Architectural Patterns

Architectural patterns give us a starting point for designing a system in the large.

The basic idea is to describe basic forms of structural organisation for software systems in a way which makes it easy to recognise the sorts of problems to which they apply and the way in which solutions in that architecture map to problems. A pattern normally is considered mature if there are several examples of systems where it has been applied successfully. This gives us experience of its advantages and liabilities.

Read Buschmann *et.al.* 1996, *A System of Patterns*, Chapter 2.

Patterns We Shall Cover

High level decompositions :

- Layers
- Pipes and filters
- Blackboard

Distributed systems :

- Broker

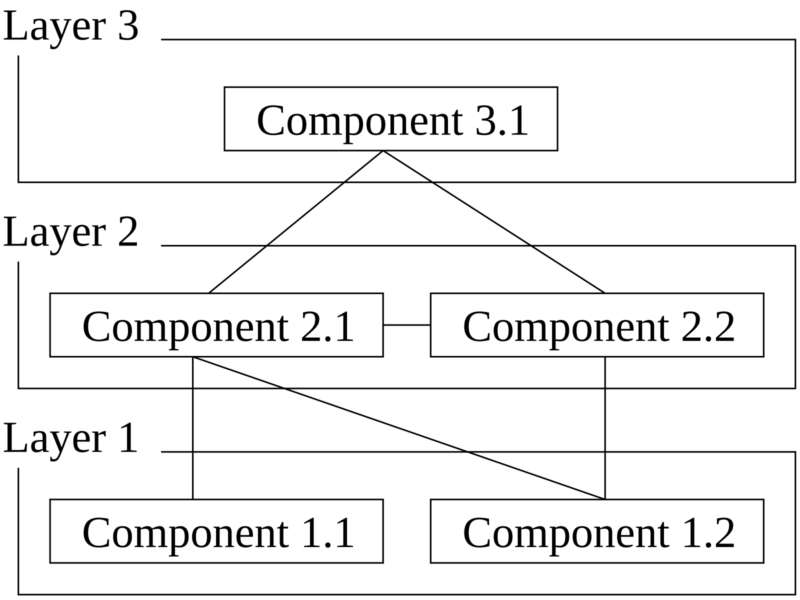
Interactive systems :

- Model-view-controller
- Presentation-abstraction-control

Adaptable systems :

- Microkernel
- Reflection

Layers: Pattern



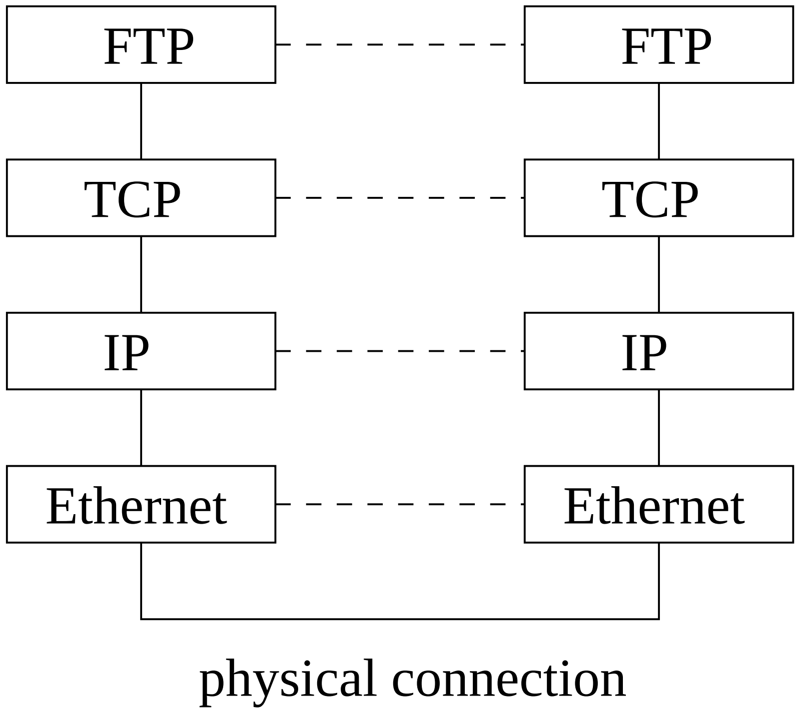
Layers: Problem

- System has different levels of abstraction.
- Typically requests downwards; notifications upwards.
- Want stable interfaces between layers.
- May want to change layers over time.
- May want to subdivide team programming.

