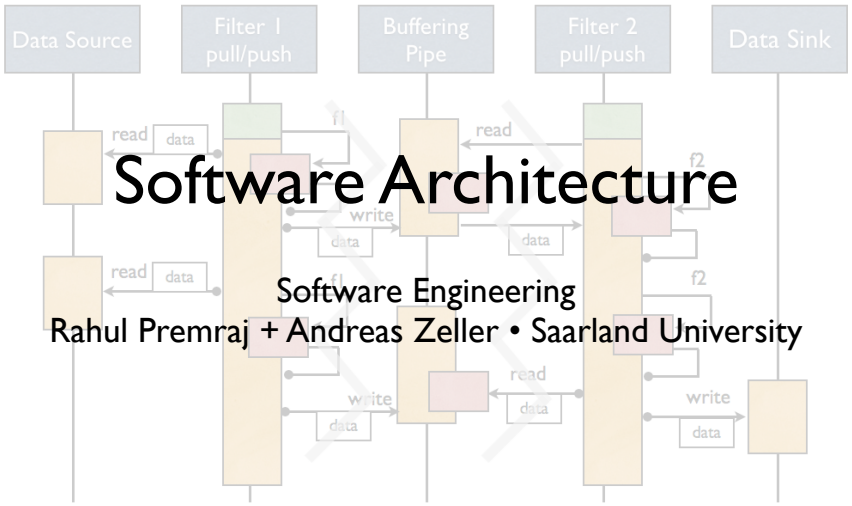
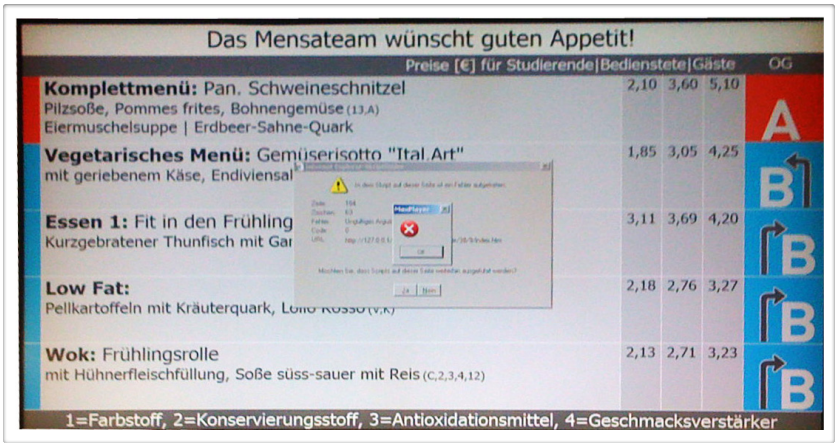


Software Architecture

Software Engineering

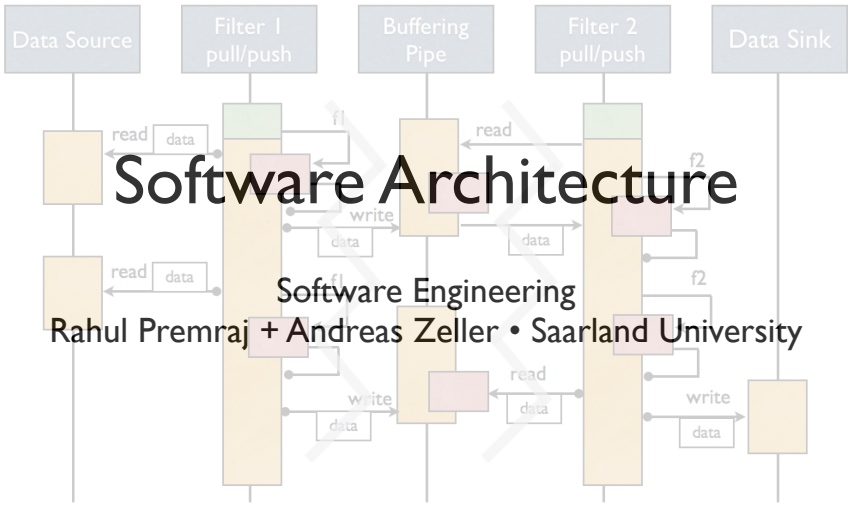
Rahul Premraj + Andreas Zeller • Saarland University

Bug of the Week



173 students. And counting.

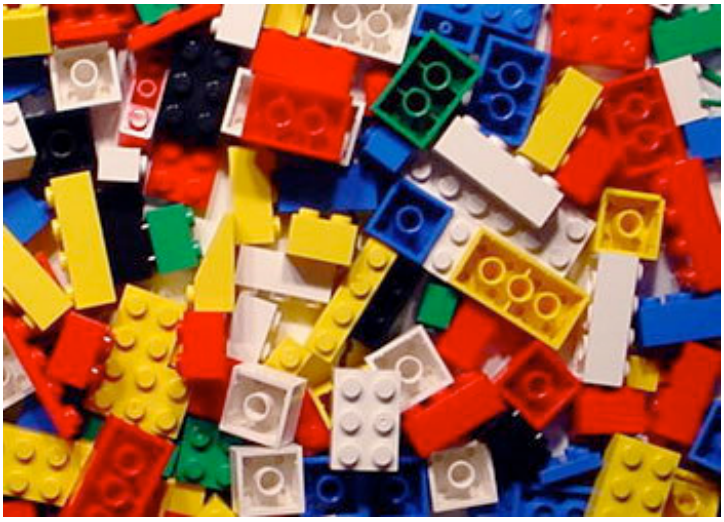
- Moved exam to **July 25, 0900–1200**
- Moved mini tests to **exercise groups**
- Final grade will be **100% exam**
- Practice mini tests **this week**
- Enjoy!



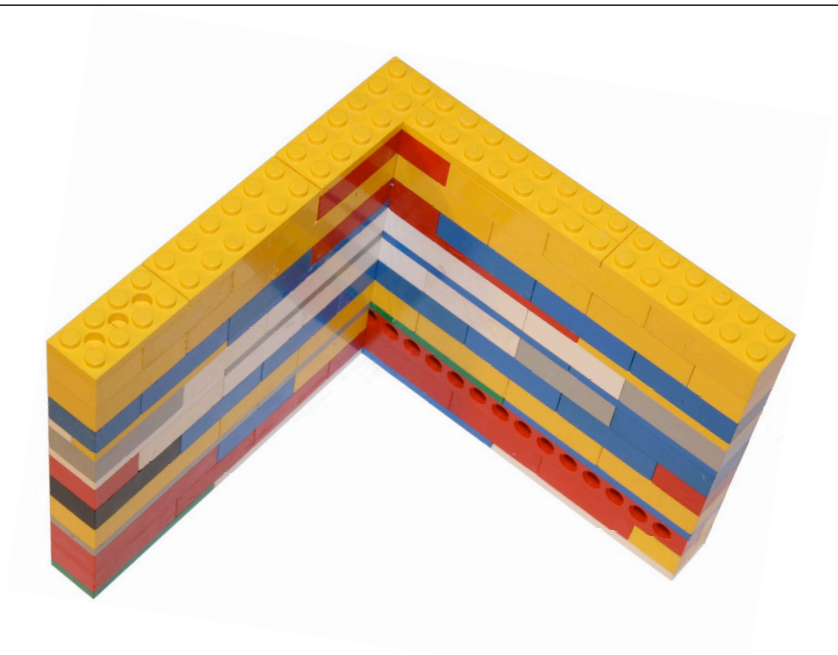
Software Architecture

Software Engineering

Rahul Premraj + Andreas Zeller • Saarland University



Software is composed out of many parts

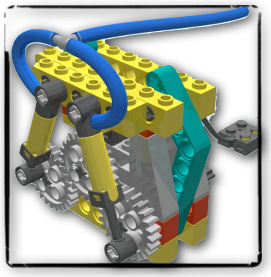


some of the tasks we do are simple...

