Architectural Patterns

Software Architecture, Process, and Management

In the previous lecture we looked at design patterns. These were defined as moulds for solutions to commonly occurring problems which are just outwith the size/complexity of problem we can formally construct a reusable solution to. In this lecture we will look at still larger patterns which cover the entire application.

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
- Architectural patterns are related to design patterns, but higher level and larger scale, and generally more **abstract**

Architectural Patterns

- The long-term goal is often to build reusable components
- The short-term goal is to allow components to be built without knowing everything about all the other components, because the software is large enough that **nobody** can know everything about all components
- This is true even for software developed by one person, they may at one time have known everything about all components, but never simultaneously

Buildings

- Recall that one of the main difficulties in authoring software projects is the fact that each software project is new
- Buildings, cannot be efficiently copied, so when a new one is required, it is possible to build something similar to one you have built before
- Software is inherently shareable, so if you had already built the system you are about to build, you would not need to build it again
- Architectural patterns are an attempt to identify aspects of software projects which are *similar* in many otherwise different projects
- Sometimes architectural patterns are turned into *frameworks* which **may** help with re-usability

Architectural Pattern Examples

- Architectural patterns covered here:
  - High level decompositions:
    - Layered Abstractions
    - Pipes and filters
    - Blackboard
  - Interactive systems:
    - Model-view-controller
    - Presentation-abstraction-control
  - Adaptable/reusable systems:
    - Microkernel
- There are of course others, eg. Broker, SOA

Layered Abstraction:

Problem Solution Advantages Liabilities

- System has different levels of abstraction
- Typical: Requests go down, notification goes back up
- Generally:
  - a lower-level is simpler to implement
  - a higher-level is simpler to use and/or more powerful
- It is possible to define stable interfaces between layers
- Want to be able to change or add layers over time

Layered Abstraction:

Problem Solution Advantages Liabilities

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

Layered Abstraction:

Problem Solution Advantages Liabilities

- Reuse of layers
- Standardisation of tasks and interfaces
- Only local dependencies between layers
- Programmers and users can ignore other layers
- Different programming teams can handle each layer
- The choice of layer can say something about what is not appropriate
  - For example if you choose UDP then there should be a reason that you are not using TCP
  - This can help as the basis for a document recording design decisions

Layered Abstraction:

Problem Solution Advantages Liabilities

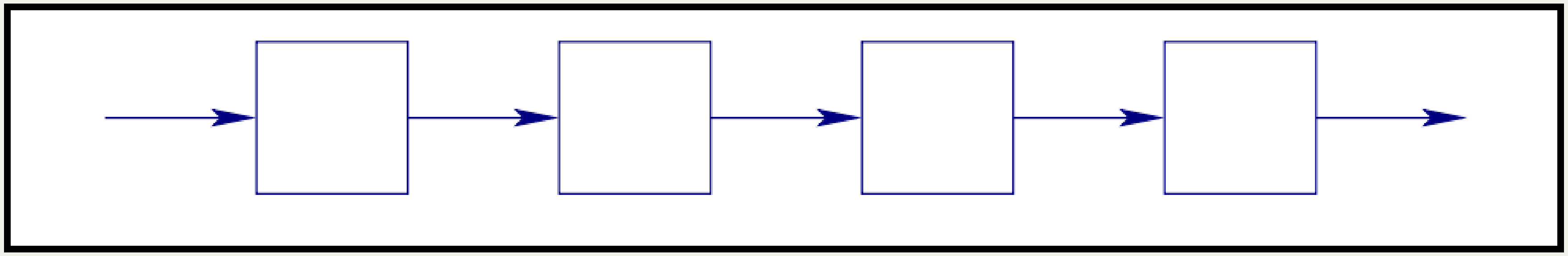
- Cascades of changing behaviour
  - Hence it can be difficult to fix behaviour at a lower-level without breaking something which depends upon the broken behaviour
- Lower efficiency due to:
  - A higher-level being unable to fully utilise a lower-level
  - Translations which occur at each level

Layered Abstraction:

Problem Solution Advantages Liabilities

- Difficult to choose granularity of layers
- Difficult to understand entire system
- Attempts at improvement at one level fail due to a misconception of an underlying level
  - Most commonly, optimisations at a higher-level fail because the higher-level operations ignore lower-level details
  - Cache behaviour is a notorious example of this

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
- Often related to the layered approach in that each filter is reducing the data to a lower level

Pipes and Filters

Problem Solution Advantages Liabilities

- Data stream processing which naturally subdivides into stages
- May want to recombine stages
- Non-adjacent stages do not share information
- May desire different stages to be on different processors
  - In this way the pipeline can work just as a pipeline in a CPU or an assembly line in a factory
  - There can be as many segments of data being simultaneously processed as there are stages in the pipeline
  - Throughput is thusly increased
- Helpful: Standardised data structure between stages

Pipes and Filters

Problem Solution Advantages Liabilities

- 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

Problem Solution Advantages Liabilities

- Can replace filters easily
- Can achieve different effects by recombination
  - thus hopefully increasing long-term usefulness
  - at least of some components
- If data stream has a standard format, filters can be developed independently
- They need not be developed in the same language

Pipes and Filters

Problem Solution Advantages Liabilities

- Difficult to share global data
- 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
  - these may communicate only with the blackboard
  - reading the current partial solution **and/or**
  - suggesting modifications to it via a control component