Layers: Solution

- Start with the lowest level.
- Build higher layers on top.
- Same level of abstraction within a layer.
- No component spreads over two layers.
- Try to keep lower layers leaner.
- Specify interfaces for each layer.
- Try to handle errors at lowest layer possible.

Layers: Example (TCP/IP)



Layers: Advantages

- Reuse of layers.
- Standardisation of tasks and interfaces.
- Local dependencies between layers.

Layers: Liabilities

- Cascades of changing behavior.
- Lower efficiency of data transfer.
- Difficult to choose granularity of layers.

Pipes and Filters: Pattern

When processing a stream of data, each processing step is done by a filter and the filters are connected by pipes carrying data. Connecting filters in different combinations gives different ways of processing the data streams.

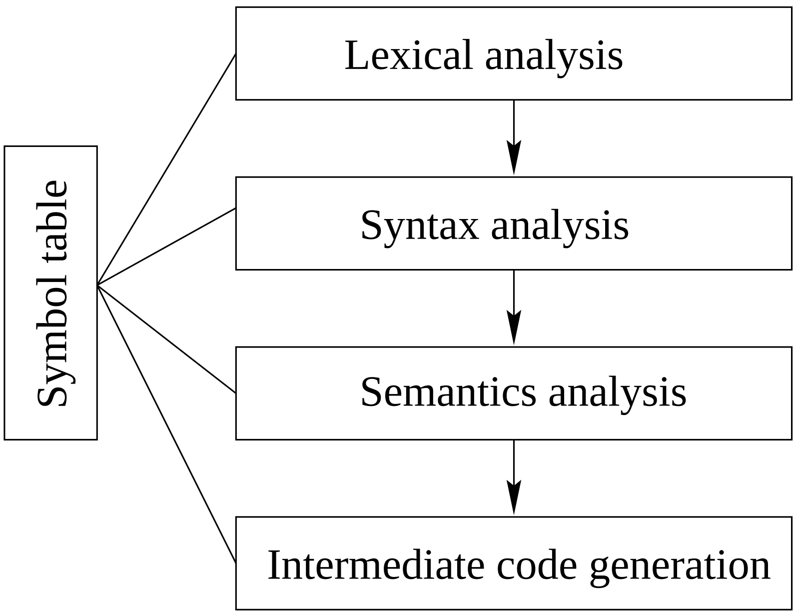
Pipes and Filters: Problem

- Data stream processing which naturally subdivides into stages.
- May want to recombine stages.
- Non-adjacent stages don't share information.
- Multi-processing of stages not ruled out.

Pipes and Filters: Solution

- Filter may consume/deliver data incrementally.
- Filters may be parameterisable (*e.g.* UNIX filters).
- Input from data source (*e.g.* a file).
- Output to a data sink.
- Sequence of filters from source to sink gives a processing pipeline.

Pipes and Filters: Example



Pipes and Filters: Advantages

- Pipes remove need for intermediate files.
- Can replace filters easily.
- Can achieve different effects by recombination.
- Incremental filters allow parallelisation.

Pipes and Filters: Liabilities

- Expensive to share global data.
- Parallelisation is less useful than it seems.
- Expensive if there are many small filters and a high data transfer cost.
- Difficult to know what to do with errors (especially if filters are incremental).

Blackboard: Pattern

A central, “blackboard” data store is used to describe a partial solution to a problem. A variety of knowledge sources are available to work on parts of the problem and these may communicate only with the blackboard, reading the current partial solution or suggesting modifications to it via a control component.

