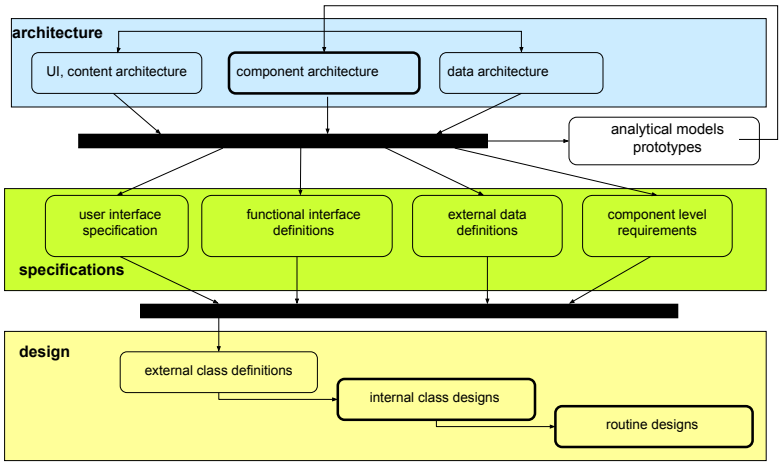
Design Patterns

- many different types of patterns
- architectural patterns
  - pipes, repository, event, client-server, proxy
- modularity class patterns
  - iterator, visitor, bridge, strategy, observer
- synchronization
  - events, locks, monitors, leases
- creational models
  - singleton, object pool, factory

Model Hierarchy/Succession



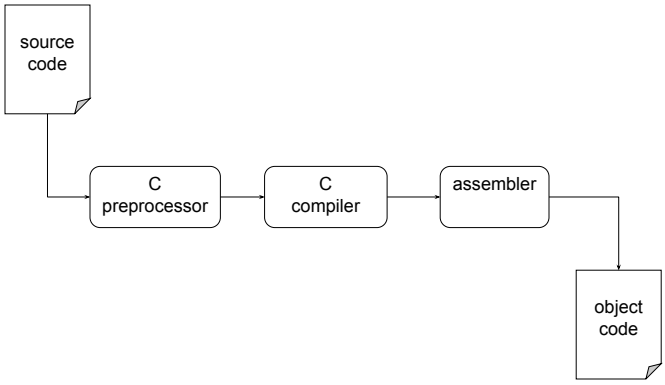
Types of Patterns

- Architectural patterns
  - functional relationships between components
- Class patterns
  - solving common problems in an OO-fashion
  - information hiding with complex interactions
- Behavioral patterns
  - algorithms and processing techniques
  - resource creation/management techniques
  - communication and synchronization techniques

Architecture: pipes and filters

- Application
  - stream data processing and transformation
- Approach
  - pipeline of generators, filters, transformers
- Advantages
  - independence of the processing elements
  - power of composing independent elements
  - reusability/replaceability of elements
- Limitations
  - no feedback between stages

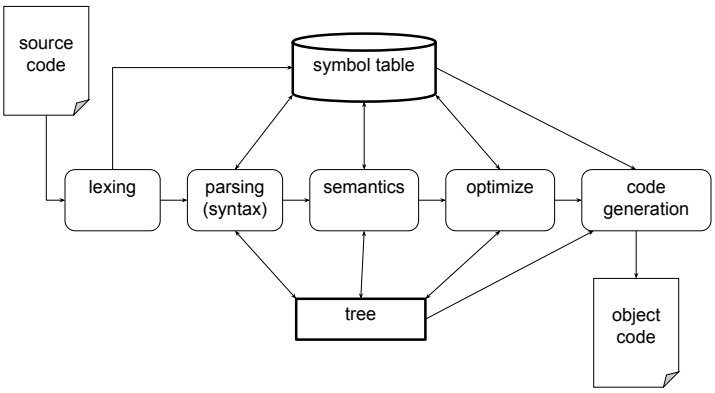
(over-simplified) Compiler Pipeline



Architecture: central repository

- Application:
  - continuing series of distinct operations
  - against a progressively evolving state
- Approach:
  - all operations operate on central database
- Advantages:
  - small, modular operations (php, applets)
  - ease of testing and debugging
- Limitations:
  - scalability, corruptibility