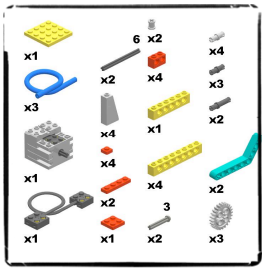
but then, it gets more complex...

7

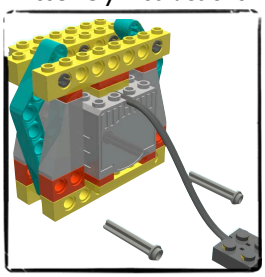
Final Model



Components

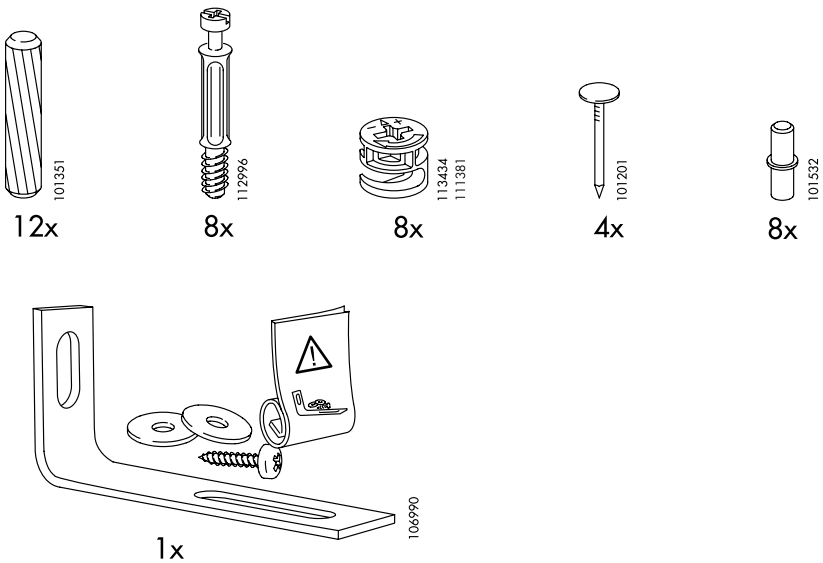


Assembly Instructions



you see the need to plan

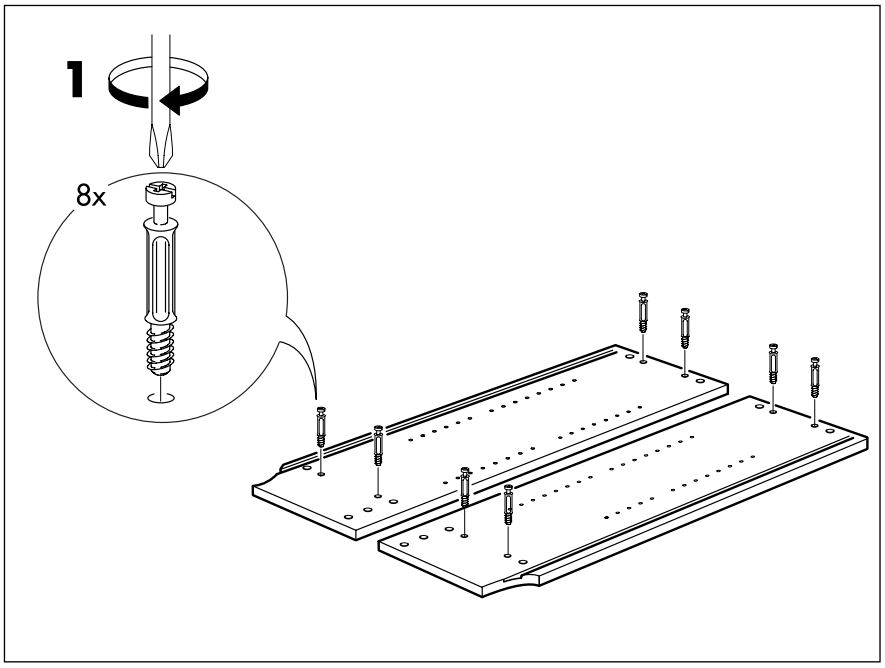
8



To build anything big or complex requires *planning*… same as in real life :–)

