Architectural Patterns

Dr. James A. Bednar

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Dr. David Robertson

dr@inf.ed.ac.uk

http://www.inf.ed.ac.uk/ssp/members/dave.htm

Design Patterns

A design pattern is a standardized solution to a problem commonly encountered during object-oriented software development (Gamma et al. 1995).

A pattern is not a piece of reusable code, but an overall approach that has proven to be useful in several different systems already.

Contents of a Design Pattern

Design patterns usually include:

- A pattern name
- A statement of the problem solved by the pattern
- A description of the solution
- A list of advantages and liabilities

  (good and bad consequences)

Architectural Patterns

The fundamental problem to be solved with a large system is how to break it into chunks manageable for human programmers to understand, implement, and maintain.

Large-scale patterns for this purpose are called *architectural patterns*. Design patterns are similar, but lower level and smaller scale than architectural patterns.

See Buschmann et al. (1996), chapter 2 for more details.

Architectural Pattern Examples

High level decompositions:

- Layers
- Pipes and filters
- Blackboard

Distributed systems:

- Broker

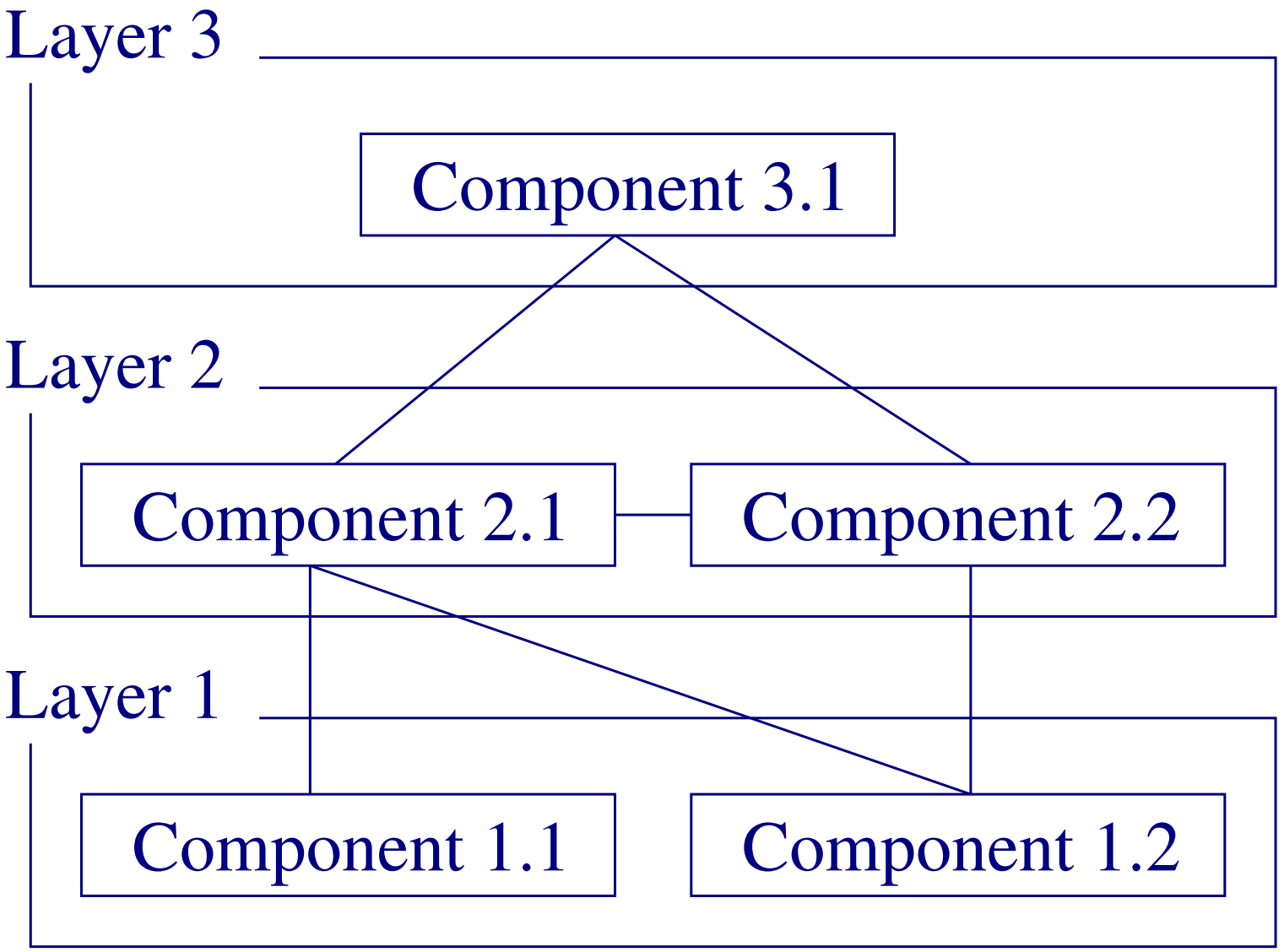
Interactive systems:

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

Adaptable systems:

- Microkernel

Layers: Pattern



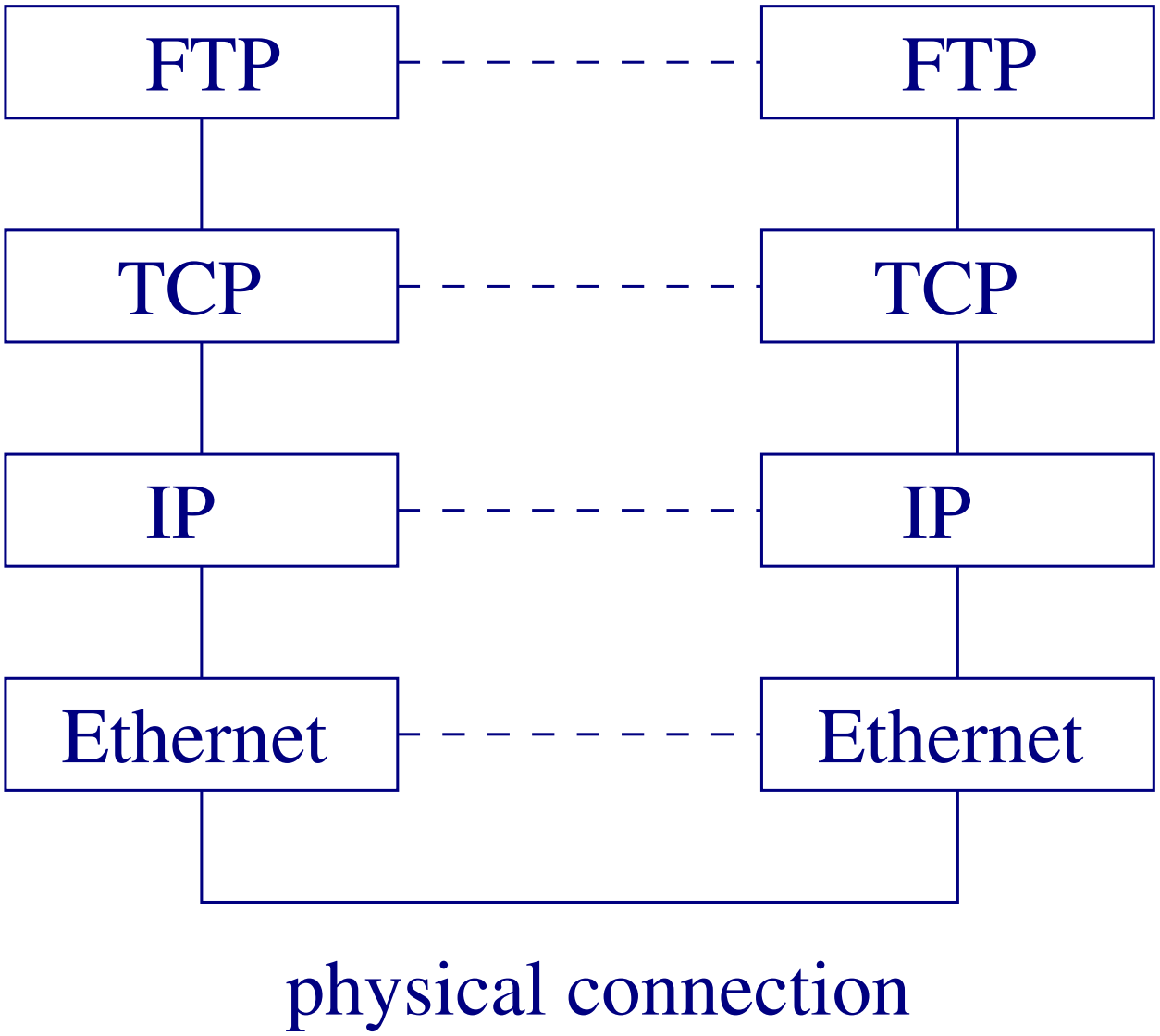
Layers: Problem

- System has different levels of abstraction
- Typical: Requests go down, notification goes back up
- It is possible to define stable interfaces between layers
- Want to be able to change or add layers over time

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)



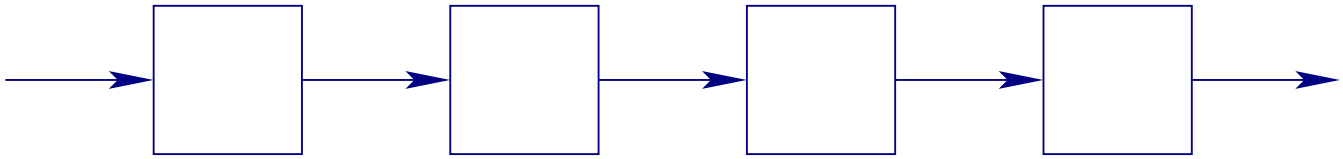
Layers: Advantages

- Reuse of layers
- Standardization of tasks and interfaces
- Only local dependencies between layers
- Programmers and users can ignore other layers
- Different programming teams can handle each layer

Layers: Liabilities

- Cascades of changing behavior
- Lower efficiency of data transfer
- Difficult to choose granularity of layers
- Difficult to understand entire system

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.