Blackboard

Problem Solution Advantages Liabilities

- Immature or poorly specified domain
  - Over which you may have little or no control
- 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

Problem Solution Advantages Liabilities

- 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

Problem Solution Advantages Liabilities

- 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

Problem Solution Advantages 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

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 Solution Advantages Liabilities

- Users care a lot about interfaces and demand changes often
- Different users ask for different changes
- The underlying GUI technologies also change rapidly
- You may want to support different *"look and feel"* standards
    - Most obviously the size of the screen can have a huge impact on the way users wish to view the model
    - The same interface on a desktop is rarely appropriate for a smart-phone
- You typically have important core code that can be separated from the interfaces

M-V-C

Problem Solution Advantages Liabilities

- Develop a core model which is independent of the style of input/output
- Define different views needed for parts of/the 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

Problem Solution Advantages Liabilities

- Multiple views of the same model
- View synchronisation
- View components can be plugged in
- Changes to interfaces without touching the model
- The controller and the model become reusable outside of the context of the interface
  - For example they can be scripted
  - A compiler separated from the IDE can be scheduled to run for nightly builds
  - Scientific software can run experiments not thought of by the interface designer

M-V-C

Problem Solution Advantages Liabilities

- Too complex for simple interface problems
- Frequent events may strain simple change propagation mechanisms
- Views and controllers are sometimes not modular in practice
- Changing the model is expensive if there are many views
  - Though when there are as many views as to make this an issue it is difficult to see how it could have been otherwise done

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
- Similar to M-V-C, the difference is there are many m-v-c-like triples, which communicate with each other only through the control component of each triple

P-A-C

Problem Solution Advantages Liabilities

- Interactive system viewed as a set of cooperating agents, often developed independently
- 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
- Many web applications are labelled as Model-View-Controller but are in fact closer to Presentation-Abstraction-Control

P-A-C

Problem Solution Advantages Liabilities

- 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

- A typical web application has:
  - The user interaction agent consisting of:
    - The abstraction in the form of the HTML
    - The presentation, in the form of CSS
    - The controller which is the web browser
  - The web service to which the user agent connects:
    - The abstraction in the form of the database/model
    - The presentation in the form of the service API or available web addresses
    - The controller which services requests to modify/pull from the model
  - May have other agents which deal with, for example scheduled jobs, data integrity checks etc.

P-A-C

Problem Solution Advantages Liabilities

- Separation of different concepts as individual agents that can be maintained separately
- Changes to presentation or abstraction in an agent does not affect other agents
- Easy to integrate/replace agents
- Suits multi-tasking
- Suits multi-user applications

P-A-C

Problem Solution Advantages Liabilities

- Can be difficult to get control right while maintaining independence of agents
- Long chains of communication may be inefficient
- Can have too many, too simple bottom-level agents
  - Essentially almost any architecture could be **labelled** as an instance of P-A-C with lots of tiny agents

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 Solution Advantages Liabilities

- 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

Problem Solution Advantages Liabilities

- 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

Problem Solution Advantages Liabilities

- Porting to a new environment normally does not need changes to external servers or clients
- Thus external servers or clients can be maintained independently of the kernel
- Can extend by adding new external servers
- Can distribute the microkernel to several machines, increasing availability and fault tolerance

Microkernel

Problem Solution Advantages Liabilities

- Performance overhead/bottleneck in that communication between components must be done via the microkernel
- May be difficult to predict which basic mechanisms should be in the microkernel

Enforcing the Architecture

- The architectural pattern does not explicitly state how one might enforce the separation specified in the pattern
- For example, how does one ensure that layer$_N$ does not have a direct dependency on layer$_{N-2}$?
- This may be done by physical separation, program separation, type-system based separation, or simply convention and discipline
- Breaking the separation enforced by the pattern may lead to new patterns

SOA, Mashups, Cloud computing

- Trends: further decentralisation, web-based distributed systems:
  - Service-oriented architecture: Build application from disparate, loosely coupled set of independently provided services, typically with web interfaces
  - Mashups: Rough, lightweight SOA, focusing on ability of end-users to reconfigure services on-the-fly to solve a specific problem or appeal to a particular audience
  - Cloud computing: Use externally provided, geographically scattered, highly decoupled services for storage and computation (esp. via virtual machines), rather than purchasing and relying on specific physical infrastructure

Combining Architectural Patterns

- As I have hinted, it is quite common that two or more architectural patterns are combined
  - Or two or more instances of the same architectural pattern
- This can be done globally, but more usually hierarchically
- In this way the component that forms a part of one architectural pattern is itself designed to follow an architectural pattern
- For example the controller in a model-view-controller may consist of sub-components arranged as a sequence of pipes and filters
- Though one should be careful not to *"over design"* their system
- Combining patterns may lead to new patterns

Architectural vs Design Patterns

- So what are the differences between architectural and design patterns?
- Design patterns are **template** solutions
  - often something we would wish to formalise in a language feature but are as yet unable or unwilling to do so
- A major attractive feature of design patterns is the vocabulary they enhance to further aid communication between developers

Architectural vs Design Patterns

- Architectural patterns, are about building **reusable/substitutable components**
  - Resuable components are good
  - Reusable and substitutable components generally have the additional benefit that they can be independently developed
  - Just as the legs of a table can be independently manufactured
- The constraints imposed by architectural patterns force developers into more cohesive, less coupled components which in turn are likely to be more reusable
- By detailing past successful project structures, architectural patterns guide us in structuring new projects