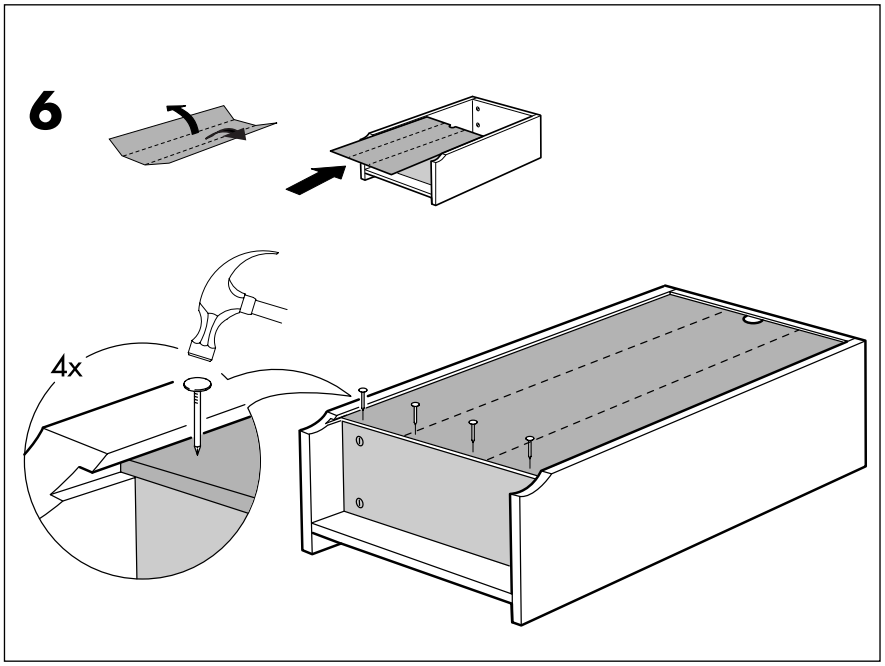
Source: IKEA

9



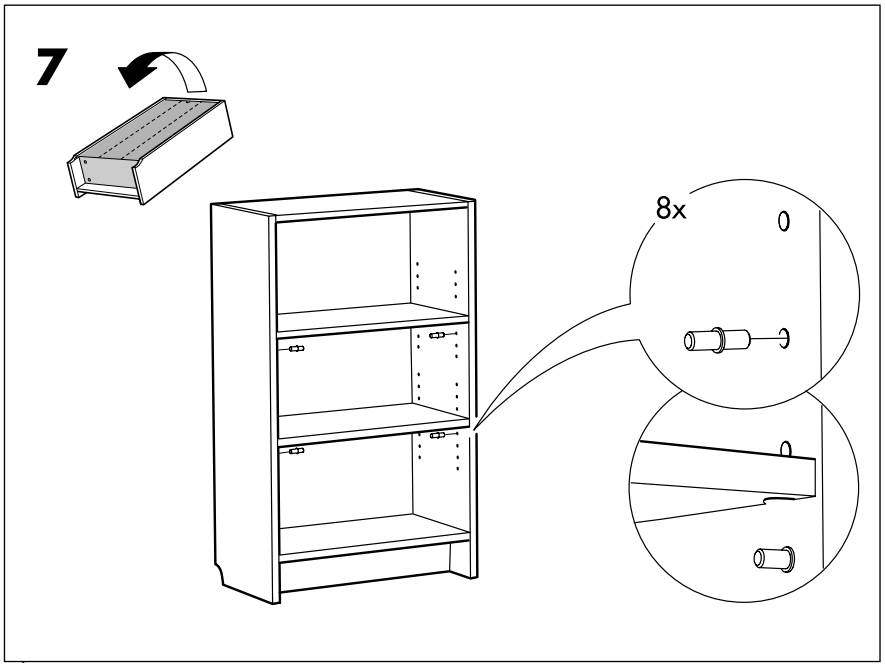
same as in real life :–)
Source: IKEA

10



same as in real life :–)
Source: IKEA

11



same as in real life :–)
Source: IKEA

12

Patterns

But there are general *patterns* that keep on occurring again and again

13

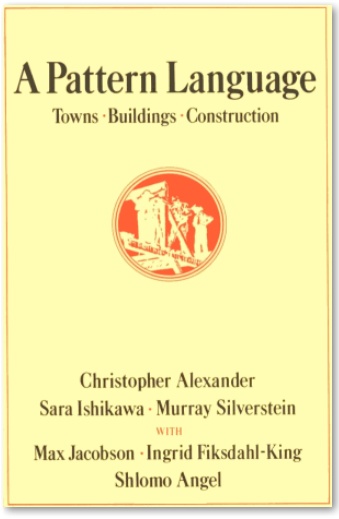
Patterns solve Problems

- The sloping roof-top helps rain water and snow to slide off the roof.
- Wood is easily available and helps contain the heat inside the house.
- Such solutions evolve as *patterns* over time

And these patterns are driven by *problems*

14

What is a Pattern?



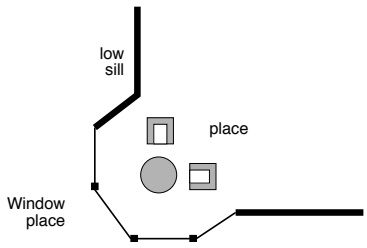
- Experts usually do not invent new solutions to problems. Instead, they reuse *existing*, *tried* and *tested* methods.
- Experts think in terms of *problem-solution pairs* (also, the basis of some artificial intelligence techniques).

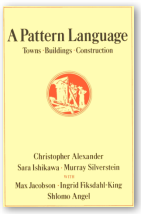
15

Window Place

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

In every room where you spend any length of time during the day, make at least one window into a "window place"



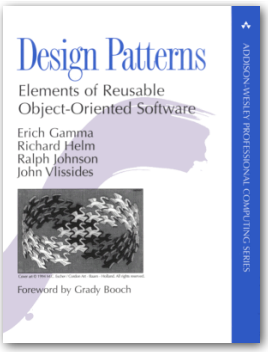


This is a pattern from architecture – stating the problem and its solution

16

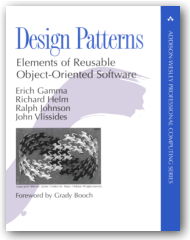
Design Patterns

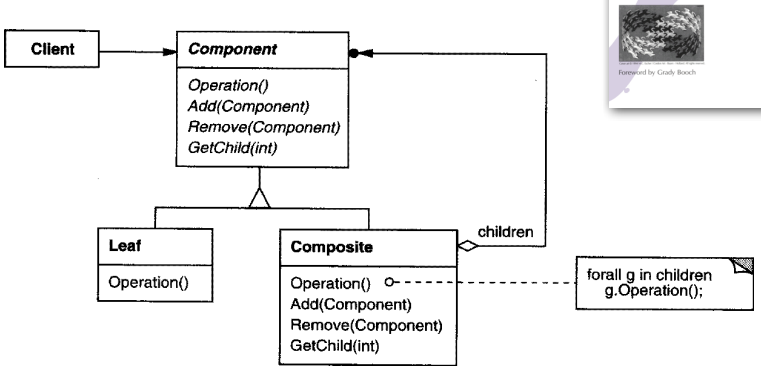
- *Composite*
- *Strategy*
- *Command*
- … and many more …



17

Composite





18

Caveats

- Patterns, their components and their relationships are not always 'atomic'
- They often need to be adapted to fit your problem
- Their application may raise new issues to be addressed
- Issues can be resolved by applying other patterns

Patterns

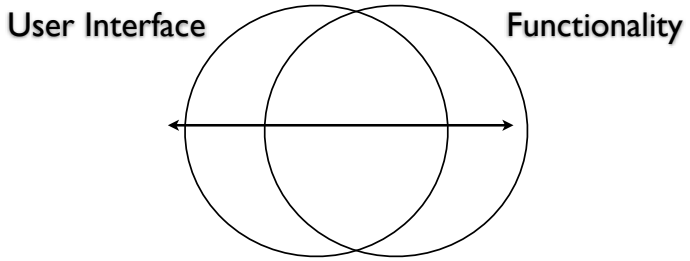
- address a *recurring design problem* that arise in specific situations
- document existing, well-proven *design experience*
- provide a common *vocabulary* and *understanding* for design principles
- help build heterogeneous and complex software
- encourage reuse!

2 – Not invented or artificially generated. Distilled, knowledge shared amongst many, tried and tested solutions offered, can apply solutions to design problems w/o having to rediscover. 4 – Patterns are concrete building blocks for construction

Architectural Patterns

both tightly interwoven – expensive and error-prone – complex to develop and maintain. Change for one type of user effects the entire system

Architectural Patterns



- Changes in the UI should be easy, and possible at runtime.
- Changing UI should not affect the code dealing with the functionality of the system.

both tightly interwoven – expensive and error-prone – complex to develop and maintain. Change for one type of user effects the entire system

Architectural Patterns

Model	View	Controller
encapsulates core data and functionality	displays information to the user	receives inputs, e.g. mouse or keyboard

This is an *architectural pattern*

Software Architecture

- A software system's architecture is the set of *principal design decisions* about the system
  where "principal" is determined by system goals
- Software architecture is the blueprint for a software system's *construction* and *evolution*
  at any point in time, a system has only one architecture
- Design decisions encompass *every facet* of the system under development
  Structure • Behavior • Interaction • Non-functional properties

From Richard Taylor, "Advances in Software Architecture", Salerno 2008

Architectural Pattern

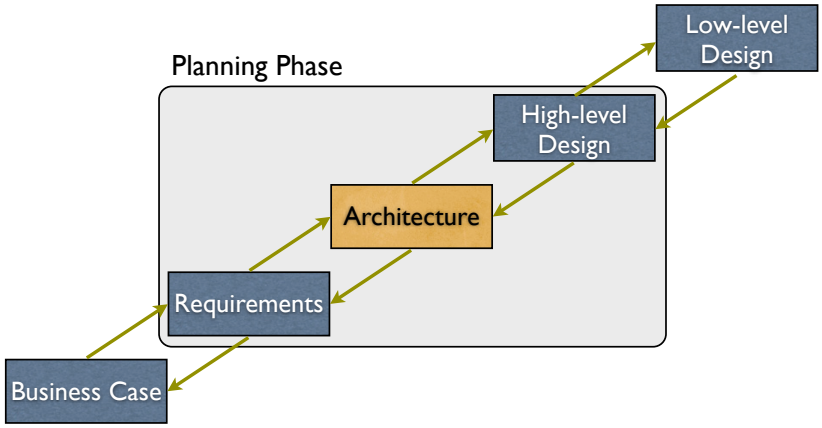
- An *architectural pattern* is a set of *architectural design decisions* that are
  - applicable to a recurring design problem,
  - parameterized to account for different software development contexts in which that problem appears

