Patterns

Architectural patterns:

Module patterns

- Layer

Component & Connector patterns

Broker, MVC, Pipe-and-filter, Client-Server, Peer-to-peer, Service Oriented architecture, Publish-Subsribe, Shared-Data

Allocation patterns

Map-Reduce, Multi-tier

Design patterns:

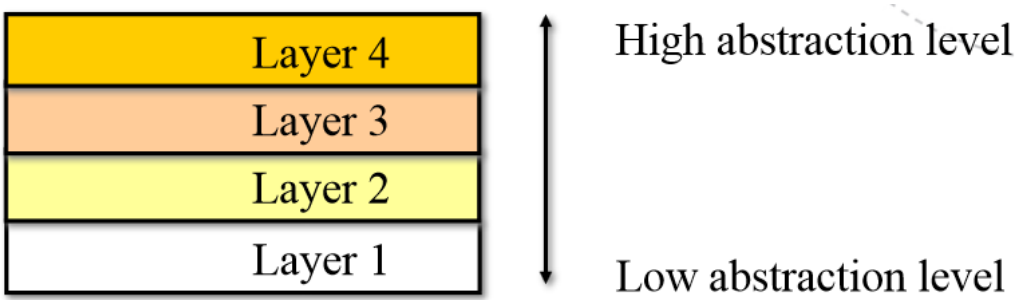
Singleton, Abstract factory, Composite, Observer, Template method, Mediator, Facade

Game and robot patterns:

Blackboard, CODGER, NASREM, Layered, Task Control, Task tree, Control loop

Layer pattern

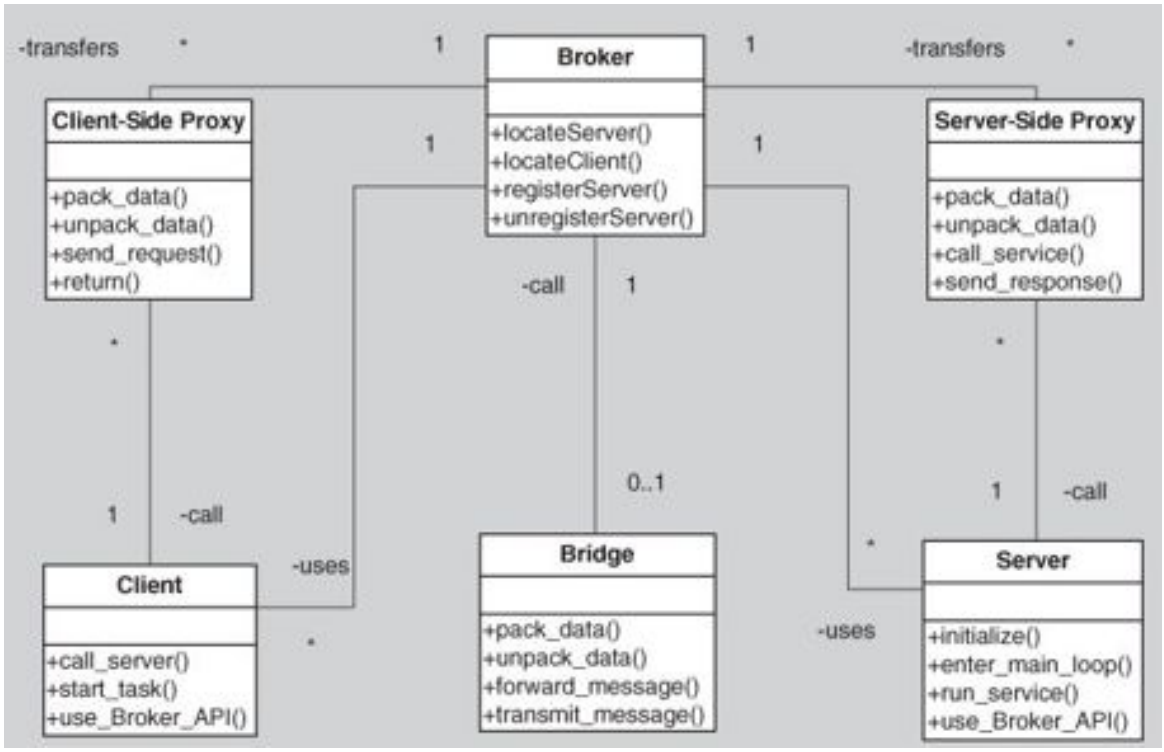
- **Context:** all complex systems experience the need to develop and evolve portions of the system independently. Need a clear and well documented separation of concerns, so that modules of the system may be independently developed and maintained
- **Problem:** The software needs to be segmented in such way that the modules can be developed and evolved sperately with little interaction among the parts, supporting portability, modifiability, and reuse.
- **Solution:** to achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.
    - **Overview:** the layered pattern defines layers and a undirectional allowed-to-use relation among the layers
    - **Elements:** Layer, a kind of module. The description of a layer should define what modules the layers contains
    - **Relations:** Allowed to use. The design should define what the layer usage rules are and any allowable wxceptions.
    - **Constraints:**
        - Every piece of software is allocared to exactly one layer
        - There are at least two layers
        - The allowed-to-use relations should not be circular



Broker pattern

- **Context:** Many systems are constructed from a collection of services distributed across multiple servers. Implementing these system is complex because you need to worry about how the systems will interoperate - how they will connect to each other and how they will exchange information - as well as the availability of the component service.
- **Problem:** How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users?
- **Solution:** the broker pattern separates users and services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

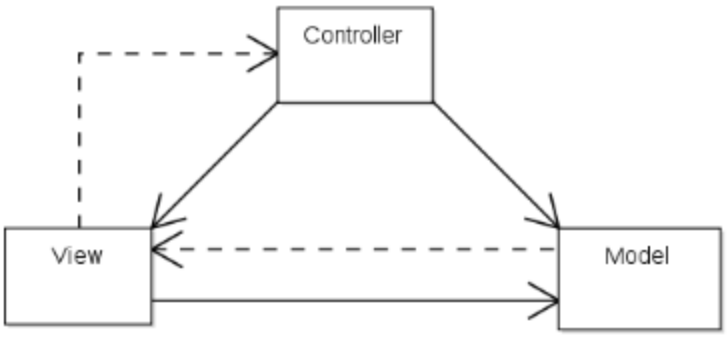
- **Overview:** The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
- **Elements:**
    - Client, a request of services
    - Server, a provider of services
    - Broker, a intermediary that locates an approperiate server to fulfill a client's request, forwards the request to the server, and returns the result to the client.
    - Client-side proxy, an itermediary that manages the actural communication with the broker, including marshaling, sending, and unmarshalig of messages.
    - Server-side proxy, an intemediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages.
- **Relations:** the attachment relation associates clients and servers with brokers
- **Constraints:** the client can only attach to a broker. The server can only attach to a broker.
- **Weaknesses:**
    - Brokers add a layer of indirectional, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
    - The broker can be a single point of failure
    - A broker adds up-front complexity
    - A broker may be a target for security attacks
    - A broker may be difficult to test


-transfers
*
1
Broker
1
-transfers
*
Client-Side Proxy
+pack_data()
+unpack_data()
+send_request()
+return()
1
+locateServer()
+locateClient()
+registerServer()
+unregisterServer()
1
Server-Side Proxy
+pack_data()
+unpack_data()
+call_service()
+send_response()
-call
1
*
*
0..1
1
-call
1
-call
-uses
Client
+call_server()
+start_task()
+use_Broker_API()
*
Bridge
+pack_data()
+unpack_data()
+forward_message()
+transmit_message()
*
-uses
Server
+initialize()
+enter_main_loop()
+run_service()
+use_Broker_API()

MVC pattern

- Context: UI is a typically frequently modified. Users often wish to look at data from different perspectives. These representations should both reflect the current state of data.
- Problem: how can user interface functionally be kept separate from application functionality and yet still be responsive to user input,or to changes in the underlaying application data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlaying application data changes?
- Solution: the model-view-controller (MVC) pattern separates application functionally into three kinds of components (model, view, controller)

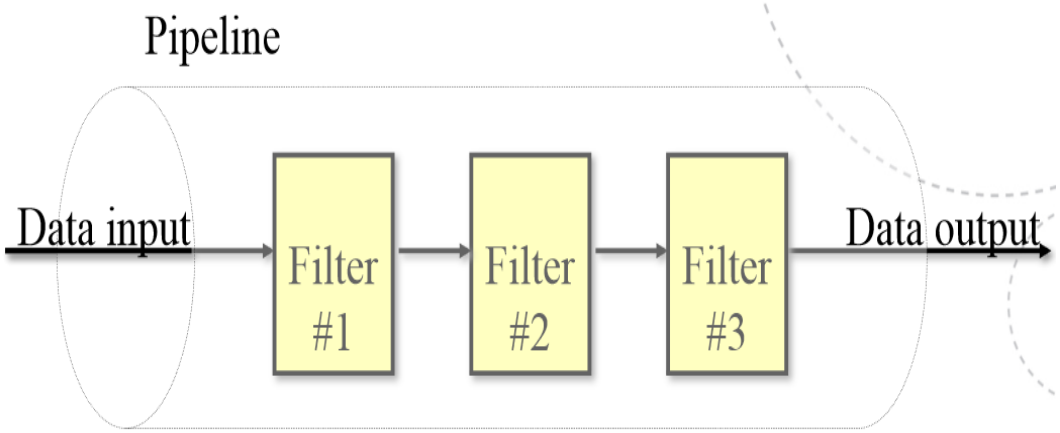
- **Overview:** the MVC pattern breaks system functionality into three components: a model, a view and a cotroller that mediates between the model and the view.
- **Elements:**
    - A model, which contains the applications data
    - A view, which displays some portion of the underlaying data and interacts with the user.
    - A controller, which mediates between the model and the view and manages the notifications of state change.
- **Relations:** the notifies relation connects instances of model, view, and controller, notifying elements of relevant state changes
- **Constraints**
    - There must be at least one instance of each model, view, and controller.
    - The model component should not interact directly with the controller.
- **Weaknesses:**
    - The complexity may not be worth it for a simple user interface
    - The model, view, and controller abstractions may not be good fits for some user interface toolkits.



Pipe-and-filter pattern

- Context: many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts
- Problem: such systems need to be divided into reusable, loosely coupled components with simplem, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.
- Solution: the pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

- **Overview:** data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected to pipes.
- **Elements:**
  - Filter, which is a component that transforms data read on its input port(s) to data written on its output port(s)
  - Pipe, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe ha a single source for its input and a single target for its output. A pipe perserves the sequence of data items, and it does not alter the data passing through
- **Relations:** the attachment relation associates the output of filters with the input of pipes and vice versa.
- **Constraints:**
  - Pipes connect filter output to filter input ports
  - Connected filters must agree on the type of data being passed along the connecting pipe.



