Patterns, Frameworks, and Architectures

Design Reuse

- Reuse of architectural and design experience is one of the most important factors in successfully building large software systems.
- Software engineers follow guidelines and good examples of working designs and architectures that help to make successful decisions.
- Instead of being based on a sufficiently strong theory and "calculating" software designs, software engineers (like other engineers) combine a little theory with a lot of experience.

Forms of Design-Level Reuse

- Reusing proven designs is essential, but there is no single reuse approach that covers all levels of granularity that software engineers have to face.
- Some of the established reuse techniques include:
  - Sharing concrete solution fragments: *libraries*.
  - Sharing contracts: *interfaces*.
  - Sharing interaction fragments: *message formats/protocols*.
  - Sharing interaction architectures: *patterns*.
  - Sharing subsystem architectures: *frameworks*.
  - Sharing overall structure: *system architectures*.

Interfaces and Contracts

- Interfaces and contracts allow the providers of a service and the clients of that service to be developed independently.
- The *interface* defines the syntax of communication between the client and server.
  - E.g. method names, parameter types, etc.
- A *contract* defines the semantics of the communication. It may be defined using:
  - Documentation written in natural language prose.
  - Pre-conditions and post-conditions of operations.
  - Unit test cases.
- Some people use the term interface more broadly, to include the contract.

Interfaces and Contracts (2)

- Attaching contracts, formal and informal ones, to a named interface leaves implementers of providers and clients with verification obligations.
  - Where a contract has been carefully crafted, providers and clients from fully independent sources will be able to interact properly.
- Interfaces with their associated contracts are logically the smallest contractual unit in a system.
- An interface and its contracts say nothing about the larger organizational structures – the architecture – of a system.

Messages and Protocols

- While interfaces reside at the endpoints of interactions, message schemas reside "on the wire" – that is, on the logical line between interacting parties.
- A message schema describes a set of valid messages, usually with no constraints on particular senders or receivers.
- Message schemas can be described at the same two levels that characterize interfaces – a syntactic level that constrains the format of messages and a semantic level that captures the contractual nature of a message.
- Protocols define the valid sequences of messages that can be exchanged between endpoints.

Messages and Protocols (2)

- A message can be made self-describing by tagging the message with the name of the message schema.
  - e.g. XML DTD (document type definition)
- Interfaces and messages are closely related.
  - At one extreme, all interfaces could be reduced to a single operation accepts a single self-describing message and returns another self-describing message. This is largely the style of web protocols such as HTTP.
  - At the other extreme, all interfaces have fully factored operations that only take and return simple atomic values. This is largely the style of traditional application programming interfaces (APIs).
  - There are many design points between these two extremes.
- Messages and protocols more naturally describe asynchronous communication, while interfaces and operations more naturally describe synchronous communication.

Patterns

- Patterns originated with the building architect Christopher Alexander and his colleagues during the 1960's and 1970's when they identified the concept of patterns for capturing architectural decisions.
- Patterns document the design commonality that exists in different successful applications.
- They are micro-architectures that describe the abstract interaction between objects collaborating to solve a particular problem.
- A pattern documents a recurring problem–solution pairing within a given context.
  - It includes both the problem and the solution, along with the rationale that binds them together.
  - A proposed solution is described in terms of its structure, and includes a clear presentation of the consequences—both benefits and liabilities—of applying the solution.

Patterns (2)

- An attempt to collect and catalog systematically the smallest recurring architectures in object-oriented software systems led to the cataloging of design patterns in 1995 by the "Gang of Four" (GoF) – Gamma, Helm, Johnson, and Vlissides.
- Since then, many more patterns have been documented.
- It is sometimes possible to identify families of patterns that fit together in a harmonious way.
- Adding heuristics that guide the combination of patterns within such a family then forms a language – a pattern language.

Frameworks

- An application framework is an integrated set of components that collaborate to provide a reusable software architecture for a family of related applications.
- In an object-oriented environment, an application framework consists of abstract and concrete classes, and *inversion of control*.
  - Unlike libraries, control flow is dictated by the framework, not the user (caller) of the framework.
- Instantiation of such a framework consists of composing and subclassing from existing classes.

Frameworks (2)

- Abstract classes require implementation inheritance for the framework to work at all.
- Other classes provide default implementations.
  - Instead of defining everything, a client merely needs to augment or replace those defaults that do not fit.
- An important role of a framework is its regulation of the interactions that the parts of the framework can engage in.
  - By freezing certain design decisions in the framework, critical interoperation aspects can be fixed.
  - A framework can thus significantly speed the creation of specific solutions out of the semi-finished design provided by the framework.

Patterns versus Frameworks

- Frameworks are less abstract and more specialized than design patterns. Frameworks are partial implementations of subsystems, while patterns have no immediate implementation at all; only examples of patterns can be found in implementations.
- Design patterns are smaller architectural elements than frameworks. Indeed, some patterns live on the granularity of individual methods – examples are the *Template Method* and the *Factory Method* patterns.
- Most frameworks apply multiple patterns in their design.

System Architecture

- Patterns are micro-architectures; frameworks are subsystem architectures. Most larger, fully functional systems will introduce multiple frameworks.
- Large distributed systems tend to be complex. In the beginning, all we have is a set of requirements and constraints that must be transformed into a working software system.
- A naive approach to development is likely to result in a big ball of mud, a software clump whose design and code is so messy that it is hard to see any coherent architecture in it.
- Such software is hard to understand, maintain, and evolve, and over time it also tends to suffer from poor stability, performance, scalability, and other essential operational qualities.

System Architecture (2)

- One of the keys to successful software development is structure. We need structure that supports the needs of developers and other stakeholders:
  - can be understood by developers
  - is resilient to the forces of change
  - favors the development process
  - respects the business and individuals who will shape it
- Without vision, the structure of a software system is likely to be overly complicated, leading not only to the loss of the big picture, but the small picture as well—the code can become mired in accidental detail and assumptions.
- In undertaking large scale software development, a coarse-grained conception of the system is needed that omits unnecessary details and organizes the system's key concepts at a broader level.