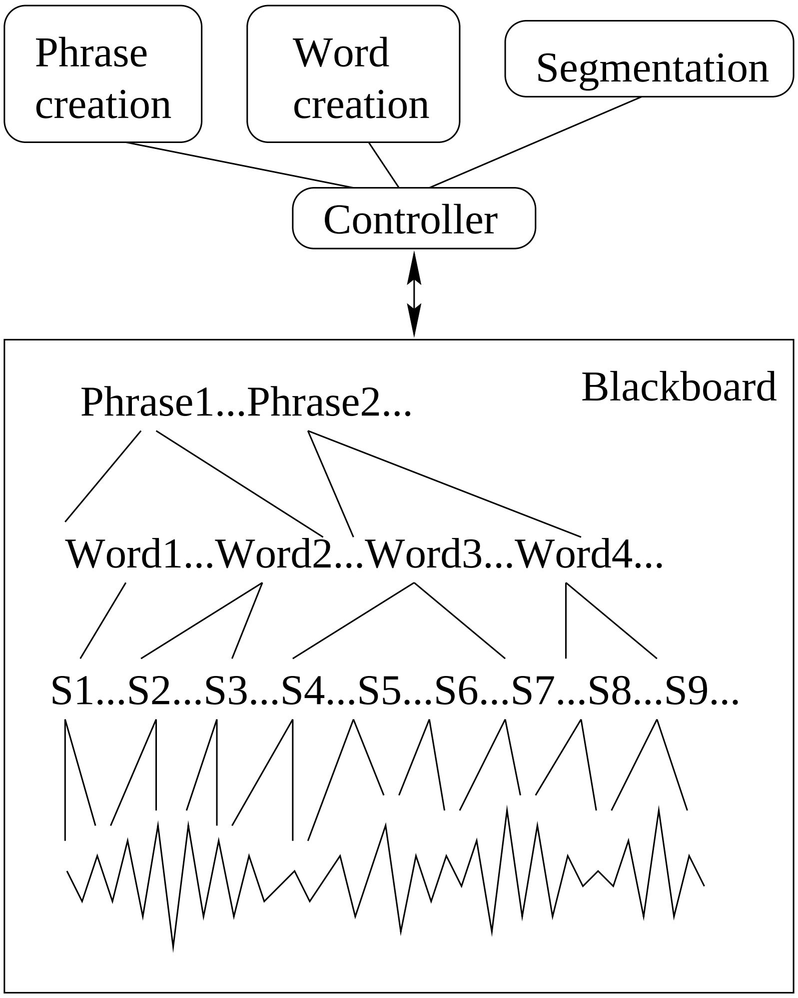
Blackboard: Problem

- No deterministic solution known for problem.
- Solutions to different parts of problem may require different representational paradigms.
- May be no fixed strategy for combining contributions of different problem solvers.
- May need to work with uncertain knowledge.

Blackboard: Solution

- Problem solvers work independently (and opportunistically) on parts of the problem.
- Share a common data structure (the blackboard).
- A central controller manages access to the blackboard.
- The blackboard may be structured (*e.g.* into levels of abstraction) so problem solvers may work at different levels.

Blackboard: Example



Blackboard: Advantages

- Easy (within limits) to experiment with different problem solvers and control heuristics.
- Knowledge sources may be re-usable.
- System may (within limits) be tolerant to dodgy problem solvers by protecting/revising the blackboard.

Blackboard: Liabilities

- Difficult to test.
- Difficult to guarantee an optimum solution.
- Control strategy often heuristic.
- May be computationally expensive.
- Parallelism possible but in practice we need much synchronisation.

Broker: Pattern

Decoupled components interact through remote service invocations. Communication is coordinated by a broker component which does things like forwarding requests and relaying results.

Broker: Problem

- Large scale system where scaling to many components is an issue.
- Components must be decoupled and distributed.
- Services required for adding, removing, activating, locating components at run time.
- Designers of individual components should not need to know about the others.

Broker: Solution

- Use brokers to mediate between clients and servers.
- Clients send requests to a broker.
- Brokers locate appropriate servers; forward requests; and relay results back to clients.
- May have client-side and/or server-side proxies.

Broker: Example

Multi-agent systems are often coordinated through brokers (*e.g.* JADE) which provide a standard mechanism for relaying messages based on a high-level communication protocol (*e.g.* FIPA-ACL). Individual agents may be implemented in any language as long as they can input/output according to the protocol.