Client-Server pattern

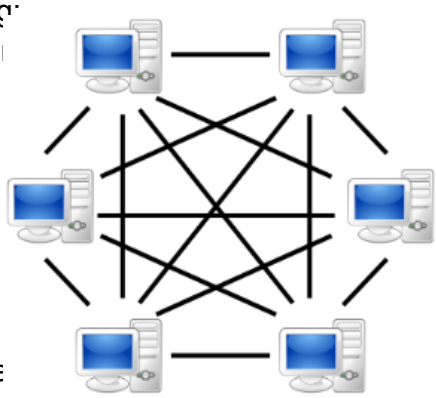
- Oveview: there are shared resources and services that lagre numbers of distributed clients wish to access, and for which we wish to control access or quality of service
- Problem: by managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small numer of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.
- Solution: clients interact by requesting services of servers, which provide a set of services. Som components may act as both client and servers. There may be one central server or multiple distributes ones.

- **Overview:** clients initiate interactions with server, invoking services as needed from those servers and waiting for the result of those requests
- **Elements**
    - Client, a component that invokes services of a server component. Clients have ports that describe the services they require.
    - Server, a component that provides services to clients. Servers have ports that describe the services they provide.
- **Request/reply connector:** a data connectpr employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are located or remote, and wheter data is encrypted.
- **Relations:** the attachment relation associates clients with servers
- **Constraints:**
    - Clients are connected to servers through request/reply connectors
    - Server components can be clients to other servers.
- **Weaknesses:**
    - Server can be a performance bottleneck
    - Server can be a single point of failure
    - Decisions about where to locate functionality are often complex and costly to change after a system has been built.

Peer-to-peer pattern

- Context: distributed computational entities - each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources - need to cooperate and collaborate to provide a service to a distributed community of users.
- Problem: How can a set of “equal” distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?
- Solution: in the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal” and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.

- **Overview:** computation is achieved by cooperating peers that request service from and provide services to one another across a network
- **Elements:**
  - Peer, which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
  - Request/reply connector, which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
- **Relations:** the relation associates peers with their connectors. Attachments may change at runtime.
- **Constraints:** Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searchig for a peer
  - Which peer know about which other peers
  - Som P2P networks are organized with star topologies, in which peers only connected to supernodes.

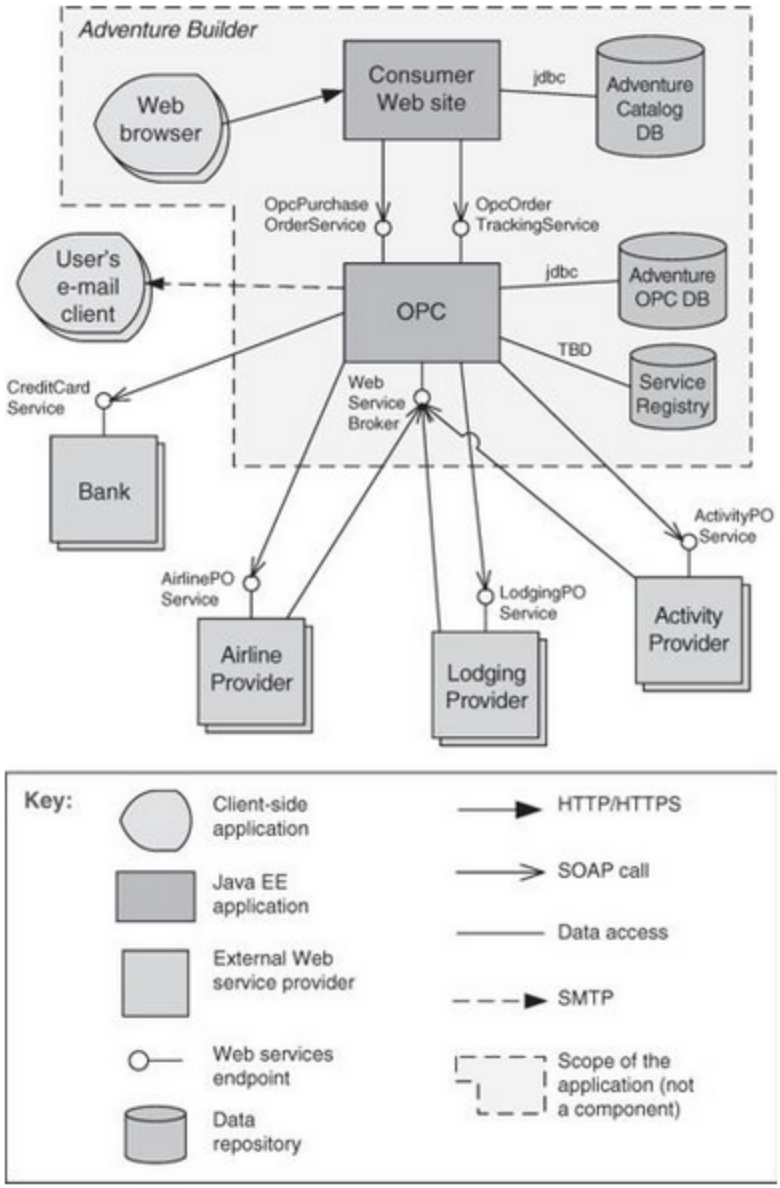


- **Weakness:**
  - Managing security, data consistency, data/service availability, backup, and recovery are all more complex
  - Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability

Service Oriented Architecture pattern

- Context: a number of services are offered (and described) by servide providers and consumed by service consumers. Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.
- Problem: how can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the internet?
- Solution: the service oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services

- **Overview:** computation is achieved by a set of cooperating components that provide and/or consume services over a network
- **Elements:**
    - Components: Service provider, which provide one or more services through published interfaces. Service consumers, which invoke services directly or through an intermediary Service providers may also be service consumers.
    - ESB, which is an intermediary element that can route and transform messages between service providers and consumers
    - Registry of services, which may be used by providers to register their services and by consumers to discover services at runtime.
    - Orchestration server, which coordinates the interactions between service consumers and providers based on language for business processes and workflow.
    - Connectors:
        - SOAP connector, which uses the SOAP prtocol for synchronous communication between web services, typically over HTTP.
        - REST connector, which relies on the basic request/reply operations of the HTTP protocol.
        - Asynchronous messaging connector, which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
- **Relations:** Attachment of the different kind of components available to the respective connectors
- **Constraints:** Service consumers are connected to service providers, but immediary components may be used.
- **Weaknesses:** Typically complex to build, Dont controll the evolution of independent services, Bottleneck


Adventure Builder
Web browser
Consumer Web site
jdbc
Adventure Catalog DB
OpcPurchase OrderService
OpcOrder TrackingService
User's e-mail client
OPC
jdbc
Adventure OPC DB
TBD
Service Registry
CreditCard Service
Web Service Broker
Bank
ActivityPO Service
AirlinePO Service
LodgingPO Service
Activity Provider
Airline Provider
Lodging Provider
Key:
Client-side application
Java EE application
External Web service provider
Web services endpoint
Data repository
HTTP/HTTPS
SOAP call
Data access
SMTP
Scope of the application (not a component)

Publish-Subscribe pattern

- Context: there are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not permitermined or fixed, nor is it the dataa that they share.
- Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?
- Solution: in the publish-describe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events.

- **Overview:** components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispateches the event to all registered subscribers
- **Elements:**
    - Any C&C component with at least one publish or subscribe port
    - The publish-subscribe connecor, which will have ennounce and listen roles for componens that wish to publish and subscribe events.
- **Relations:** the attachment relation associates components with the publish-subscribe connector by prescribing witch components announce events and which components registered to recieve events.
- **Constraints:** All components are connected to an event distributor that may be viewed as either a bus - connector - or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles.
- **Weaknesses:**
    - Typically increase latency and has a negative effect on scalability and predictability of messages delivery time.
    - Less control over ordering og messages, and delivery of messages is not guaranteed

Shared-Data pattern

- Context: various computational components need to share and manipulate lagre amounts of data. This data does not belong solely to any one of those components.
- Problem: how can systems store and manipulate persistent data that is accessed by multiple independent components?
- Solution: In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple data accessors and at least one shared-data store. Exchange may be initiated by the accessors or the data store. The connector tye is data reading and writing.

- **Overview:** Communication between data accessors is mediated by a shared data store. Control may be initiated by the data accessora or the data store. Data is made persistent by the data store.
- **Elements:**
    - Shared-data store. Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted.
    - Data accessor component
    - Data reading and writing connector
- **Relations:** attachment relation determines which data accessors are connected to which data stores.
- **Constraints:** Data accessors interact only with the data store(s).
- **Weaknesses:**
    - The shares-data store may be a performance bottleneck
    - The shared-data store may be a single point of failure
    - Producers and consumers of data may be thigthly coupled.

Map-Reduce pattern

- Context: business have a pressing need to quickly analyze enormous volumes of data they generate of access, at petabyte scale.
- Problem: For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map and reduce patterns solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple meands for the programmer to specify the analysis to be done.
- **Solution:** the map-reduce patterns requires three parts:

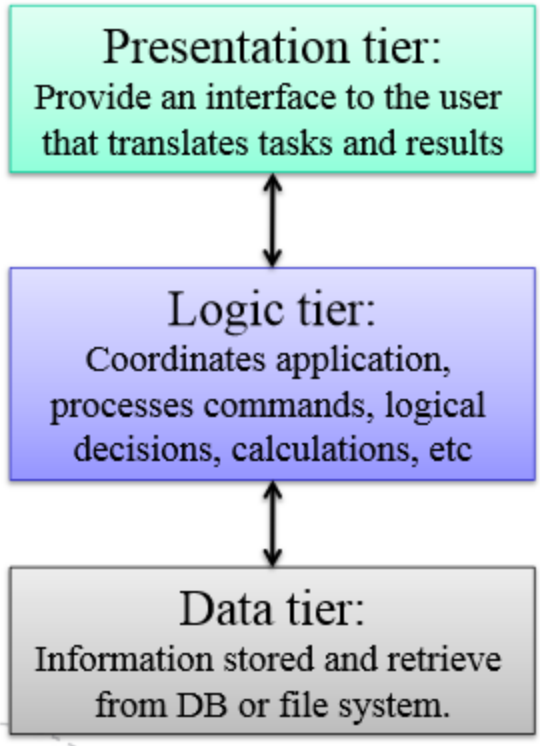
1. A specialiced infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and hanles sorting the data as needed.
2. A programmer specified component called the map reduce which filters the data to recive those items to be combined.
3. A programmer specified component called reduce which combines the result op the map.