From Richard Taylor, "Advances in Software Architecture", Salerno 2008

Pattern Categories

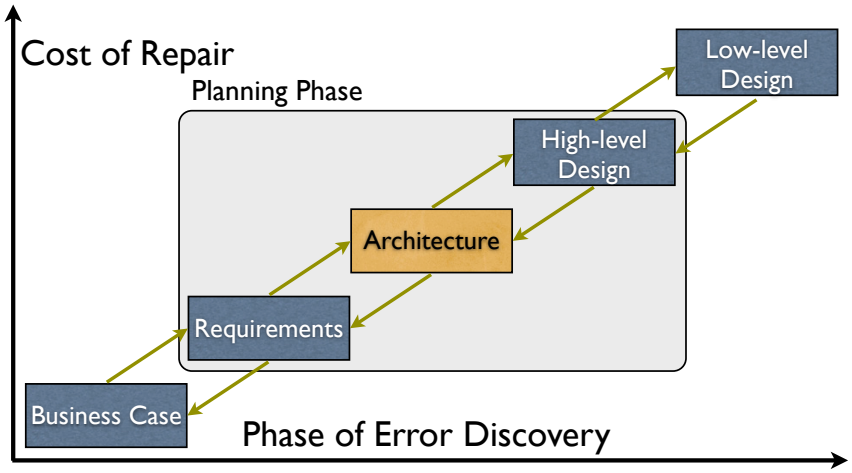
Architectural Patterns	provide fundamental structural organisation schema for software systems.
Design Patterns	provide schemes for refining the sub-systems or components of software systems, or relations between them
Idioms	low-level patterns specific to programming languages

Where does Architecture fit?



Credits: Kenneth M. Anderson

Good Architecture lowers Cost

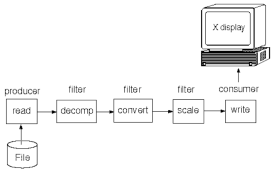


Credits: Kenneth M. Anderson

28

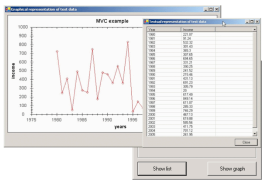
Layers

Pipes and Filters



Four Patterns

Model-View-Controller



Representational State Transfer

http://en.wikipedia.org/wiki/Representational_State_Transfer

29

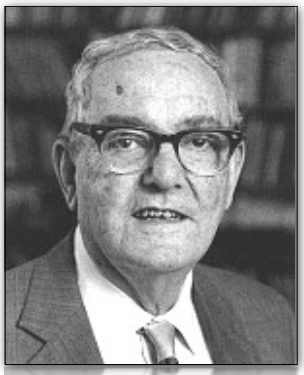
Layers

huge task – needs to be decomposed. There are high level and low level issues. There are several levels of abstraction.

30

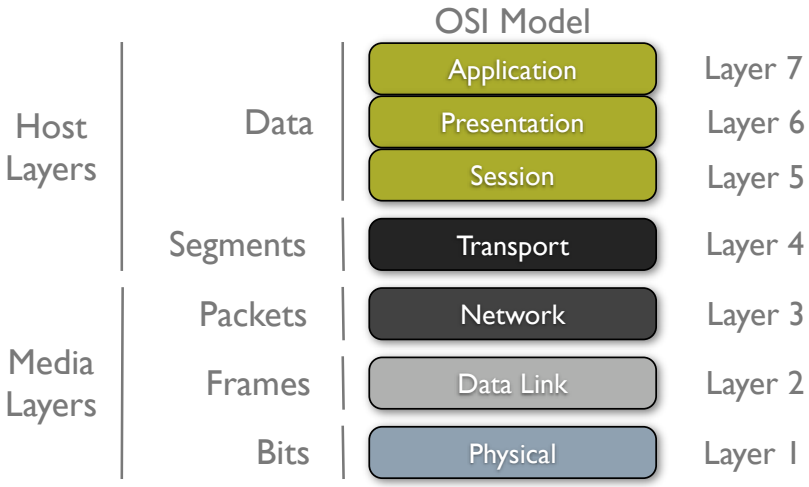
Simon's Law

Herb A. Simon
Nobel Prize in Economics, 1978



Hierarchical structures reduce complexity.

Layers



7 - Provides misc. protocols for common activities.
6 - Structures info. and attaches semantics.
5 - Provides dialog control and synchronisation facilities.
4 - Breaks messages into packets and guarantees delivery.
3 - Selects a route from sender to receiver.
2 - Detects and corrects errors in bit seqs.
1 - Transmits bits.

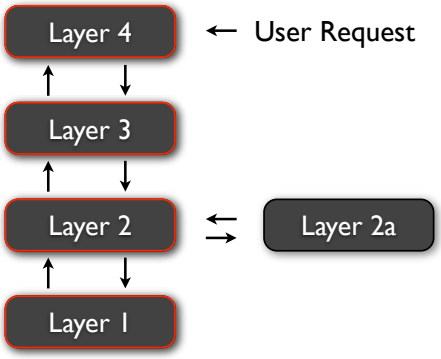
Layers

CRC Card

Class Layer J	Collaborators • Layer J - 1
Responsibility • Provides services used by Layer J + 1. • Delegates sub-tasks to Layer J - 1.	

Layers

Context: A large system that requires decomposition.

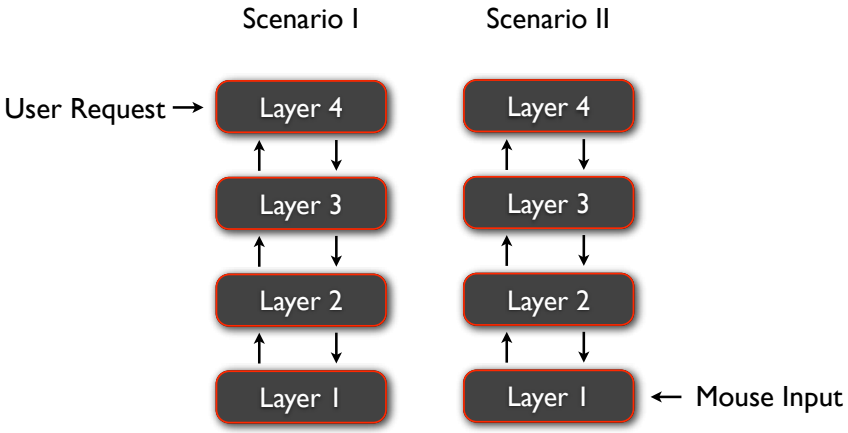


for example

Transport

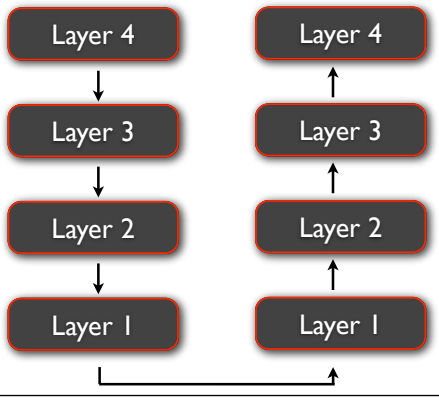
Breaks messages into packets and guarantees delivery

Layers

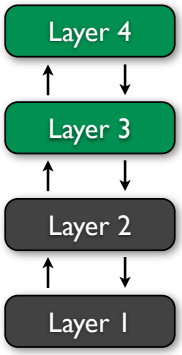


Layers

Scenario III



Layers

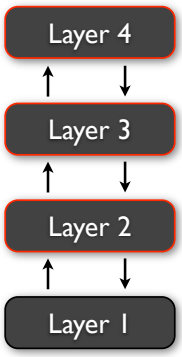


Benefits

- Layers can be reused
- Support for standardisation
- Dependencies are kept local
- Exchangeability

1 - well defined abstraction and documented interface, layer can be reused in multiple contexts.
1 - but many ppl prefer to rewrite the functionality.
2 - Clearly defined and commonly accepted levels of abs. Diff. implementations of the same interface... diff. vendors.
3 - Supports portability of the sys. Changes to the OS do not affect the data format.
4 - old imprementation with new one. Even ones with diff. interface using the Adaptor pattern.