Broker: Advantages

- Location transparency.
- Components easily adapted.
- Broker easily adapted.

Broker: Liabilities

- Low fault tolerance.
- Limited efficiency.
- Difficult to test.

Model-View-Controller: Pattern

Interactive system arranged around a model of the core functionality and data. View components present views of the model to the user. Controller components accept user input events and translate these to appropriate requests to views and/or model. A change propagation mechanism takes care of propagation of changes to the model.

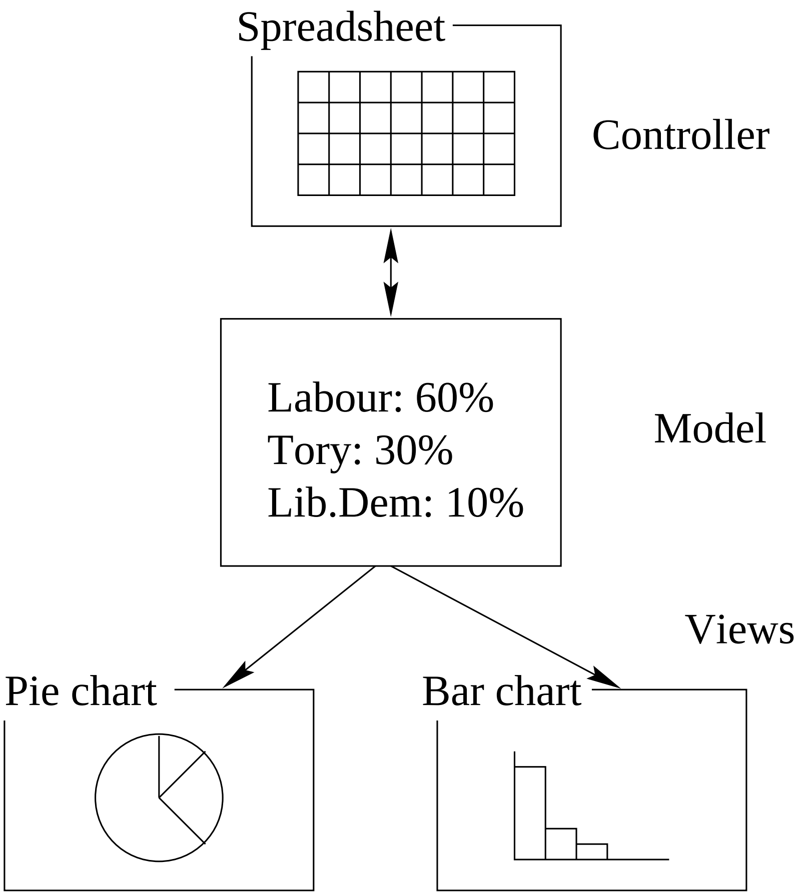
M-V-C: Problem

- User interfaces are prone to change requests.
- Different users ask for different changes.
- You may also want to support different “look and feel” standards.
- If the user interface is interwoven with the core functionality of the system then you may introduce errors in the core when changing the (many) interfaces.

M-V-C: Solution

- Develop a core model which is independent of style of input/output.
- Define different views needed for parts/whole model.
- Each view component retrieves data from the model and displays it.
- Each view component has an associated controller component to handle events from users.
- The model component notifies all appropriate view components whenever its data changes.

M-V-C: Example



Spreadsheet
Controller
Labour: 60%
Tory: 30%
Lib.Dem: 10%
Model
Views
Pie chart
Bar chart

M-V-C: Advantages

- Multiple views of the same model.
- View synchronisation.
- View components can be plugged in.
- Changes to interfaces without touching the model.

M-V-C: Liabilities

- Too complex for simple interface problems.
- Frequent events may strain simple change propagation mechanisms.
- Views and controllers aren't modular in practice.
- Changing the model is expensive if there are many views.

Presentation-Abstraction-Control: Pattern

System implemented as a hierarchy of cooperating agents. Each agent is responsible for a specific aspect of functionality. Each agent consists of: a presentation component responsible for its visible behaviour; an abstraction component which maintains the data model for the agent; and a control component which determines the dynamics of agent operation and communication.