(more realistic) Compiler Pipeline



Architecture: events

- Application
  - cooperating asynchronous parallel processes
  - where one operation triggers others
- Approaches
  - direct: *Listener* registration
  - anonymous: pub/sub distribution network
- Advantages
  - flexibility and independence of participants
- Limitations
  - scalability, throughput

Architecture: client-server

- Application
  - unique, large, or expensive resource
  - to be shared by many (and changing) clients
- Approach
  - resource is owned by a single server
  - clients use it by sending request messages
- Advantages
  - flexibility, generality, heterogeneity
- Limitations (complexities)
  - discovery, new protocols, extra components

Architecture: proxies

- Application
  - local s/w interacts with remote resources
  - interactions are frequent (i.e. expensive)
- Approach
  - create a local simulation of remote resource
  - with intelligently managed communication
- Advantages
  - local proxy can be smart/efficient
- Limitations (complexity)
  - cost/complexity of remote interactions
  - awkward modes of failure

Modularity: singletons

- Goal: single-instance, semi-global object
  - one master, global/shared information
- Approach: a class with only one instance
  - static getInstance method, private constructor
- Advantages:
  - shared data advantages of globals
  - modularity advantages of a class
- Costs: (no free lunches)
  - synchronization, susceptibility to corruption

Singletons - example

```
public class Singleton {

  private Singleton() {
     …
  }

  static private Singleton _instance = null;

  static public Singleton getInstance() {
     if (_instance == null)
        _instance = new Singleton();
     return _instance;
  }
}
```

Modularity: iterator

- Goal: enumerate all elements in a class
  - obtain references to each element of a set
  - without understanding aggregation structure
- Approach: abstract iterator interface
  - e.g. getFirst(), current(), next(), prev()
  - implementation encapsulates aggregation
- Issues
  - effect of aggregation changes on iteration
  - ensuring they are multi-thread safe

Modularity: visitor

- Goal: enumerate more complex structures
  - walk a composite heterogeneous object
  - performing a wide range of operations
  - without understanding aggregation structure
- Approach: walker with per-node call-back
  - decouple operations from aggregation
  - allow one walker to serve many purposes
- Issues (complexity)
  - visitor must support all possible node types
  - visits are per-element, no structural info

Modularity: bridge/strategy

- Goal: decouple clients & implementations
  - clients don't depend on implementing class
  - permit interchangeable implementations
  - client & implementation evolve separately
- Approach:
  - client depends only on an interface
  - strategy object, references an implementation
  - client unaware of implementing class
- Limitations:
  - all strategies must take same client inputs

Modularity: observer

- Goal: server-to-client notifications
  - where server has no knowledge of clients
- Approach: implement a call-back interface
  - server implements register-callback method
  - clients implement callback methods
  - clients call server.register(callback)
  - server distributes events by calling callback()
- Issues:
  - knowing which server generated this event
  - system state must be up-to-date before call

Architectural/Class Exercise

- consider processing in your project
  - identify non-trivial control/information flows
  - consider patterns that might speak to it
- sketch the solution with this approach
  - how well did this solve your problem?
  - how easy was it to apply?
  - does this change your architectural thoughts?
- be prepared to discuss your results
- do you have needs no pattern addresses?

