Software Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. [BCK03]

Software Architecture [2]

- A software architecture is a high-level design of the overall system structure.
- Why focus on software architecture?
  - New systems can be built as variations of old systems
    - Learn from successes!
    - Use it to make principled design decisions
  - Selection of software architecture can make/break the project
    - Software needs to be stable, reliable, modifiable, etc
  - Architectural system representation is often essential for analysis of high-level properties of a complex system [GS94]

System Descriptions

SAs are part of our vocabulary for describing systems:

- “Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and services.”
- “We have chosen a distributed, object oriented approach to managing information”
- “The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors…” [GS94]

A few architectural styles

- Pipe and filter
- Object oriented:
  - Client-Server; Object Broker
- Event based
  - Layered
    - Designing Layered Architectures
  - Repositories:
    - Blackboard, MVC
  - Process control

SA Notation

- SAs can be described visually using a graph
  - Nodes: components
    - E.g., computational, memory, ...
  - Edges: connectors (show interactions between components)
    - E.g., procedure calls, data flow, implicit invocation,...
Pipe-and-filter

- Examples:
  - UNIX shell commands
  - Compilers:
    - Lexical Analysis -> parsing -> semantic analysis -> code generation
- Interesting properties:
  - Filters don’t need to know anything about what they are connected to.
  - Filters can be implemented in parallel.
  - Behaviour of the system is the composition of behaviour of the filters.
  - Specialized analysis such as throughput and deadlock analysis is possible.

Object Oriented Architectures

- Interesting properties:
  - Data hiding (internal data representations are not visible to clients).
  - Can decompose problems into sets of interacting agents.
  - Can be multi-threaded or single thread.
- Disadvantages:
  - Objects must know the identity of objects they wish to interact with.

Object Oriented Architectures: Variant 1 - Client Server

- Interesting properties:
  - Special case of object oriented architecture.
  - Clients do not need to know about one another.
- Disadvantages:
  - Client objects must know the identity of the server.

Object Oriented Architectures: Variant 2 - Object Brokers

- Interesting properties:
  - Adds a broker between the clients and servers.
  - Clients no longer need to know which server they are using.
  - Can have many brokers, many servers.
- Disadvantages:
  - Broker can become a bottleneck.
  - Degraded performance.

Broker Architecture Example

- Examples:
  - Debugging systems (listen for particular breakpoints).
  - Graphical user interfaces.
- Interesting properties:
  - Announcers of events don’t need to know who will handle the event.
  - Supports re-use, and evolution of systems (add new agents easily).
- Disadvantages:
  - Components have no control over ordering of computations.

Event based (implicit invocation)

- Examples:
  - Graphical user interfaces.
- Interesting properties:
  - Announcers of events don’t need to know who will handle the event.
- Disadvantages:
  - Components have no control over ordering of computations.
Layered Systems

- **Examples**
  - Operating Systems
  - Communication protocols
- **Interesting properties**
  - Support increasing levels of abstraction during design
  - Support enhancement (add functionality) and re-use
  - Can define standard layer interfaces
- **Disadvantages**
  - May not be able to identify (clean) layers

Layered Systems: Variant - 3-layer data access

- Presentation layer
- Application Logic layer
- Storage layer

Open vs. Closed Layered Architecture

- **Closed architecture**
  - each layer only uses services of the layer immediately below;
  - Minimizes dependencies between layers and reduces the impact of a change.
- **Open architecture**
  - a layer can use services from any lower layer.
  - More compact code, as the services of lower layers can be accessed directly
  - Breaks the encapsulation of layers, so increase dependencies between layers

How many layers?

- 2-layers:
  - application layer
  - database layer
  - e.g. simple client-server model
- 3-layers:
  - separate out the business logic
  - helps to make both user interface and database layers modifiable
- 4-layers:
  - Separates applications from the domain entities that they use:
    - Boundary classes in presentation layer
    - Control classes in application layer
    - Entity classes in domain layer
  - Partitioned 4-layers
    - identify separate applications

Moving to Design

- Representing our architectures
  - UML elements can be grouped together in packages
  - Elements of a UML package may be:
    - other packages (representing subsystems or modules);
    - classes;
    - models (e.g. use case models, interaction diagrams, statechart diagrams, etc)
  - Each element of a UML model is owned by a single package
  - Packages need not correspond to elements of the analysis or the design
    - they are a convenient way of grouping other elements together

Moving to Design [2]

- Criteria for decomposing a system into packages:
  - **Ownership**
    - who is responsible for working on which diagrams
  - **Application**
    - each problem has its own obvious partitions
  - **Clusters of classes with strong cohesion**
    - e.g., course, course description, instructor, student,…
  - **Or use an architectural pattern to help find a suitable decomposition**
UML Package Diagrams

- Dependencies:
  - Similar to compilation dependencies
  - Captures a high-level view of coupling between packages:
    - If you change a class in one package, you may have to change something in packages that depend on it
  - A good architecture minimizes dependencies
    - Fewer dependencies means lower coupling
    - Dependency cycles are especially undesirable

Package Diagram Example

- UI
  - Swing
  - Web
- Domain
  - Sales

Package Diagram: Layers

- UI
  - Swing
    - not the Java Swing libraries, but our GUI classes based on Swing
  - Web
- Domain
  - Sales
  - Payments
  - Taxes
- Technical Services
  - Persistence
  - Logging
  - RulesEngine

Dependency Cycles

- UML Package Diagrams
  - Client
  - Sub-system A
- Sub-system B
- Server
- Sub-system C
- Peers
- Sub-system D

The server sub-system does not depend on the client sub-system and is not affected by changes to the client’s interface.

Each peer sub-system depends on the other and each is affected by changes in the other’s interface.

References

