	use one class as an API to a number of others
flyweight	transparently reuse existing instances of objects with similar/identical state
front_controller	single handler requests coming to the application
mvc	model<->view<->controller (non-strict relationships)
proxy	an object funnels operations to something else

Scalable System Design Patterns

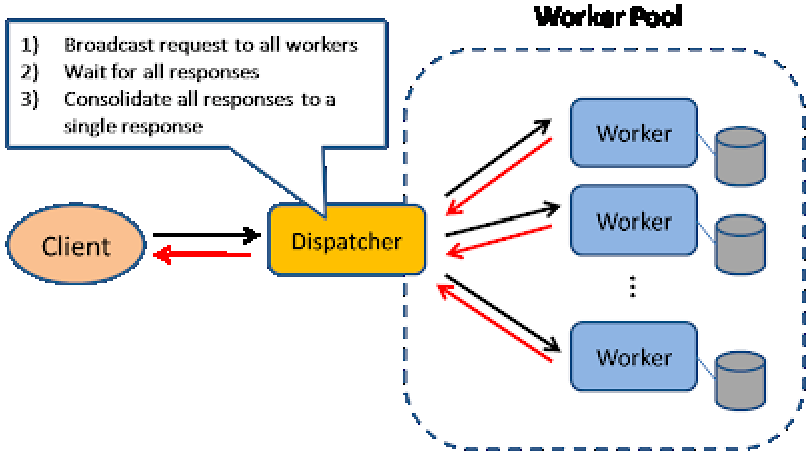
https://dzone.com/articles/scalable-system-design

Load Balancer



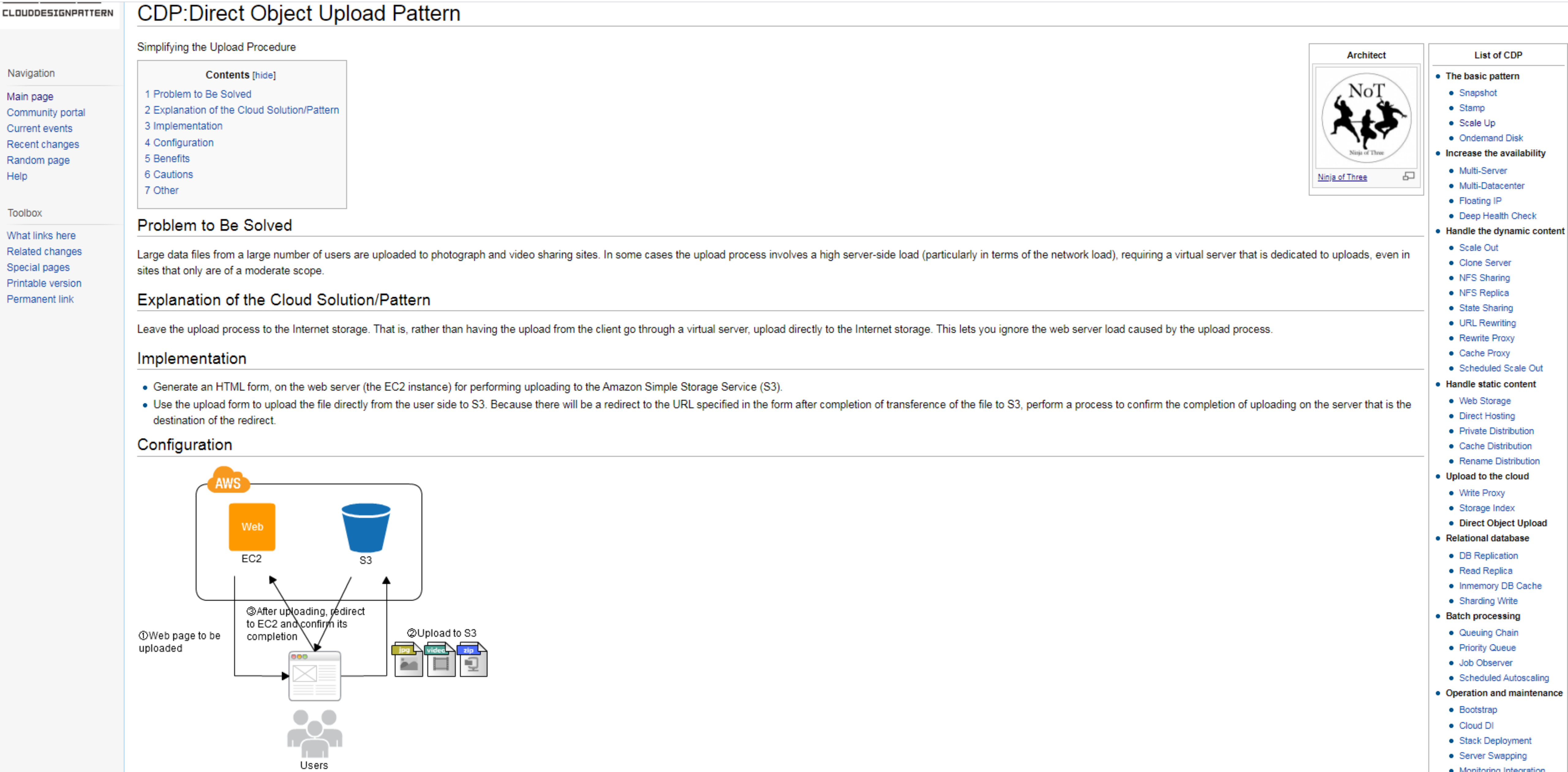
1) Pick a worker to forward request
   - Random
   - Round robin
   - Least busy
   - Sticky session / cookies
   - By request parameters
2) Wait for its response
3) Forward the response to client