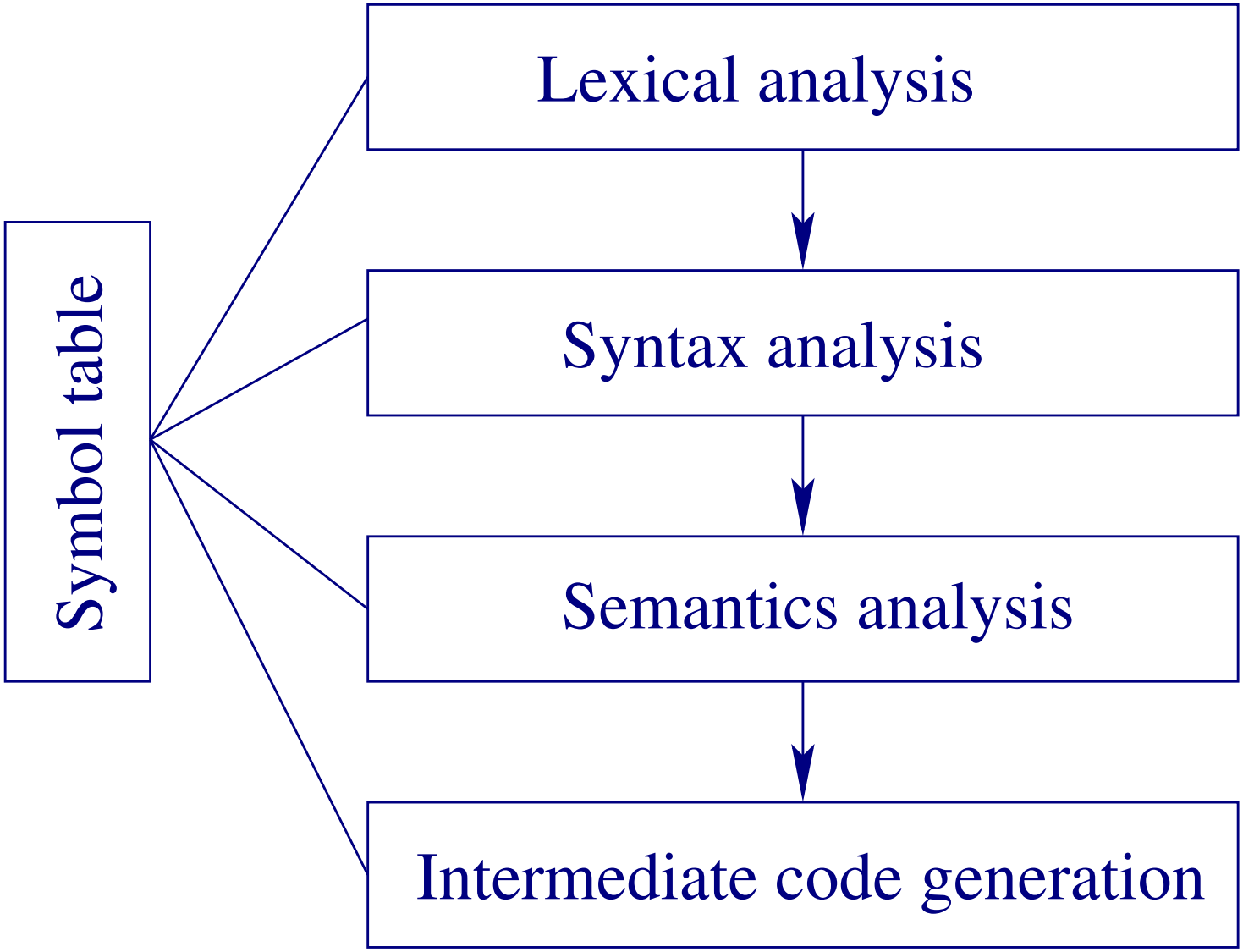
Pipes and Filters: Problem

- Data stream processing which naturally subdivides into stages
- May want to recombine stages
- Non-adjacent stages don’t share information
- May desire different stages to be on different processors
- Helpful: Standardized data structure between stages

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 (increases long-term usefulness)
- If data stream has a standard format, filters can be developed independently
- Incremental filters allow parallelization

Pipes and Filters: Liabilities

- Difficult to share global data
- Parallelization is less useful than may seem
- 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.