Layers



Liabilities

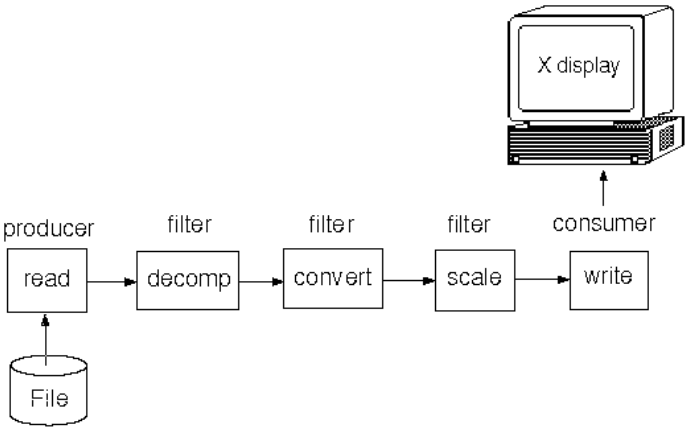
- Cascades of changing behavior
- Lower efficiency
- Unnecessary work
- Difficulty of establishing the correct granularity of layers

1 - Replace 10 Mbit/sec ethernet layer in Layer 1 and on top, 150 Mbit.
2 - If high level objects rely on lower level ones... then we have to go through everything. same reverse!
3 -excessive or duplicate work not required by higher layers.

Layers

huge task – needs to be decomposed. There are high level and low level issues. There are several levels of abstraction.

Pipes and Filters



- divides task into sequential processing steps
- connected by data flow, input --> output
- processing pipelines = seq. of filters connected by pipes.

Pipes and Filters

- Future enhancements should be possible by exchanging processing steps
- Small processing steps are easier to reuse
- Non-adjacent steps do not share information
- Explicit storage of intermediate files clutters and is error-prone
- Can we run steps in parallel?

Pipes and Filters

CRC Card

Class Filter	Collaborators • Pipe
Responsibility • Gets input data. • Performs function on its input data. • Supplies output data.	

– processing units –– enriches, refines, transforms the inputs.

Pipes and Filters

A filter's activity can be triggered by several events –

Passive Filters

- The subsequent pipeline element pulls output data from the filter.
- The previous pipeline element pushes input data to the filter.

Active Filters

- More commonly, the filter is an active loop pulling its input from and pushing its output down the pipeline.

Pipes and Filters

CRC Card

Class Pipe	Collaborators • Data Source • Data Sink • Filter
Responsibility • Transfers data. • Buffers data. • Synchronises active neighbours.	

Pipes and Filters

CRC Card

Class Data Source	Collaborators • Pipe
Responsibility • Delivers input to processing pipeline.	

– files, lines of text, sequence of numbers from sensor... etc.
– Can actively push or passively wait.

Pipes and Filters

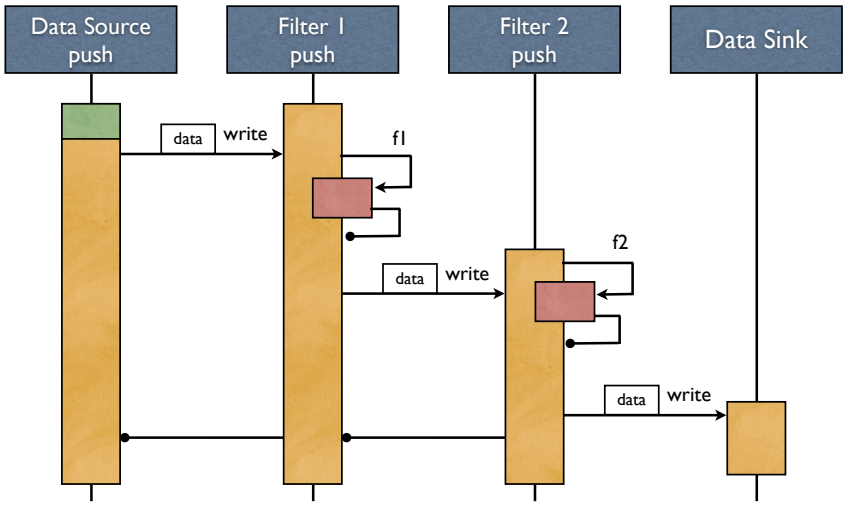
CRC Card

Class Data Sink	Collaborators • Pipe
Responsibility • Consumes output.	

– Can actively push or passively wait.
– Terminal, text file.

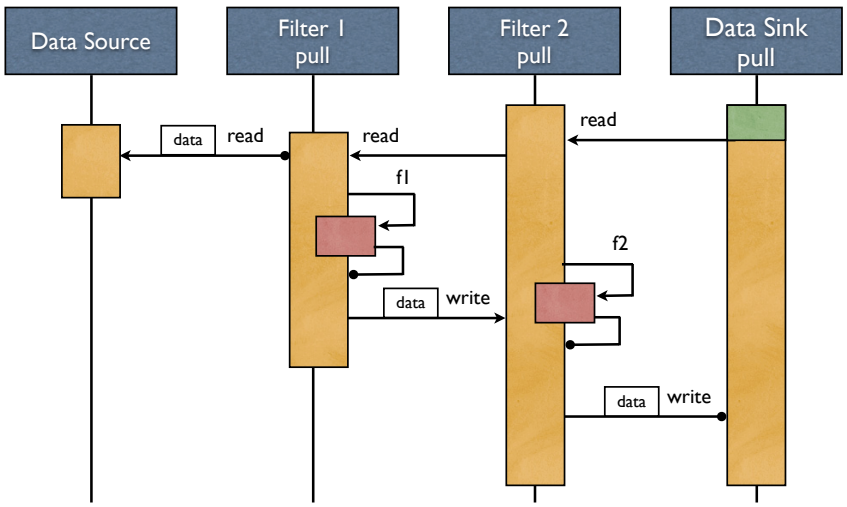
Pipes and Filters

Scenario I (push pipeline)



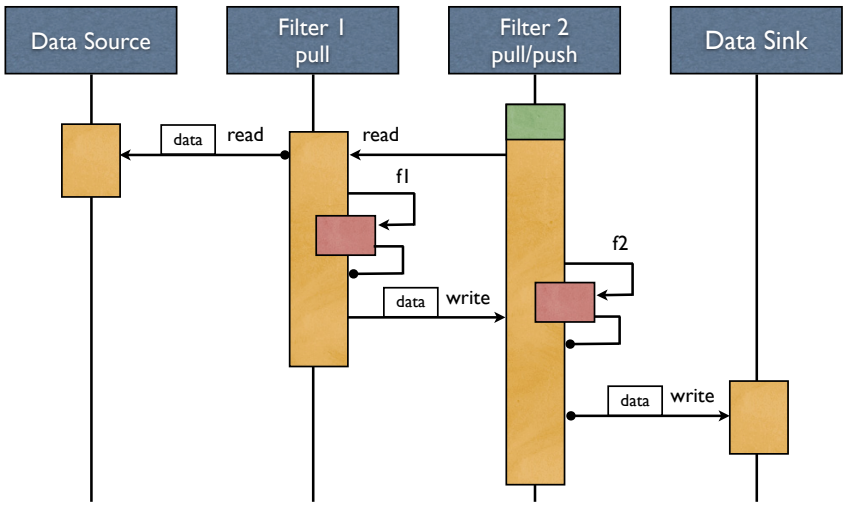
Pipes and Filters

Scenario II (pull pipeline)