Scatter and Gather



1) Broadcast request to all workers
2) Wait for all responses
3) Consolidate all responses to a single response

The AWS Cloud Design Patterns (CDP) are a collection of solutions and design ideas for using AWS cloud technology to solve common systems design problems. To create the CDPs, we reviewed many designs created by various cloud architects, categorized them by the type of problem they addressed, and then created generic design patterns based on those specific solutions. Some of these problems could also be addressed using traditional data-center technology, but we have included cloud solutions for these problems because of the lower cost and greater flexibility of a cloud-based solution.



CLOUDDESIGNPATTERN

Navigation

- Main page
- Community portal
- Current events
- Recent changes
- Random page
- Help

Toolbox

- What links here
- Related changes
- Special pages
- Printable version
- Permanent link

CDP:Direct Object Upload Pattern

Simplifying the Upload Procedure

Contents [hide]

Architect

Ninja of Three

List of CDP

- **The basic pattern**
  - Snapshot
  - Stamp
  - Scale Up
  - Ondemand Disk
- **Increase the availability**
  - Multi-Server
  - Multi-Datacenter
  - Floating IP
  - Deep Health Check
- **Handle the dynamic content**
  - Scale Out
  - Clone Server
  - NFS Sharing
  - NFS Replica
  - State Sharing
  - URL Rewriting
  - Rewrite Proxy
  - Cache Proxy
  - Scheduled Scale Out
- **Handle static content**
  - Web Storage
  - Direct Hosting
  - Private Distribution
  - Cache Distribution
  - Rename Distribution
- **Upload to the cloud**
  - Write Proxy
  - Storage Index
  - **Direct Object Upload**
- **Relational database**
  - DB Replication
  - Read Replica
  - Inmemory DB Cache
  - Sharding Write
- **Batch processing**
  - Queuing Chain
  - Priority Queue
  - Job Observer
  - Scheduled Autoscaling
- **Operation and maintenance**
  - Bootstrap
  - Cloud DI
  - Stack Deployment
  - Server Swapping
  - Monitoring Integration

Problem to Be Solved

Large data files from a large number of users are uploaded to photograph and video sharing sites. In some cases the upload process involves a high server-side load (particularly in terms of the network load), requiring a virtual server that is dedicated to uploads, even in sites that only are of a moderate scope.

Explanation of the Cloud Solution/Pattern

Leave the upload process to the Internet storage. That is, rather than having the upload from the client go through a virtual server, upload directly to the Internet storage. This lets you ignore the web server load caused by the upload process.

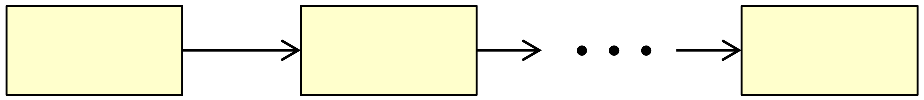
Implementation

- Generate an HTML form, on the web server (the EC2 instance) for performing uploading to the Amazon Simple Storage Service (S3).
- Use the upload form to upload the file directly from the user side to S3. Because there will be a redirect to the URL specified in the form after completion of transference of the file to S3, perform a process to confirm the completion of uploading on the server that is the destination of the redirect.

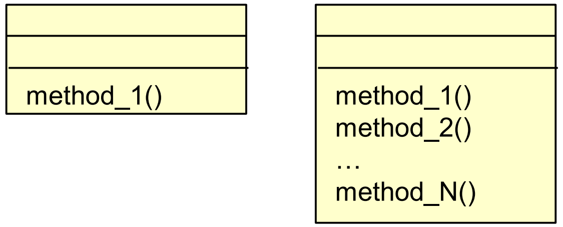
Configuration

Characteristics of Good Designs

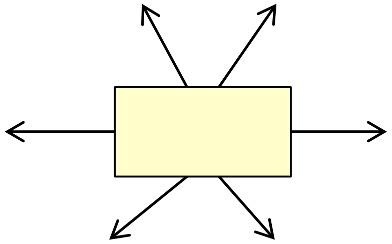
- Short communication chains between the objects



- Balanced workload across the objects



- Low degree of connectivity (associations) among the objects



Design Issues

Any design problem you are facing may have an associated pattern that can be applied