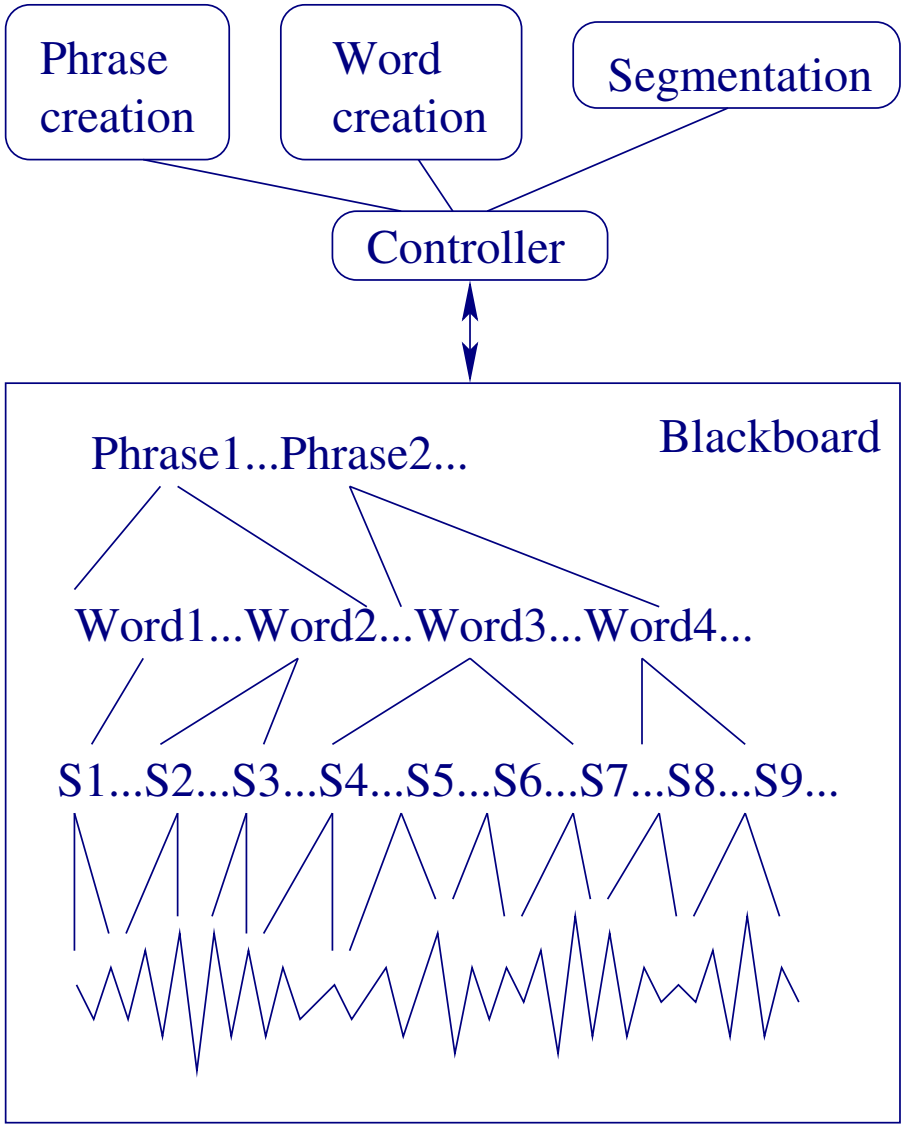
Blackboard: Problem

- Immature or poorly specified domain
- No deterministic or optimal 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 contains original input and/or partial solutions

Blackboard: Example



Blackboard: Advantages

- Allows problem solvers to be developed independently
- Easy (within limits) to experiment with different problem solvers and control heuristics
- System may (within limits) be tolerant to broken problem solvers, which result in incorrect partial solutions

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 synchronization

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

CORBA: Common Object Request Broker Architecture (e.g. JADE)

OLE/DCOM/Active X

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 .

Individual agents may be implemented in any language as long as they can input/output according to the protocol.

Broker: Advantages

- Components can be developed independently
- Location transparency
- Components easily adapted
- Broker easily adapted

Broker: Liabilities

- Low fault tolerance
- Limited efficiency (high communications cost)
- 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.