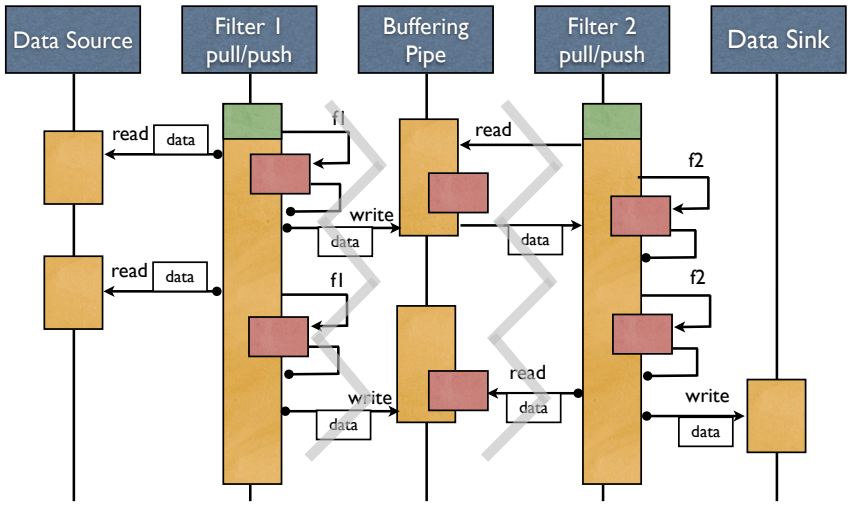
Pipes and Filters

Scenario III (mixed pull-push pipeline)



Pipes and Filters

Scenario IV



Filters are synchronised by buffering pipe. Pipe can hold only one value.

Pipes and Filters

Consequences

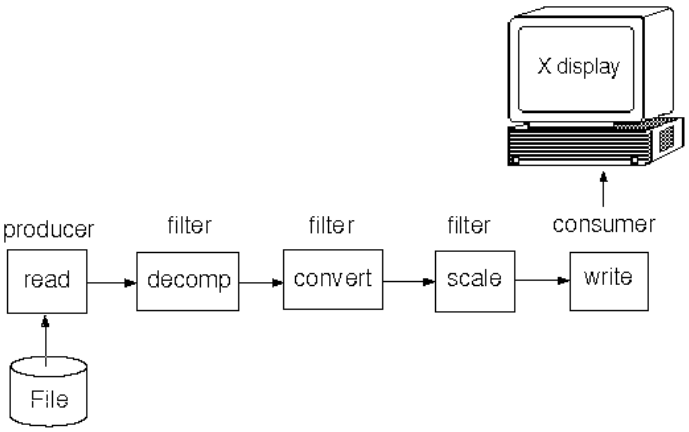
- No intermediate files necessary
- Flexibility by filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping of pipelines
- Efficiency by parallel processing

Pipes and Filters

Liabilities

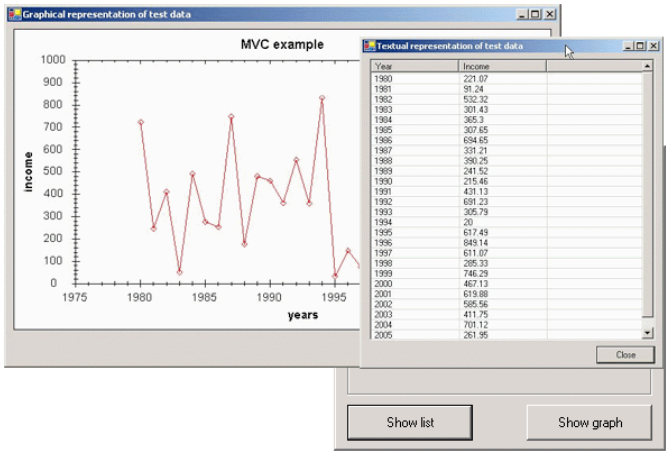
- Sharing state information is expensive or inflexible
- Efficiency gain by parallel processing is often an illusion
- Data transformation overhead
- Error handling

Pipes and Filters



- divides task into sequential processing steps
- connected by data flow, input --> output
- processing pipelines = seq. of filters connected by pipes.

Model-View-Controller



Model-View-Controller

Problems

- User interfaces are especially prone to change requests
- Different users place conflicting requirements on the user interface
- Interwoven interface and functionality is expensive and error-prone

- Same information on different windows
- Display should reflect change in data immediately
- Changes to UI should be simple, even in runtime
- 'look and feel' support should not affect functionality

Model-View-Controller

Model	View	Controller
encapsulates core data and functionality	displays information to the user	receives inputs, e.g. mouse or keyboard

Model-View-Controller

CRC Card

Class Model	Collaborators • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

Model-View-Controller

Change-Propagation Mechanism

- Model maintains a *registry of dependent components* (views and controllers).
- Changes to the model's state triggers the change-propagation mechanism.
- It is the only link between models and views and controllers.

Model-View-Controller

CRC Card

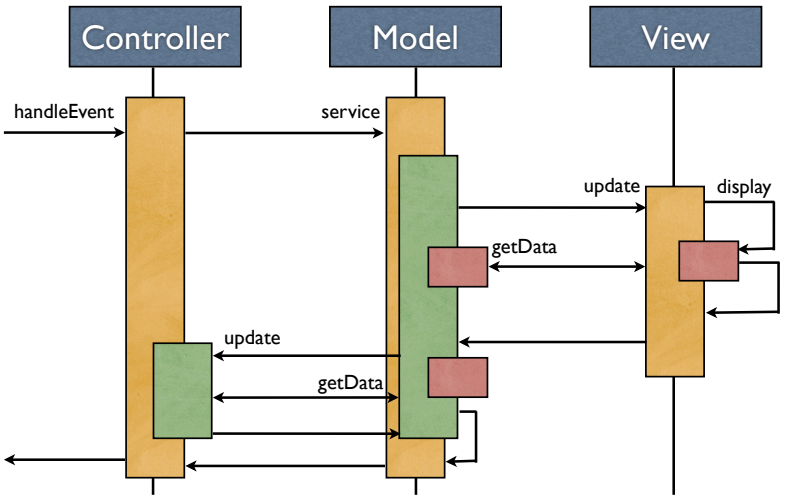
Class View	Collaborators • Controller • Model
Responsibility • Creates and initialises its controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Model-View-Controller

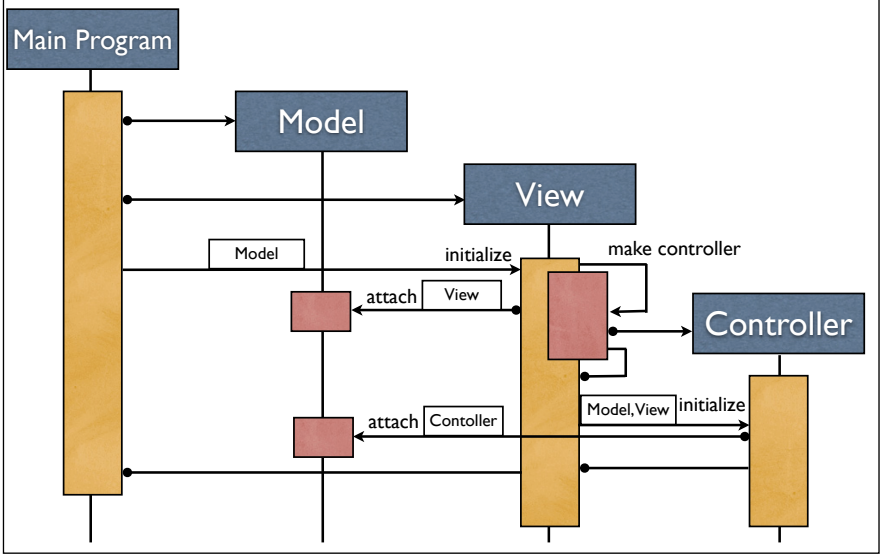
CRC Card

Class Controller	Collaborators • View • Model
Responsibility • Accepts user inputs as events. • Translates events to service requests or display requests. • Implements the update procedure.	