P-A-C: Problem

- Interactive system viewed as a set of cooperating agents.
- Some agents specialise in HCI; others maintain data; others deal with error handling, *etc.*
- Some notion of levels of responsibility in the system.
- Changes to individual agents should not affect the whole system.

P-A-C: Solution

- Define top-level agent with core functionality and data model. It coordinates the other agents and may also coordinate user interaction.
- Define bottom-level agents for specific, primitive semantic concepts and/or services in the application domain.
- Connect top and bottom levels via intermediate agents which supply data to groups of lower level agents.
- For each agent separate core functionality from HCI

P-A-C: Example

Information system for political elections, with spreadsheet for entering data; various tables and charts for presenting current standings. Users interact through graphical interface but different versions exist for different user needs. Top-level agent holds data repository. Different bottom-level agents for different types of charting, analysis and error handling. Intermediate agent to coordinate views of system.

P-A-C: Advantages

- Separation of different concepts as individual agents.
- Changes to presentation or abstraction in an agent doesn't affect other agents.
- Easy to integrate/replace agents.
- Suits multi-tasking.
- Suits multi-user applications.

P-A-C: Liabilities

- Can have too many, too simple bottom-level agents.
- Can be difficult to get control right while maintaining independence of agents.
- Long chains of communication may be inefficient.

Microkernel: Pattern

For systems which must be easily adaptable to changing requirements, separate minimal functional core from extended functionality and customer-specific parts. Provide sockets in the microkernel for plugging in these extensions and coordinating them.

Microkernel: Problem

- Some software systems have long life spans, during which their environment evolves.
- Such systems must be adaptable to changes in existing platforms and must be portable to new platforms.
- There may be a large number of different but similar platforms.
- To conform with standards on different platforms it may be necessary to build emulators on top of the core functionality.
- So the core should be small.

Microkernel: Solution

- Build a microkernel component which encapsulates all the fundamental services of your application.
- The microkernel maintains system-wide resources (*e.g.* files); enables other components to communicate; and also allows other components to access its functionality.
- External servers implement their own view of the microkernel, using the mechanisms available from the microkernel.
- Clients communicate with external servers using the communication facilities of the microkernel.

Microkernel: Example

Operating system for desktop computers. Must be portable to relevant hardware platforms and adapt as these evolve. Must be able to run applications developed for other established operating systems - users being able to choose which OS from a menu. Each of these OSs is built as an external server on top of the microkernel.

Microkernel: Advantages

- Porting to a new environment normally doesn't need changes to external servers or clients.
- Extension by adding new external servers.
- Can distribute the microkernel to several machines, increasing availability and fault tolerance.

Microkernel: Liabilities

- Performance overhead in having multiple views of system.
- May be difficult to predict which basic mechanisms should be in the microkernel.

Reflection: Pattern

The application is split into two parts: the base level which gives the application logic; and the meta level which provides information about selected system properties and makes the software "self aware". Changes to the base level occur by changing information at the meta-level.

Reflection: Problem

- Modifying software can be error prone because a single change may propagate widely.
- If we have many techniques for doing modifications then maintenance engineers become confused.
- Sometimes small conceptual changes require large (though mundane) changes to the base level logic.

Reflection: Solution

- Build a system of meta level components which represent information about the base level logic.
- Only allow base level components to operate through this meta level.
- Then we can re-configure by changing the meta level, not the base level logic.

Reflection: Example

Prolog meta interpretation.

```
solve((X1,X2)) :- solve(X1),
                  solve(X2).
solve(true).
solve(X) :- clause(X,Y),
            solve(Y).

f_solve((X1,X2)) :- f_solve(X1),
                    f_solve(X2).
f_solve(X) :- clause(X,true).
f_solve(X) :- clause(X,Y),
            \+ Y = true,
            f_solve(Y).
```

Reflection: Advantages

- Most changes dealt with at meta level.
- This allows specific techniques to be hidden from engineers.
- Can change fundamental aspects of behaviour this way.

Reflection: Liabilities

- Who protects against erroneous changes to the meta level?
- Meta level description can become bloated.
- Reflection normally requires more processing.
- Not all languages support reflection.