- **Elements:**
    - Map is a function with multiple instances deployed across multiple processors that performs the extract and information portions of the analysis
    - Reduce is a function that may be deplyment as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.
    - The infrastructure is the framwork responsible for deploying map and reduce instances, shephering the data between them, and detecting and recovering from failure.
- **Relations:**
    - Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed
    - Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce
- **Constraints:**
    - The data to be analyzed must exist as a set of files
    - Map functions are stateless and do not communicate with each other.
    - The only communication between map reduce instances is the data emitted from the map instances as <key, value> pairs.
- **Weaknesses:**
    - If you do not have lagre data sets, the overhead of map-reduce is not justified.
    - If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost
    - Operations that require multiple reduces are complex to otchestrate.

Multi-tier pattern

- Context: in a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets
- Problem: how can we split the system into a number of computionally indenpendent execution structures (gropus of software and hardware) connected by some communications media?
- Solution: the execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a tier.

- **Elements:** Tier, which is a logical grouping of software components
- **Relations:**
    - Is a part of, to group components into tiers
    - Communicates with, to show how tiers and the components they contain interact with each other.
    - Allocated to, in the case that tiers map to computing platforms.
- **Constraints:** a software component belongs to exactly one tier
- **Weaknesses:** Substrantial up-front cost and complexity.



(3-tier, client-server where UI, logics and data are seperated)

Singleton

- Context: a class represents a concept that requires a single instance

- Problem:

- Clients could use this class in an inappropriate way
- It must be accessible to clients from a well-known access point.

- Solution:

- Make the class itself responsible for keeping track of its sole instance
- The class can ensure that no other instance can be created, and it can provide a way to access the instance

- Singleton advantages

- Single instance is controlled absolutely
- The class have direct control of how many instances can be created
- Reduced name space: improvement of global variable
- Easily extended to allow a controlled number of “singleton” objects to be created

- Singleton applied:

- The abstract factory, Builder. amd prototype patterns can use singletons in their implementation
- Facade objects are often singletons because only one Facade object is required
- State objects are often singleton

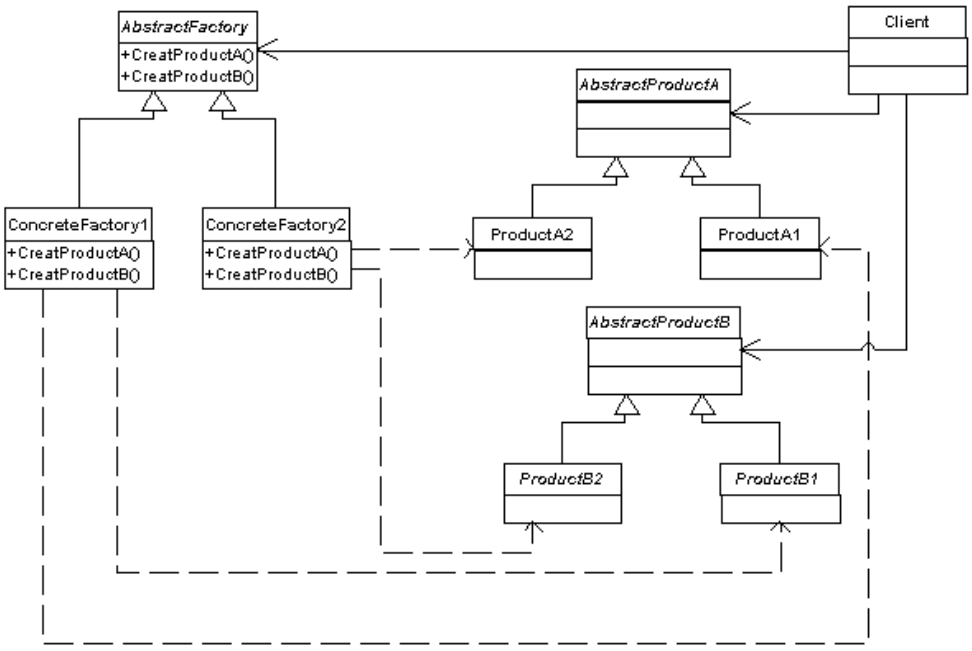
- Beware:

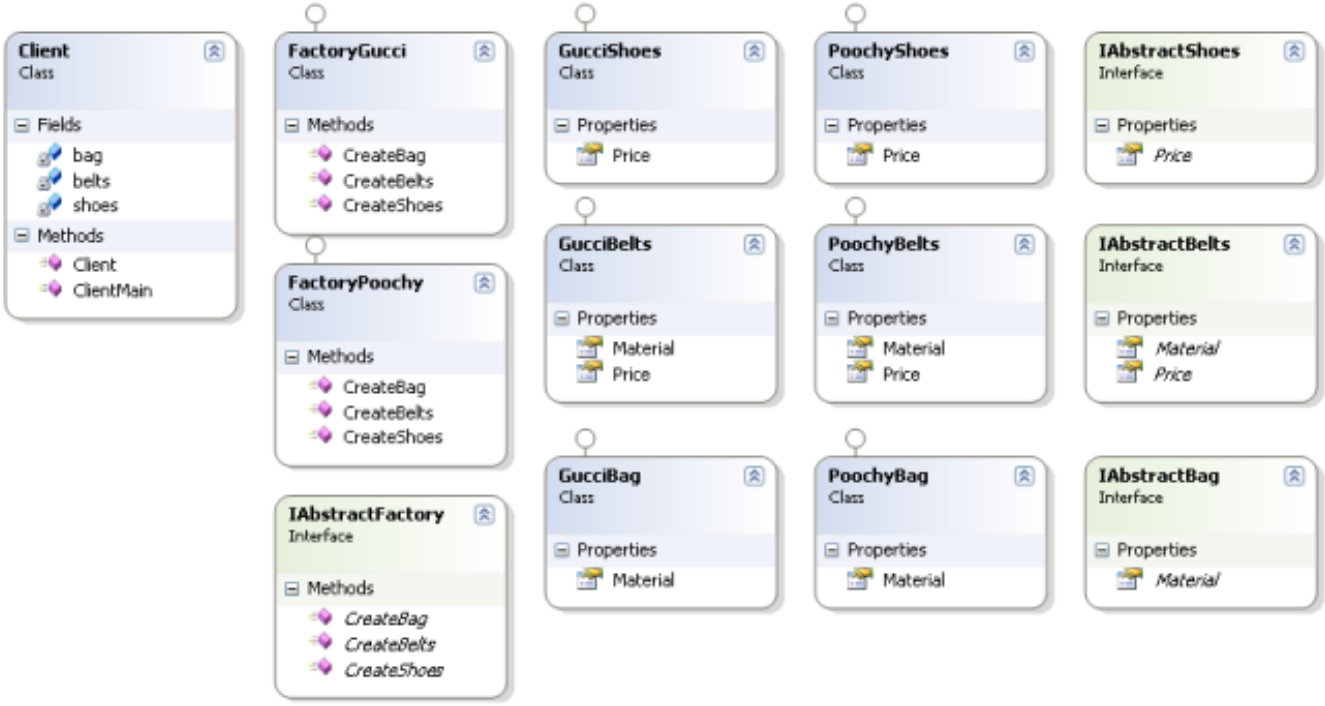
- Singleton is frequently misused, is sometimes seen as a legitimate way of doing “OO global variables”
- Complicates unit testing is stateful
- Consider threading

Abstract factory

- Context: a familty of realted classes can have different implementation detail

- Problem: the client should not know anything about which variant they are usin





Factory method:

- Context:

- A class cannot anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- Classes delegate responsability to one of several helper subclasses, and you want to localise the knowledge of which helper subclass is the delegate

- Problem

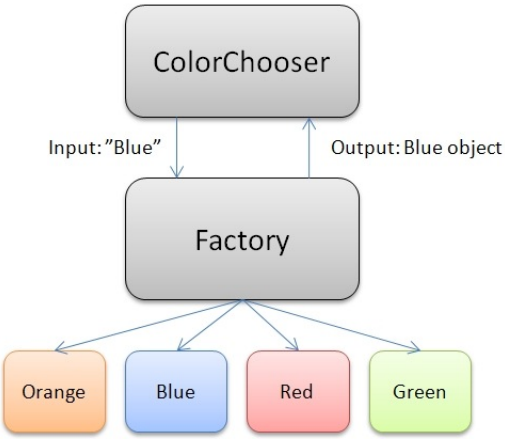
- How to create an object without knowing its concrete class?

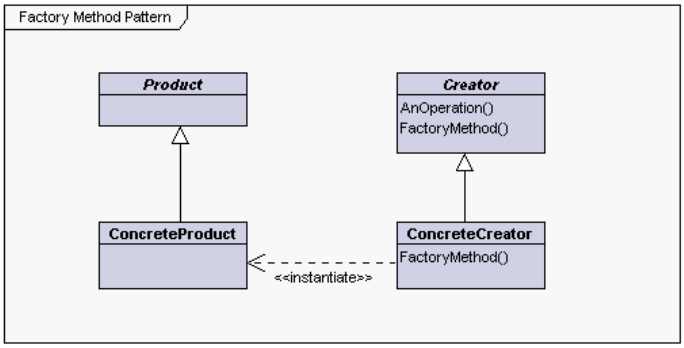
- Solution:

- Provide a method for creation at the interface level
- Defer the actual creation responsibility to subclasses.

- Factory method usage

- Factory methods are common in toolkits and frameworks
- Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from another
- Factory methods are used in test-driven development to allow classes to be put under test.





Composite

- Context

- You need to represent part-whole hierarchies of objects
- A client manipulating target objects either individually or grouped together

- Problem

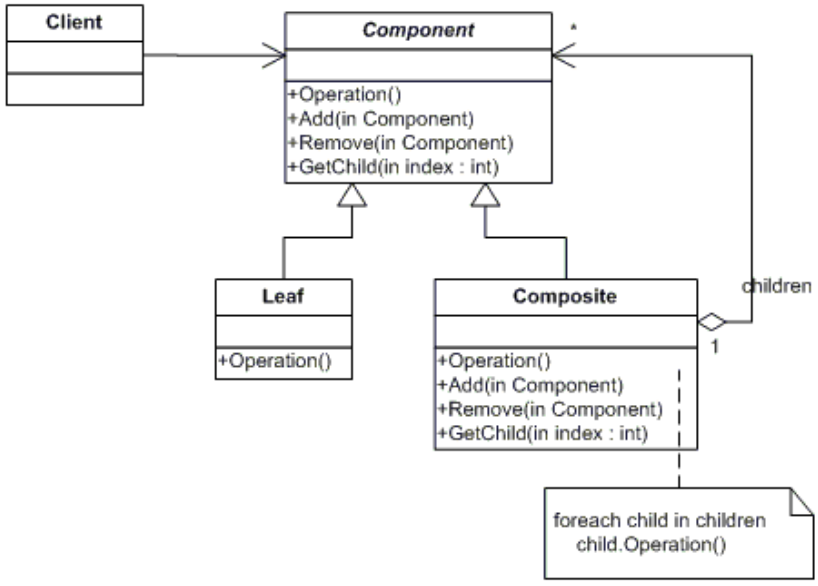
- Transparent treatment of single and multiple objects would simplify client code
- Not all differences between single and multiple object manipulation can be ignored
- Difference between composition objects and individual objects
- Allow arbitrary grouping of elements and uniform treatment of groups and primitives

- Solution:

- Common interface for group and primitives
- Introduce a composite object that implements the same interface as a leaf object through a common component interface or abstract class
- The composite object holds liks to many component objects, each of which may be either leaf objects or further composite objects, and forwards its operation to each of them.