Model-View-Controller



Model-View-Controller



Model-View-Controller

Consequences

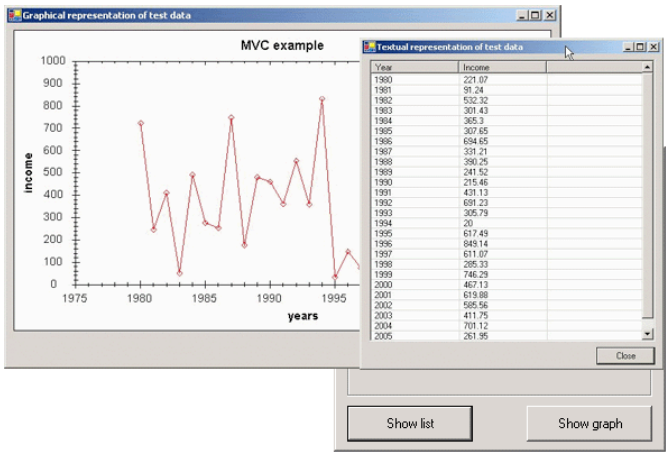
- Multiple views of the same model
- Synchronized views
- Pluggable views and controllers
- Exchangeability
- Framework potential

Model-View-Controller

Liabilities

- Increased complexity
- Potential for excessive number of updates
- Connection between view and controller
- Inefficiency of data access in View

Model-View-Controller



Representational State Transfer

http://en.wikipedia.org/wiki/Representational_State_Transfer

Representational State Transfer

Problems

- How do we address individual *resources* within a complex system?
- How do we manage the plethora of access methods (caches, proxies, servers…)?

Representational State Transfer

- Application state and functionality are abstracted into *resources*
- Every resource is uniquely addressable using a *universal syntax* for use in hypermedia links
- All resources share a *uniform interface* for the transfer of state between client and resource

Representational State Transfer

- A protocol which is
  - Client-server
  - Stateless
  - Cacheable
  - Layered

REST Example

The World Wide Web is a perfect example of REST design

Representational State Transfer

Consequences

- Supports *caching* of representations
- Improves server scalability by reducing the need to maintain session state
- A single browser can access any application and any resource
- Good long-term evolvability and scalability

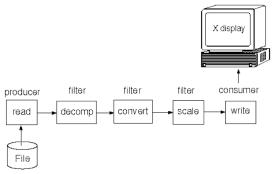
Proponents of REST argue that the Web's scalability and growth are a direct result of its key design principles.

Representational State Transfer

http://en.wikipedia.org/wiki/Representational_State_Transfer

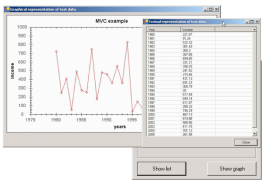
Layers

Pipes and Filters



Four Patterns

Model-View-Controller

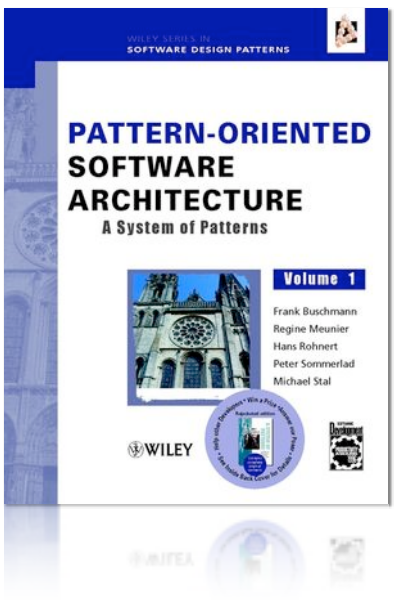


Representational State Transfer

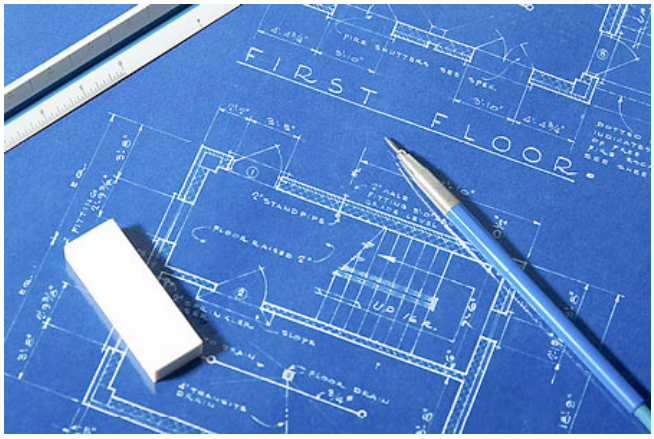
http://en.wikipedia.org/wiki/Representational_State_Transfer

More Patterns

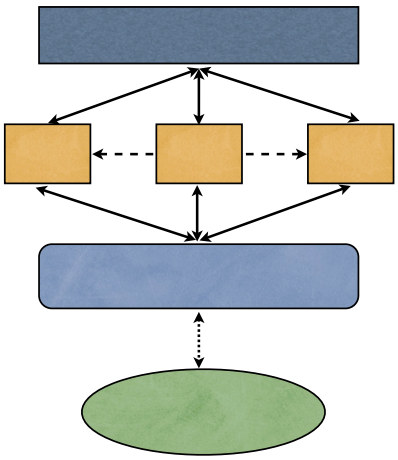
- Blackboard
- Client-Server
- and many others…
- More in next lecture!



Specifying Architectures



Specifying Architectures



Box/Circles and Lines

- Do the arrows represent data flow, control or other connection?
- What do ellipses represent– classes, methods, objects?
- What do the bi-directional arrows mean?
- How do we ensure this specification?

Architecture Languages

- ADL = *Architectural Description Language*
- Architectures are roughly described as a set of components connected by connectors.
- Several ADLs have been developed to meet different needs.
- ADLs specify well-defined syntax and some semantics – combined to form structures.

Architecture Languages

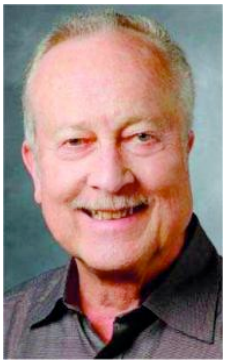
```
System C-S- Example
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-Connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-Connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end C-S-Example.
```

Figure 1: A Client-Server system in Wright

Advantages of ADL

- Some formal analysis can be performed, e.g., checking for consistency and completeness.
- The design can be unambiguously understood by different stake holders.
- Hopefully, transform a formal architectural description to programming language.

Rapide



David Luckham
Stanford University

- ADL that builds on partially ordered sets.
- Introduces new and powerful programming constructs.
- Is an ADL and an executable programming language.
- No. of tools built, e.g., static checking and for simulation.