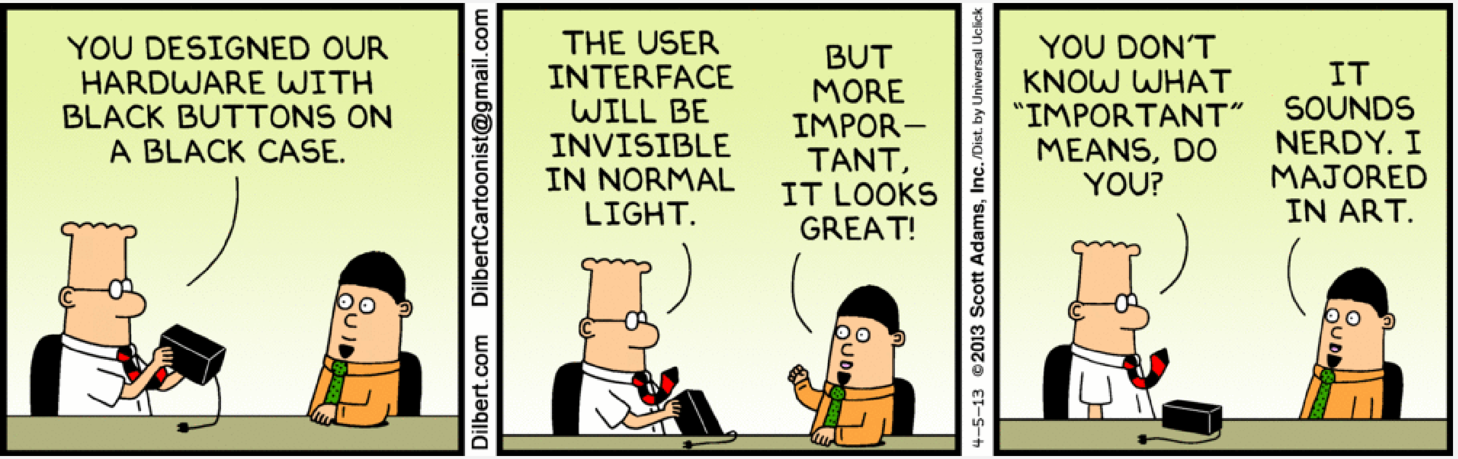
- Tell several objects that the state of some other object has changed → **Observer**
- Tidy up the interfaces to a number of related objects that have often been developed incrementally → **Façade**
- Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented → **Iterator**

Implementation Issues

Besides programming, other implementation issues may be:

- **Reuse** → Most modern software is constructed by reusing existing components or systems. You should make use of existing code
- **Configuration management** → You have to keep track of the many different versions of each software component
- **Host-target development** → You usually develop software on one computer (the host system) and execute it on a separate computer (the target system)

DOGBERT THE PRODUCT DESIGNER
YOU MIGHT THINK MY JOB IS TO MAKE PRODUCTS THAT ARE EASY TO USE.
Dilbert.com DilbertCartoonist@gmail.com
BUT THAT WOULDN'T HELP ME. SO INSTEAD, I DESIGN STUFF THAT LOOKS GOOD IN MY PORTFOLIO BUT IS IMPOSSIBLE TO USE.
3-23-15 © 2015 Scott Adams, Inc. /Dist. by Universal Uclick
THIS LOOKS GREAT, BUT NO ONE WILL BE ABLE TO SEE BLACK BUTTONS ON A BLACK CASE.
NOT MY PROBLEM.


YOU DESIGNED OUR HARDWARE WITH BLACK BUTTONS ON A BLACK CASE.
Dilbert.com DilbertCartoonist@gmail.com
THE USER INTERFACE WILL BE INVISIBLE IN NORMAL LIGHT.
BUT MORE IMPOR–TANT, IT LOOKS GREAT!
4-5-13 ©2013 Scott Adams, Inc. /Dist. by Universal Uclick
YOU DON'T KNOW WHAT "IMPORTANT" MEANS, DO YOU?
IT SOUNDS NERDY. I MAJORED IN ART.

THIS IS PHIL, OUR NEW VICE PRESIDENT OF MARGINALLY LEGAL ACTIVITIES.
www.dilbert.com scottadams@aol.com
HE'LL BE LEADING THE EFFORT TO MAKE OUR USER INTERFACES SO CONFUSING THAT PEOPLE HAVE TO PAY US FOR TRAINING.
3-31-09 © 2009 Scott Adams, Inc./Dist. by UFS, Inc.
WE ALREADY DO THAT UNINTEN–TIONALLY.
SURE, BUT WE CAN'T ALWAYS RELY ON LUCK.

NEXT WEEK on SE – UI/UX

Disclaimer

Content is adapted from Ian Sommerville’s book slides, Ivan Marsic’s lecture slides at Rutgers University, and William Y. Arms’ lecture slides from Cornell University

Thank You

mervat.abuelkheir@guc.edu.eg