- Consequences

- Composite and leaf objects are treated uniformaly
- The component interface may include operations specific to the composite that hold no meaning for the leaves.



Observer

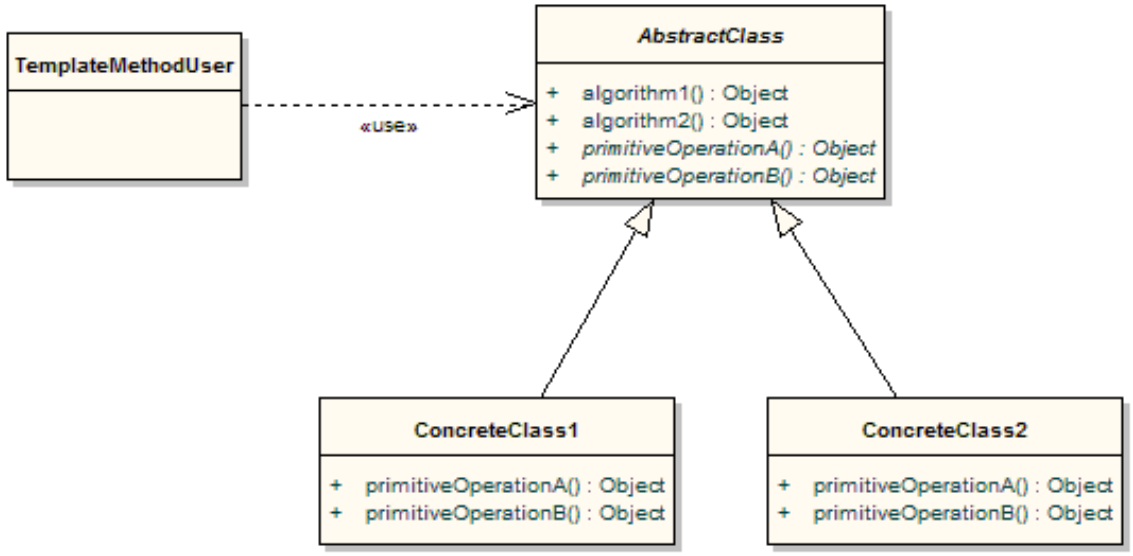
- Context: the change in one object may influence one or more other objects
- Problem:

- High coupling
- Number and type of objects to be notified may not be known in advance

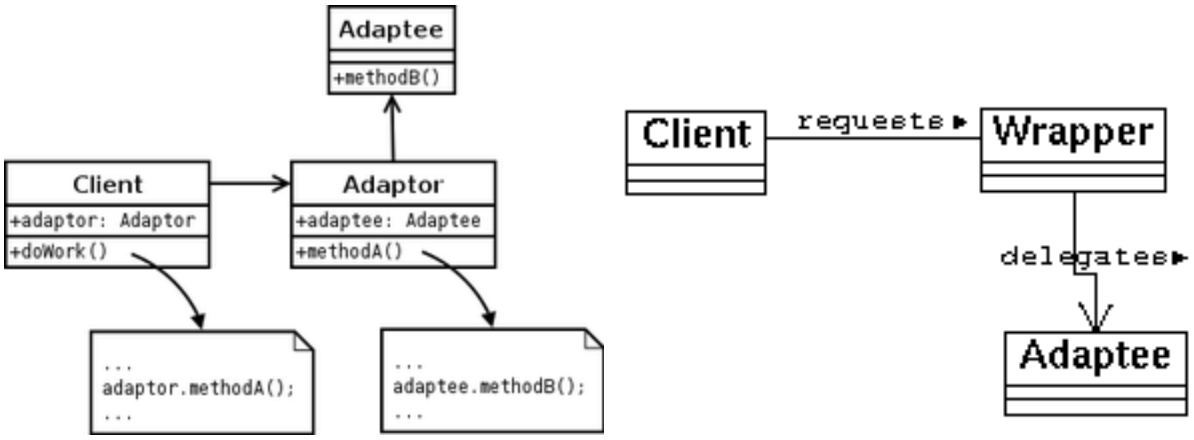
Template method

- Context: an algorithm/behaviour has a stable core and several variation at given points
- Problem: you have to implement/maintain several almost identical pieces of code
- Template method usage

- Let subclasses implement (overriding) behaviour that can vary
- Avoid duplication in the code
-

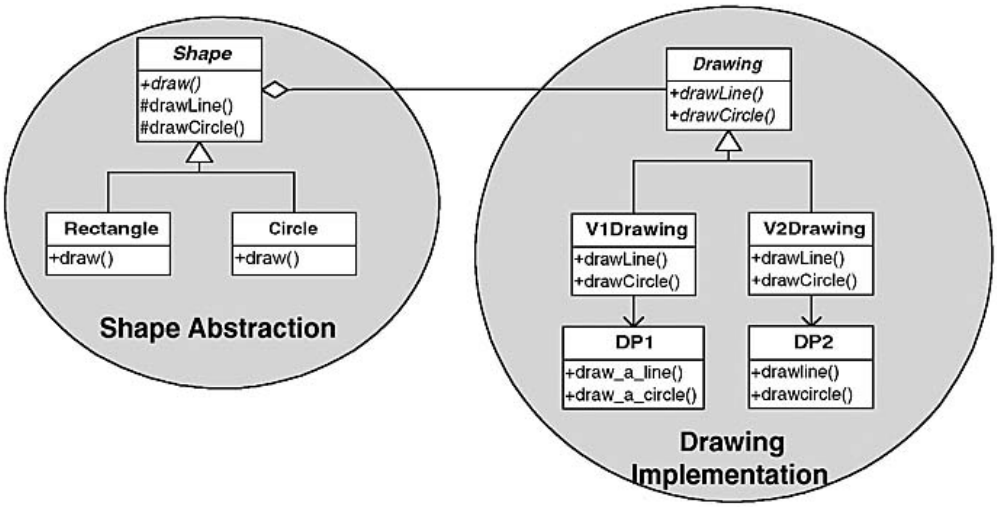


Adapter

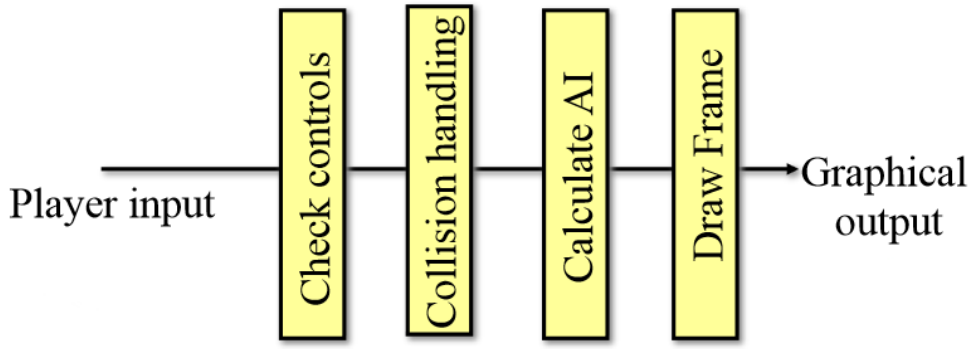


Mediator

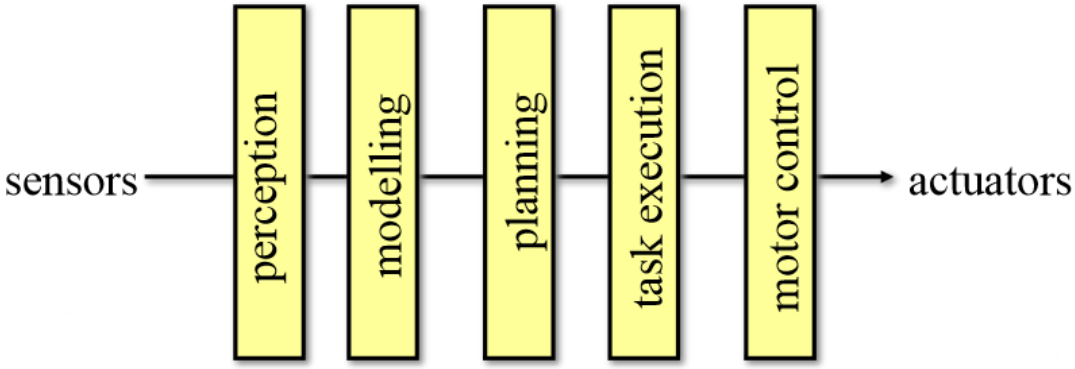
Bridge


Shape
+draw()
#drawLine()
#drawCircle()
Rectangle
+draw()
Circle
+draw()
Shape Abstraction
Drawing
+drawLine()
+drawCircle()
V1Drawing
+drawLine()
+drawCircle()
V2Drawing
+drawLine()
+drawCircle()
DP1
+draw_a_line()
+draw_a_circle()
DP2
+drawline()
+drawcircle()
Drawing Implementation

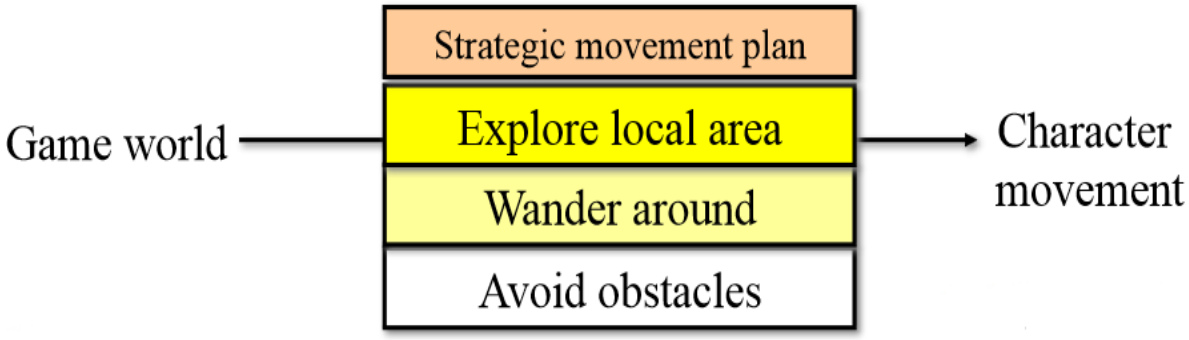
Pipe and filter control loop(Game)