Behavior: locks

- Goal: resource/critical-section serialization
- Approach: enforced lock/unlock operations
  - associate a lock object with each resource
  - obtain lock prior to entering critical section
  - release lock after completing critical section
- Issues:
  - granularity: bottlenecks and convoys
  - circular dependency: deadlocks
  - errors: locks obtained but never released

Behavior: leases

- Goal: failure-robust serialization
- Approach: limited-time leases
  - client obtains lease, uses it as access token
  - lease can be returned just like a lock
  - if lease expires, access token is revoked, resource can be reallocated
- Issues:
  - fencing out holders of expired leases
  - restoring reclaimed resource to like-new state

Behavior: atomic transactions

- Goal: robust multi-step operations
- Approach: all-or-none transactions
  - create a transaction object
  - submit operations against that transaction
  - **commit** (apply) or **abort** (roll-back) entire batch
- Benefits:
  - general and powerful abstraction for client use
- Issues:
  - passes atomicity problem down to lower layer

Behavior: object pools

- Goal: reduce cost of object instantiation
- Approach:
  - create an object pool class (w/acquire,release)
  - recycle a few objects many times
- Advantages:
  - huge performance gains
- Complexities:
  - handling allocation failures
  - serialization and cleanup

Behavior: Dynamic Equilibrium

- Goal: adapt resource allocation to load
- Approach: resource stealing
  - separate pools for different clients
  - if your pool is empty, steal space from another
    - but only if it is less stressed than your pool
  - allocation automatically adapts to changing load
- Issues:
  - assumes resources can be stolen
  - damping wasteful “marginal” stealing
  - doesn’t address total system overload

Design Patterns

- much like Chess openings or Go *joseki*
  - using them will improve your game
    - they address commonly recurring situations
    - they will give you new ideas and options
  - studying them will improve your understanding
    - as you come to appreciate how each one works
- but they are only useful paradigms
  - they can’t analyze the board for you
  - they are only tactics, you still need a strategy

Further Reading

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, 1995
- Mary Shaw, David Garlan, Software Architectures: Perspectives on an emerging discipline, Prentice-Hall, 1996

For the next lecture

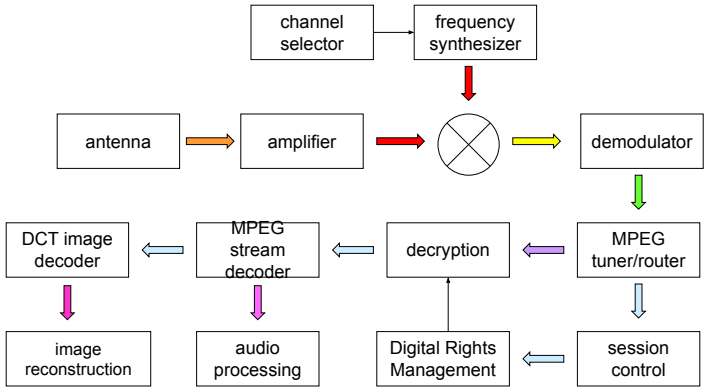
- McConnell ch 19.5-6
  - Code Complexity
- McConnell ch 22
  - Testing theory and practice
- Kampe: Introduction to S/W test cases
  - risk & testing, black/white-box testing, plans
- Cornett: Code coverage
  - types and relative advantages
- (review) Kampe: Testability

Supplementary Slides

For Next Lecture

- McConnell 30.2, source-code tools)
- McConnell 31, layout and style
- McConnell 32.1-4, commenting
- McConnell 34.3, understandability
- McConnell 34.5, conventions
- Wikipedia: Doxygen
  - documentation generation

Digital TV Data Flow Model



Architecture Pattern Exercise

- each team, choose an architecture
  - central repository, event, client-server, proxy
- identify a problem that begs for this
  - perhaps in your own project
- sketch a solution without this approach
  - reasonable solution, not a straw-man
- sketch the solution with this approach
- how is the resulting architecture better?
- be prepared to discuss your results

Class Pattern Exercise

- each team, choose a class pattern
  - iterator, visitor, bridge/strategy, observer
- identify a problem that begs for this
  - perhaps in your own project
- sketch a solution without this approach
  - reasonable solution, not a straw-man
- sketch the solution with this approach
- why is this a better solution?
- be prepared to discuss your results

For the next lecture

- McConnell ch 19.5-6
  - Code Complexity
- McConnell ch 22
  - Testing theory and practice
- Kampe: Introduction to S/W test cases
  - risk & testing, black/white-box testing, plans
- Cornett: Code coverage
  - types and relative advantages
- (review) Kampe: Testability