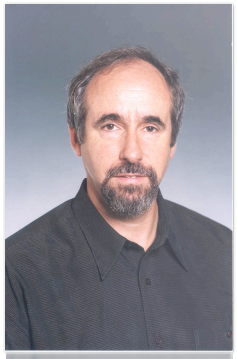
UniCon

Mary Shaw
Carnegie Mellon University

- Allows defining architectures in terms of abstractions.
- Designed to make "a smooth transition to code."
- Components and connectors can be of types built-in, or more complex types and user-defined types-code templates, code generators and informal guidelines.

Æsop

David Garlan
Carnegie Mellon University



- Concentrates on the problem of style reuse.
- One can define different styles and then use them when constructing an actual system.
- Provides a generic toolkit and communication architecture.
- Examples of tools integrated are cycle detectors, C-code generators, compilers, editors and structured language editors.

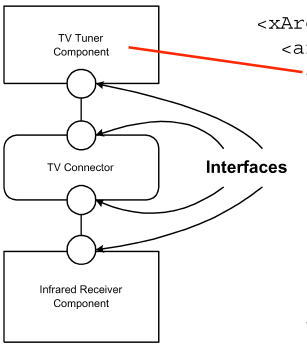
xADL

Richard Taylor
University of California, Irvine

- Concentrates on *extensibility*
- Leverages XML extension mechanisms
- Applied in industrial contexts, modeling software architectures for aircraft and spacecraft systems

ERIC M. DASHOFY, ANDRÉ VAN DER HOEK, and RICHARD N. TAYLOR
A Comprehensive Approach for the Development of Modular Software Architecture Description Languages
ACM Transactions on Software Engineering and

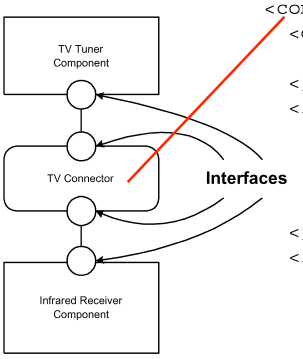
A TV System




```
<xArch>
  <archStructure>
    <component id="tuner">
      <description>
        TV Tuner Component
      </description>
      <interface id="tuner.channel">
        <description>
          ChangeChannel Interface (in)
        </description>
        <direction>in</direction>
      </interface>
    </component>
```

To see how these schemas can be used to model the core of a software architecture, consider a software system that might be used to drive a low-end television set. Such a device might have only two software components, one to interface with the TV tuner, and one to drive the infrared detector used to pick up signals from the remote control.
(ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 2, April 2005.)

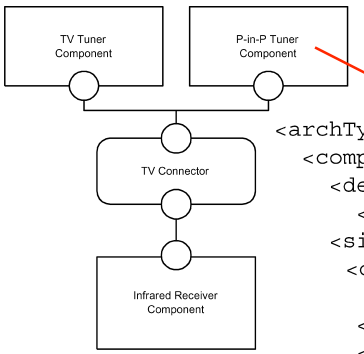
A TV System




```
<connector id="tvconn">
  <description>
    TV Connector
  </description>
  <interface id="tvconn.in">
    <description>
      ChangeChannel Interface (in)
    </description>
    <direction>in</direction>
  </interface>
  <interface id="tvconn.out">
    <description>
      ChangeChannel Interface (out)
    </description>
    <direction>out</direction>
  </interface>
</connector>
```

These two components are connected by a software connector that allows the infrared receiver component to send signals to the TV tuner to change the channel.
(ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 2, April 2005.)

Types




```
<archTypes>
  <componentType id="tuner_type">
    <description>TV Tuner Type
      </description>
    <signature id="tuner_type.channel">
      <description>ChangeChannel Sig (in)
        </description>
      <direction>in</direction>
      <type href="changechannel_type"/>
    </signature>
  </componentType>
```

Typing is an important construct in most ADLs. Each component, connector, or interface can optionally contain an XML link to a type; multiple elements can share a type. The type serves as a construct where common properties of elements. To demonstrate this, we will add types to our television example. We will also add another component to our television, a picture-in-picture tuner. Since the main tuner and the picture-in-picture tuner are basically identical, they will share a type.
(ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 2, April 2005.)

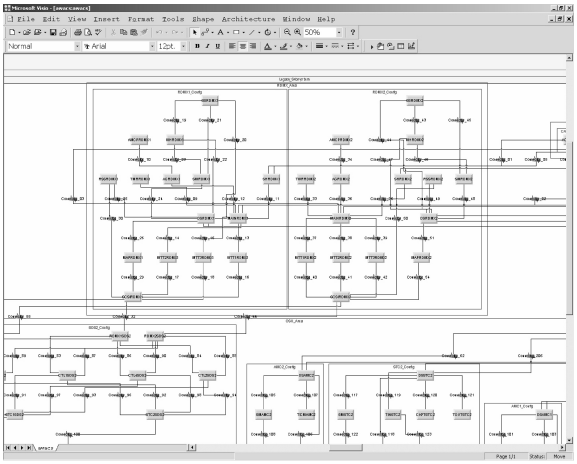
Implementation Details

```
<xArch>
  <archStructure>
    ...
  </archStructure>

  <archTypes>
    <componentType id="tuner_type">
      <description>TV Tuner Type</description>
      ...
      <implementation>
        <mainClass>
          <javaClassName>edu.uci.isr.tv.TelevisionTuner</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
          <initializationParameter>
            <name>tv_model</name>
            <value>Astro 5000</value>
          </initializationParameter>
        </mainClass>
        <auxClass>
          <javaClassName>edu.uci.isr.tv.TelevisionUtils</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
        </auxClass>
      </implementation>
    </componentType>
  </archTypes>
</xArch>
```

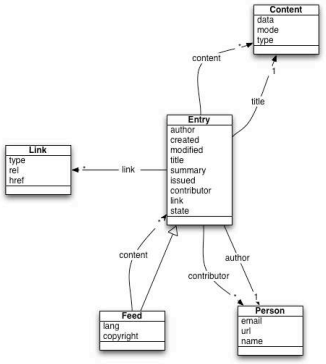
We can add implementation data to component and connector types in our television example, as shown in Figure 6. Here, we show that the television tuner component is implemented by two Java classes residing in the same JAR archive. The main class takes an initialization parameter of the television's model, which is useful if the component's implementation is multipurpose or reusable in multiple contexts (e.g., for many television models).
(ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 2, April 2005.)

ADLs in Practice



In practice, one uses graphical notations and editors for ADL specifications; these tools can also validate an implementation against its ADL.
(ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 2, April 2005.)

UML



- The de facto design language for OO systems
- But several artifacts needed for architecture are present, such as processes, nodes, and views
- But it still lacks adequate number of artifacts
- Some artifacts are not transferrable
- UML is designed to be a OO modelling language. Architecture is independent of programming languages
- At best, UML can deliver an informal description of the architecture

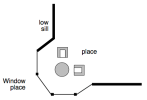
Software Architecture

- A software system's architecture is the set of *principal design decisions* about the system
  where "principal" is determined by system goals
- Software architecture is the blueprint for a software system's *construction* and *evolution*
  at any point in time, a system has only one architecture
- Design decisions encompass *every facet* of the system under development
  Structure • Behavior • Interaction • Non-functional properties

Window Place

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

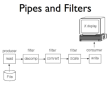
In every room where you spend any length of time during the day, make at least one window into a "window place"



Summary

Layers

Pipes and Filters



Four Patterns

Model-View-Controller

Representational State Transfer

http://en.wikipedia.org/wiki/Representational_State_Transfer

Specifying Architectures