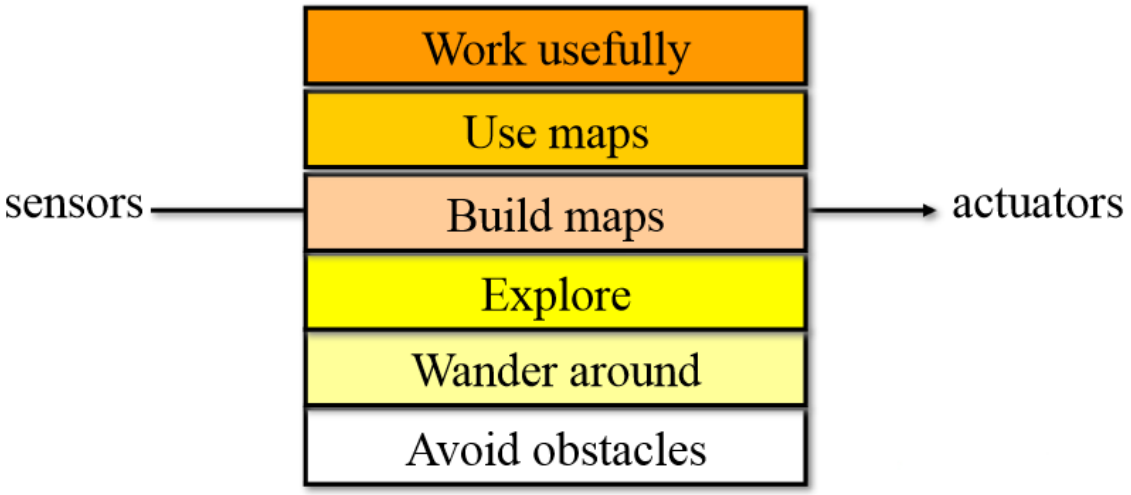
AI approach and pipe and filter (Robot)

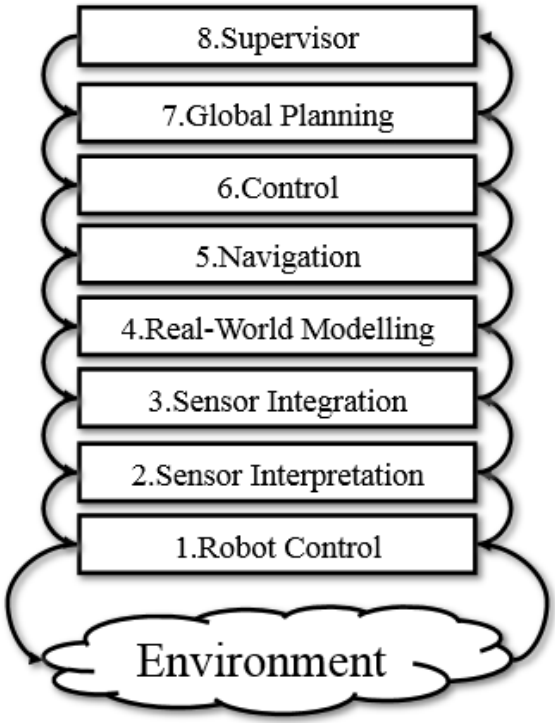


Layered architecture (game)



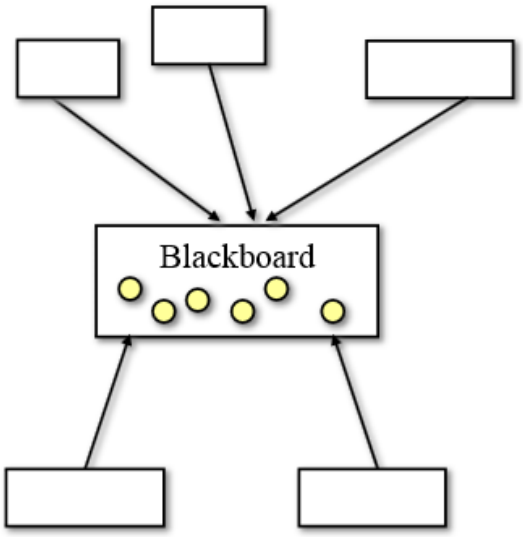
Layered architecture AI approach (Robot) → subsumption reference architecture



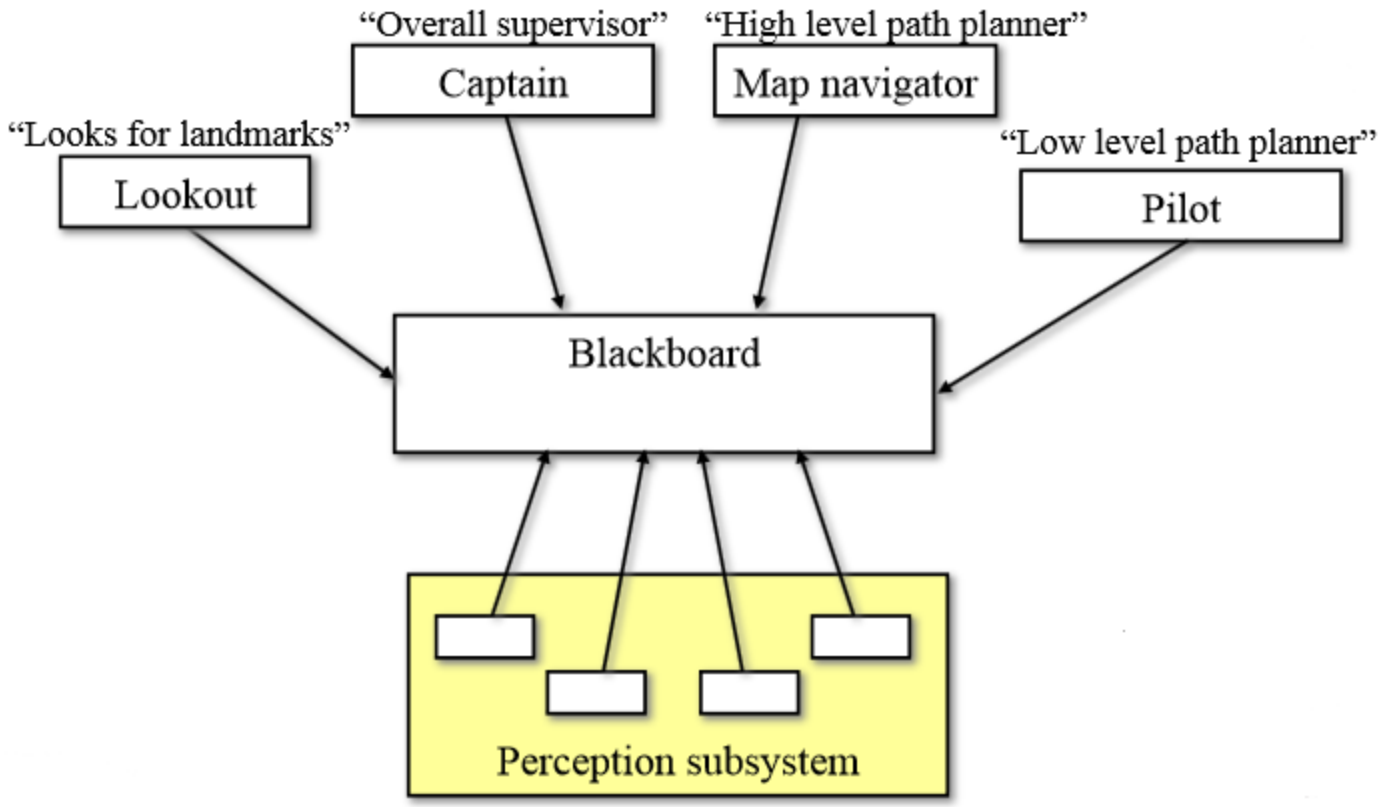


8: UI and supervision functions
7: High level scheduling and planning
6: Low level scheduling and planning
5: Managing navigation
4: Maintaining world model
3: Combined analysis of sensors
2: Individual sensor analysis
1: Provide robot control routines (motors etc.)

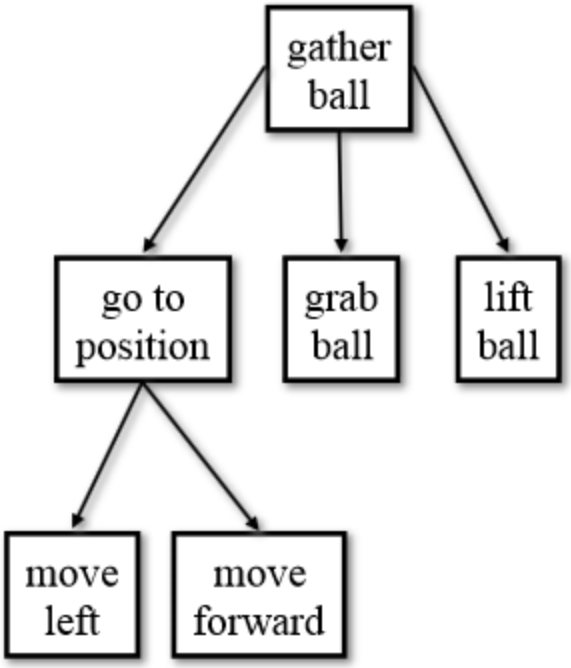
Blackboard



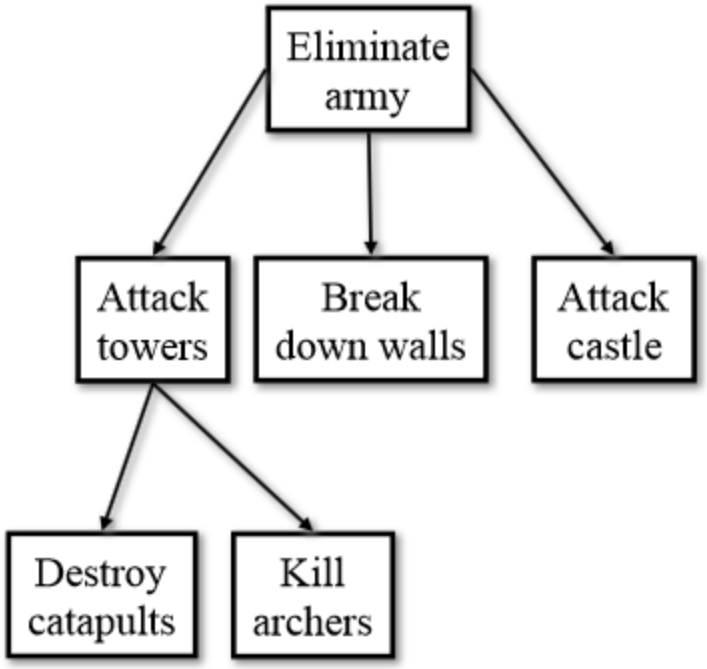
Blackboard CODGER (Robot)