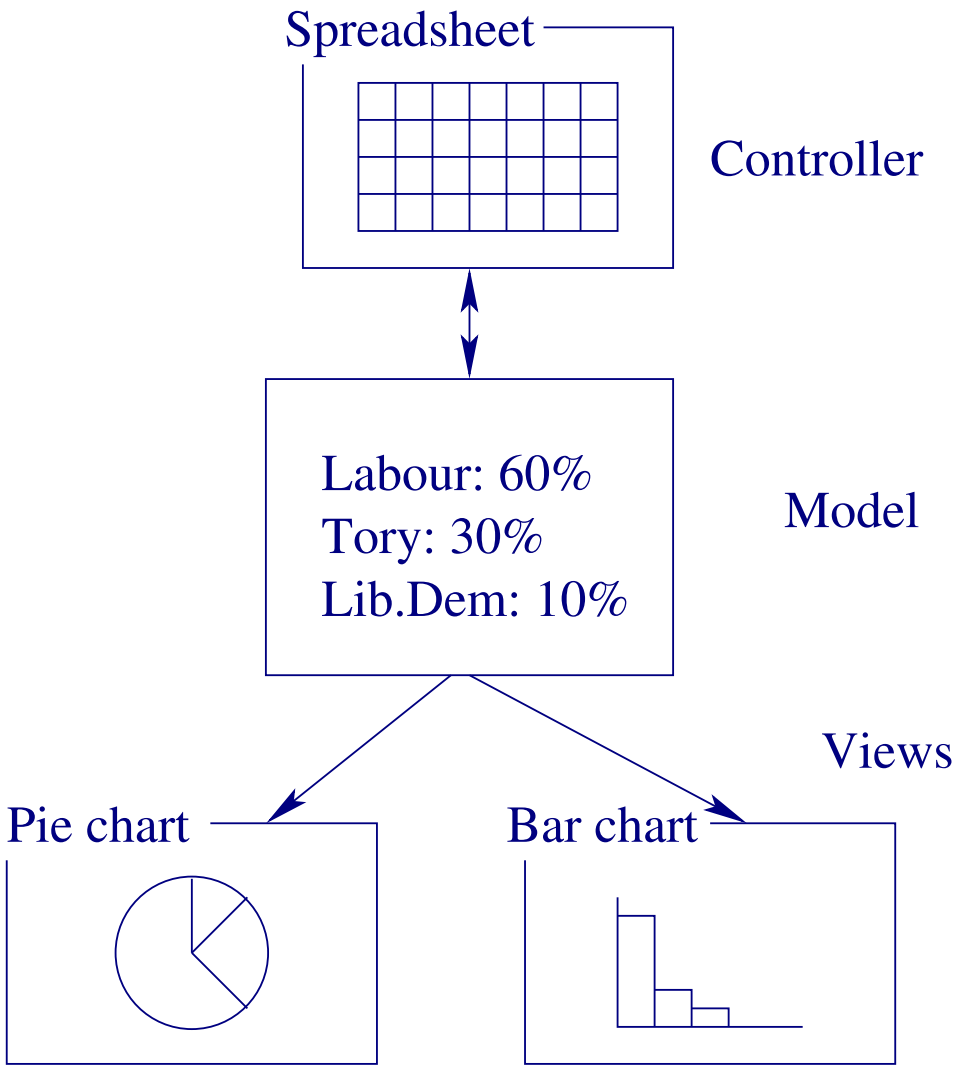
M-V-C: Problem

- User interfaces are prone to change requests over time
- Different users ask for different changes
- User interface technologies change rapidly
- You may want to support different “look and feel” standards
- Important core code can be separated from the 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 synchronization
- 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

A system implemented as a hierarchy of cooperating agents. Each agent is responsible for a specific aspect of functionality, and consists of:

- A presentation component responsible for its visible behaviour
- An abstraction component which maintains the data model for the agent
- 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, often developed independently
- Some agents specialize 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, to coordinate the other agents and (possibly) 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 that can be maintained separately
- 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

- Software systems with long life spans must evolve as 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
- Thus 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
  - 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
- Thus external servers or clients can be maintained independently of kernel
- Can extend 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

Summary

- Architectural patterns allow systems to be broken into chunks that can be developed (to some degree) and maintained independently
- These patterns support large-scale, long-term development and maintenance
- Not a recipe, just an approach

References

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Hoboken, NJ: Wiley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.