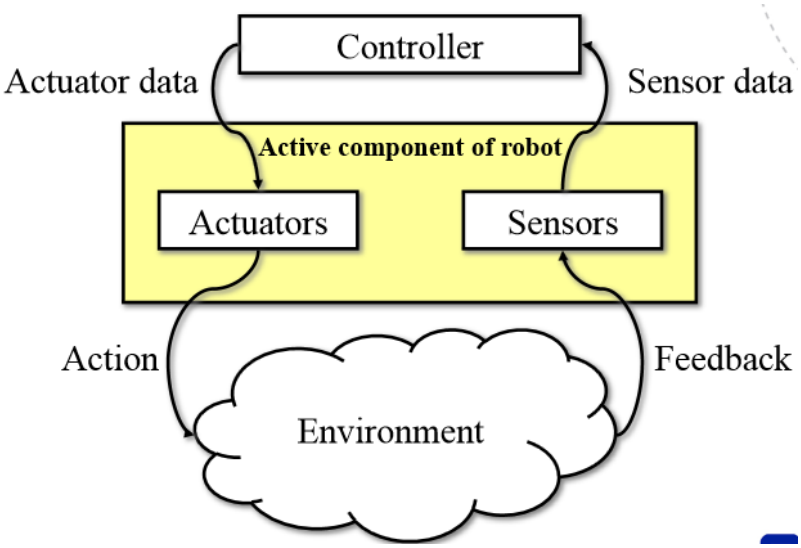
Task tree (Robot)



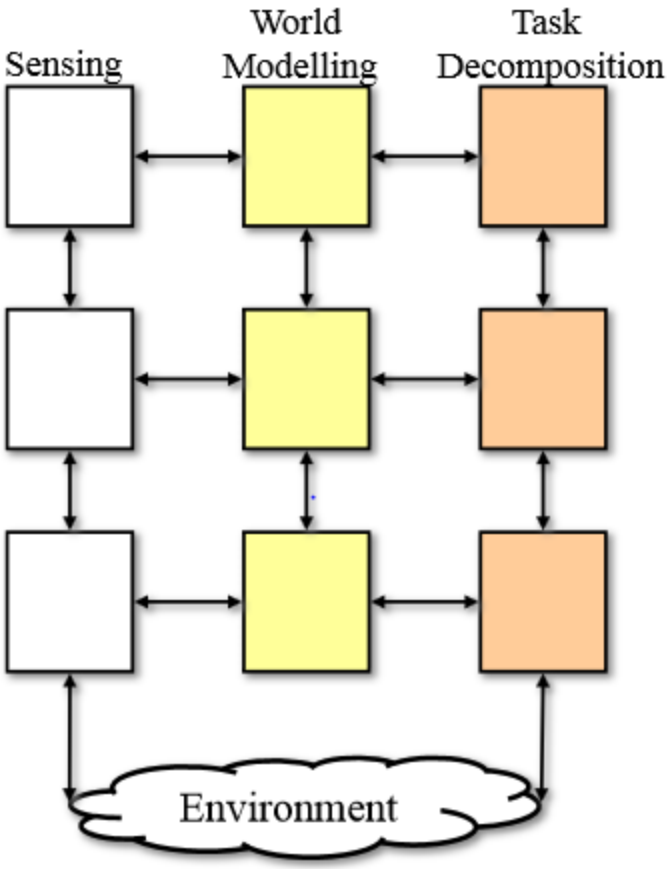
Task tree (Game)



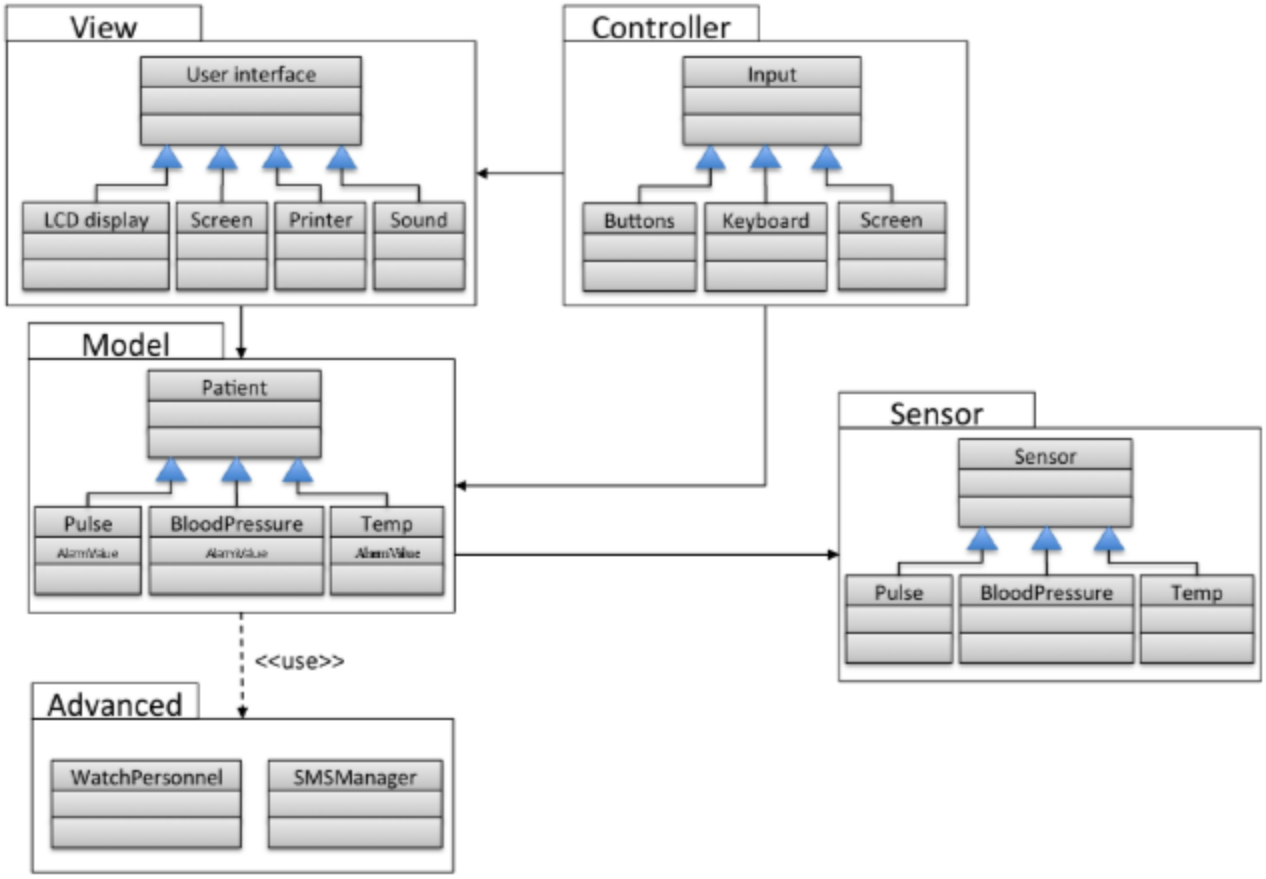
Control Loop (Robot)

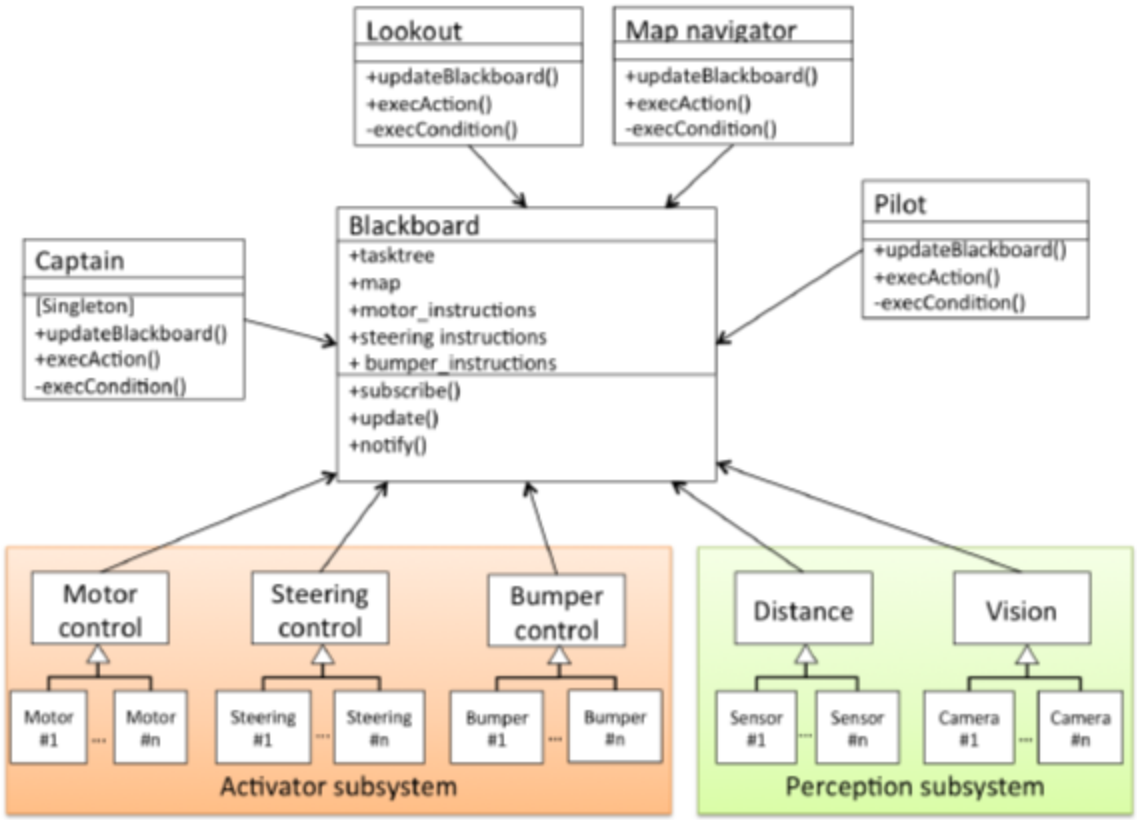


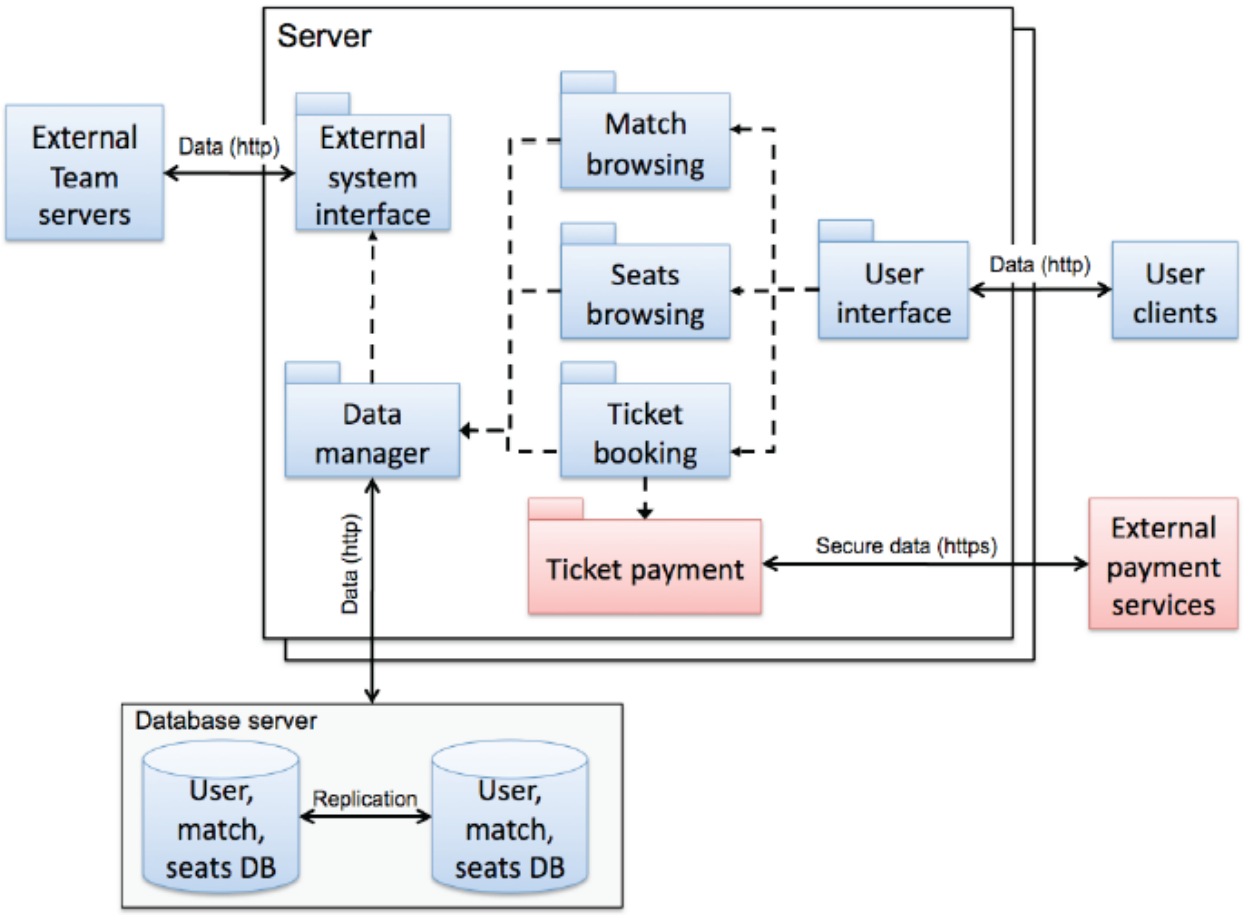
Task control NASREM (Robot)

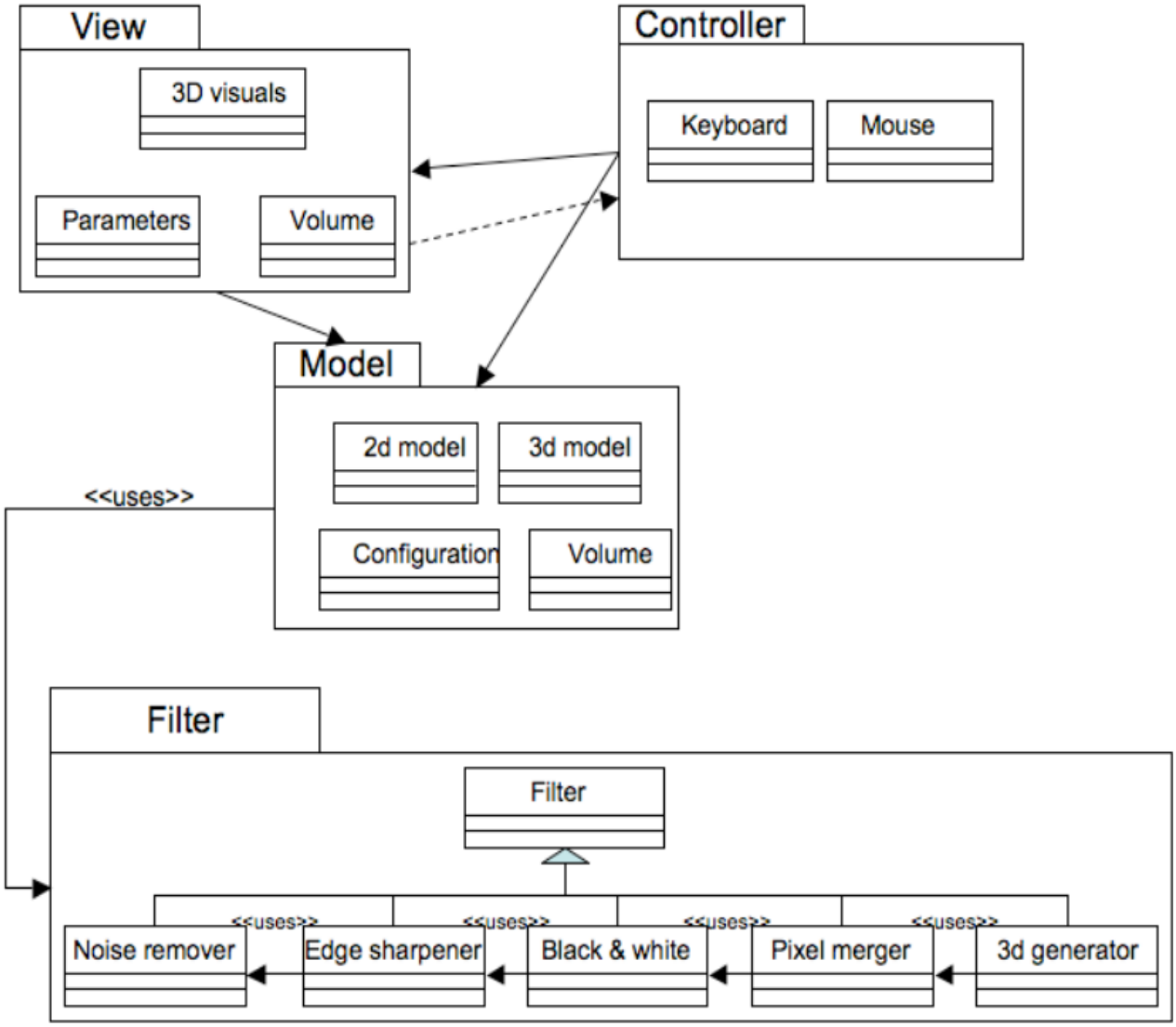


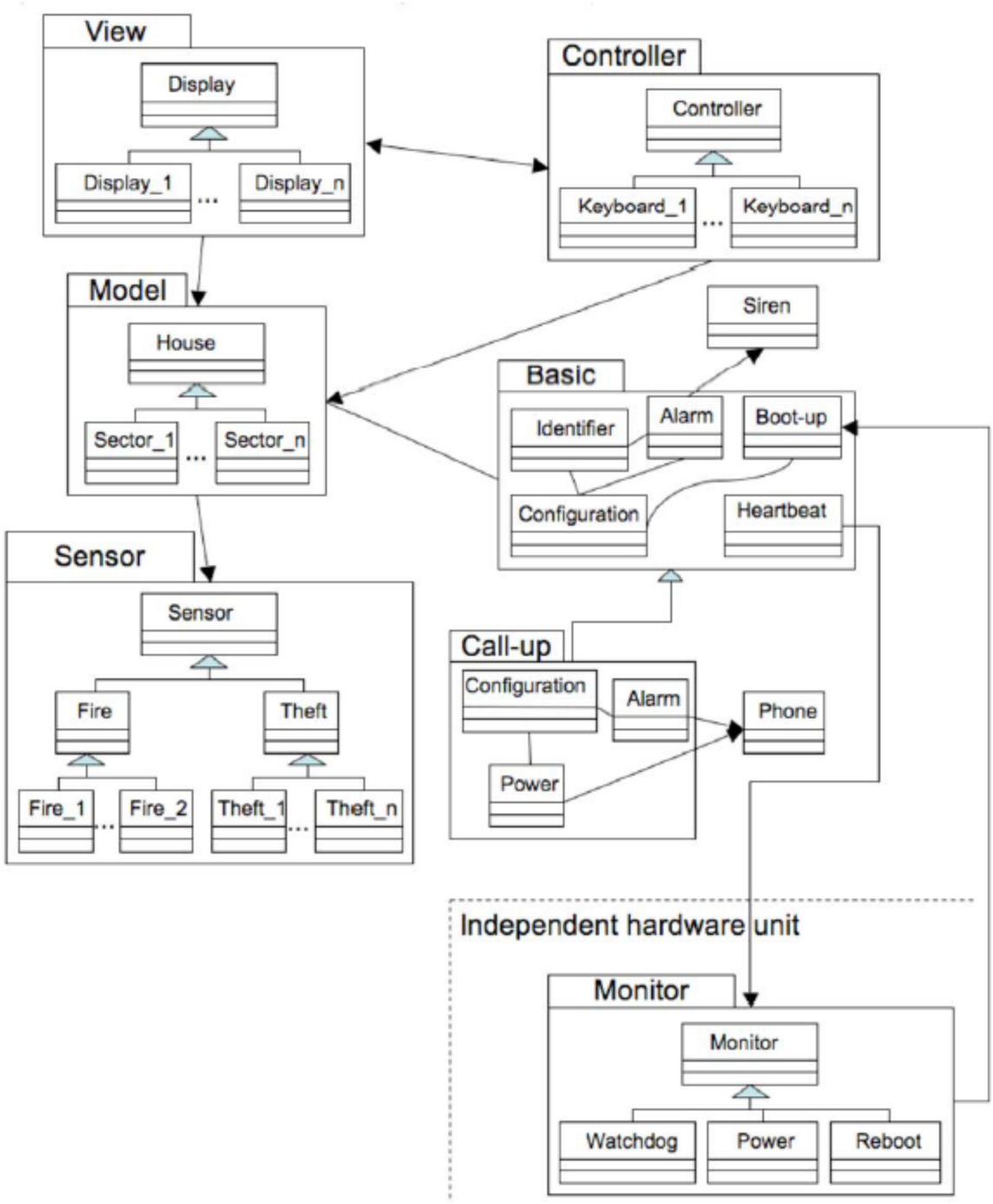
Eksempler i implementasjon/Views:

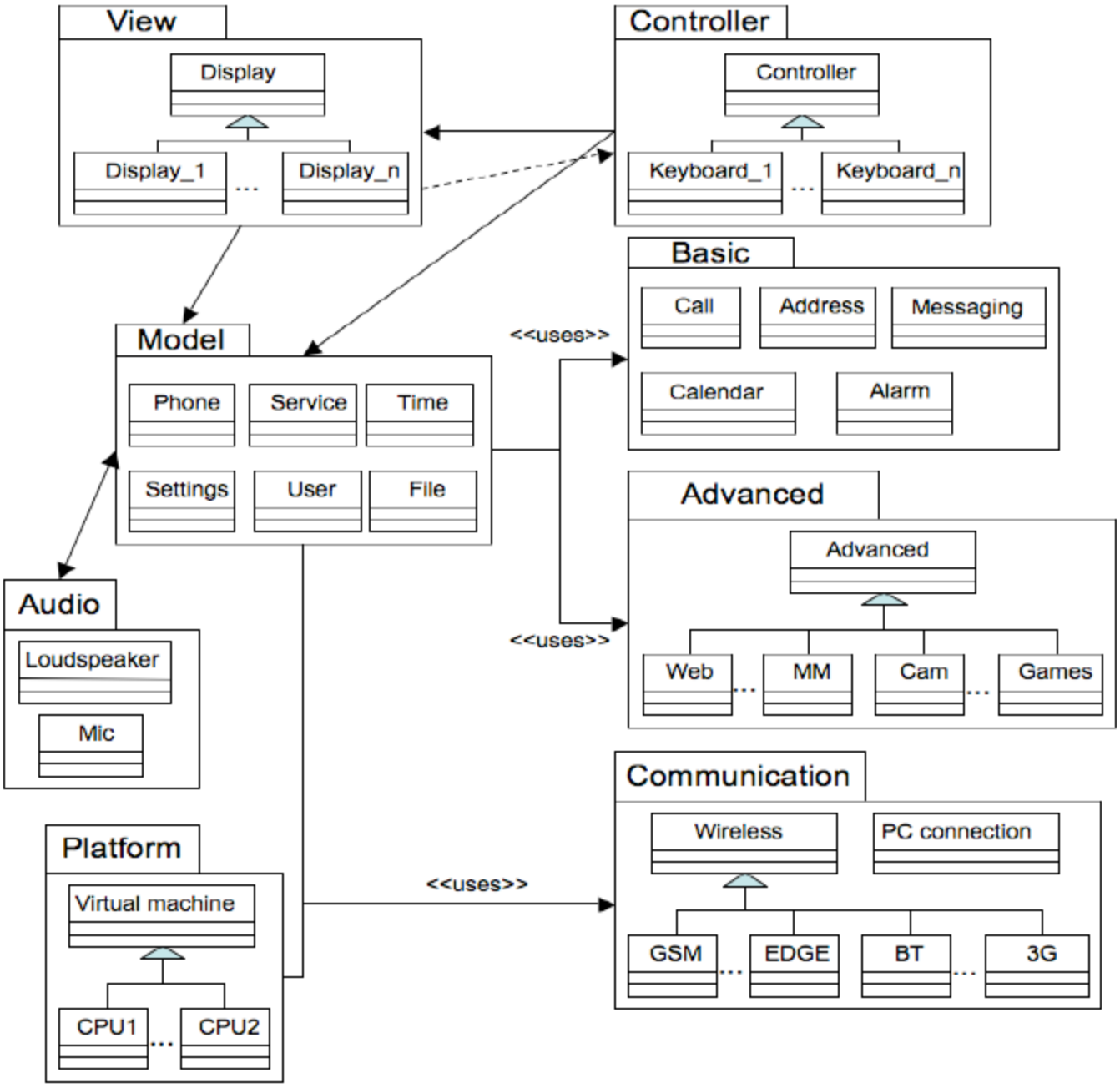


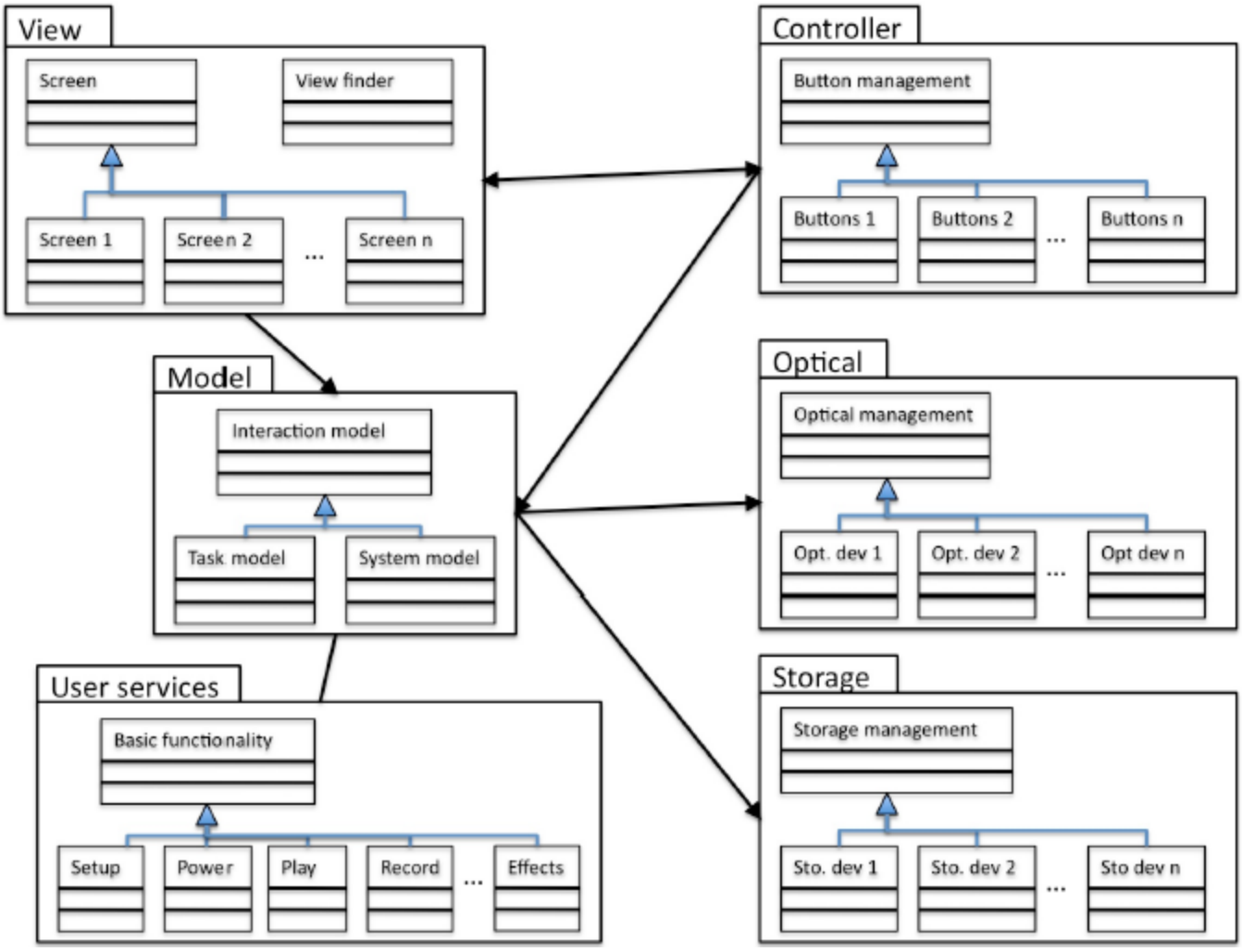

Lookout
+updateBlackboard()
+execAction()
-execCondition()
Map navigator
+updateBlackboard()
+execAction()
-execCondition()
Pilot
+updateBlackboard()
+execAction()
-execCondition()
Captain
[Singleton]
+updateBlackboard()
+execAction()
-execCondition()
Blackboard
+tasktree
+map
+motor_instructions
+steering instructions
+ bumper_instructions
+subscribe()
+update()
+notify()
Motor control
Steering control
Bumper control
Motor #1
Motor #n
Steering #1
Steering #n
Bumper #1
Bumper #n
Activator subsystem
Distance
Vision
Sensor #1
Sensor #n
Camera #1
Camera #n
Perception subsystem


Server
External Team servers
Data (http)
External system interface
Match browsing
Seats browsing
User interface
Data (http)
User clients
Data manager
Ticket booking
Ticket payment
Secure data (https)
External payment services
Data (http)
Database server
User, match, seats DB
Replication
User, match, seats DB


View
3D visuals
Parameters
Volume
Controller
Keyboard
Mouse
Model
2d model
3d model
Configuration
Volume
<<uses>>
Filter
Filter
<<uses>>
<<uses>>
<<uses>>
<<uses>>
Noise remover
Edge sharpener
Black & white
Pixel merger
3d generator


View
Display
Display_1
...
Display_n
Controller
Controller
Keyboard_1
...
Keyboard_n
Model
House
Sector_1
...
Sector_n
Siren
Basic
Identifier
Alarm
Boot-up
Configuration
Heartbeat
Sensor
Sensor
Fire
Theft
Fire_1
...
Fire_2
Theft_1
...
Theft_n
Call-up
Configuration
Alarm
Phone
Power
Independent hardware unit
Monitor
Monitor
Watchdog
Power
Reboot


View
Display
Display_1
Display_n
Controller
Controller
Keyboard_1
Keyboard_n
Basic
Call
Address
Messaging
Calendar
Alarm
Model
Phone
Service
Time
Settings
User
File
<<uses>>
<<uses>>
Advanced
Advanced
Web
MM
Cam
Games
Audio
Loudspeaker
Mic
Communication
Wireless
PC connection
GSM
EDGE
BT
3G
Platform
Virtual machine
CPU1
CPU2
<<uses>>


View
Screen
View finder
Screen 1
Screen 2
...
Screen n
Controller
Button management
Buttons 1
Buttons 2
...
Buttons n
Model
Interaction model
Task model
System model
Optical
Optical management
Opt. dev 1
Opt. dev 2
...
Opt dev n
User services
Basic functionality
Setup
Power
Play
Record
...
Effects
Storage
Storage management
Sto. dev 1
Sto. dev 2
...
Sto dev n