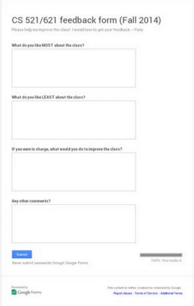
# Midterm Review
and
Architecture

## Course updates

- Homework 4 in class: Nov 17 and Nov 23
  - bring a laptop!
  - (if a laptop is an issue, talk to me privately)
- Homework 5 due Dec 1
- Final report due Dec 5
  - final presentations Dec 1, in class
- Midterm next Monday, Nov 10, in class

## Feedback

- http://goo.gl/rqRRln



CS 521/621 feedback form (Fall 2014)

Please help me improve the class! I would love to get your feedback. --Yuriy

What do you like MOST about the class?

What do you like LEAST about the class?

If you were in charge, what would you do to improve the class?

Any other comments?

Submit
100%: You made it.
Never submit passwords through Google Forms.
Powered by Google Forms
This content is neither created nor endorsed by Google.
Report Abuse - Terms of Service - Additional Terms

## Today's plan

- Midterm review
  - What kinds of questions to expect
  - Examples of questions
  - How to attack the hard questions
  - Topics to be covered
  - Your questions
- Software architecture

## What's the midterm like?

- Some true/false questions
- Some multiple choice questions
- Some reasoning questions

## True / False Example

Automatically predicting collaboration conflicts, if applied properly, would eliminate the need for resolving conflicts, which would greatly improve software development productivity.

## Multiple Choice Example

Rational Purify can find the following types of bugs (check all that apply):

A. Writing past the end of an array

B. Reading past the end of an array

C. Writing past the end of the first object in an array of objects

D. Null pointer exceptions

E. Using a different method than the developer intended

## Reasoning

- Reasoning are the harder questions that require abstraction and application of what you learnt.
- Reasoning questions will largely cover the papers presented in class, and the homework assignments

## Reasoning Example

Consider this simple concurrent program.

…

Does it have any races?

Why does CheckSync, from homework 3, report the following race?

…

## When Solving Reasoning Problems

- Important to pause for a moment to think about how to proceed.
- Plan your attack and evidence you will use to support your answer.
- You will have scratch paper to use to organize your thoughts (scratch will not be graded).

Come up with an answer, and its support, and write it clearly, concisely in the provided space.

## Topics to be covered

- Dynamic analysis
  - Daikon and Purify
- Test generation
  - Korat, Chronicler (field failures), web security testing, invariants to localize bugs, bias in bug prediction, coverage and mutation score measures of test quality
- Automated Bug Fixing
  - redundant methods, SemFix, configuration error diagnosis, invariant-driven bug diagnosis

## Topics to be covered

- Pair Programming
- Speculative Analysis
  - Quick fix scout
  - Crystal
- Refactoring
- Debugging (logging and compiler debugging)

## Now your chance

…to ask me questions about these topics

- dynamic analysis
- automated bug fixing
- test generation
- pair programming
- speculative analysis
- refactoring
- debugging

## Software Architecture

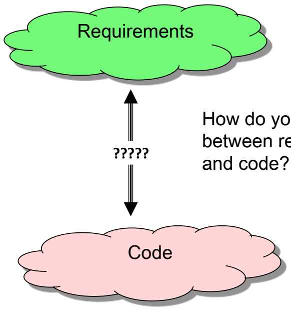
## Architecture

MIT Stata Center by Frank Gehry

## Why architecture?

> "Good software architecture makes the rest of the project easy."
> Steve McConnell, Survival Guide

## The basic problem



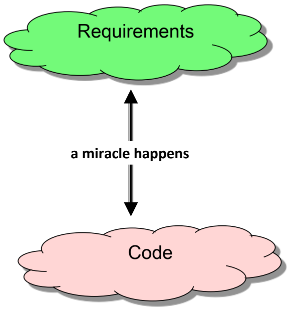
Requirements

?????

Code

How do you bridge the gap between requirements and code?

## One answer

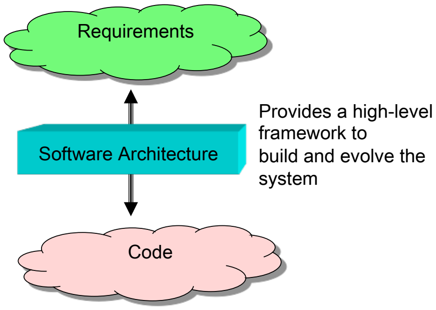


Requirements

a miracle happens

Code

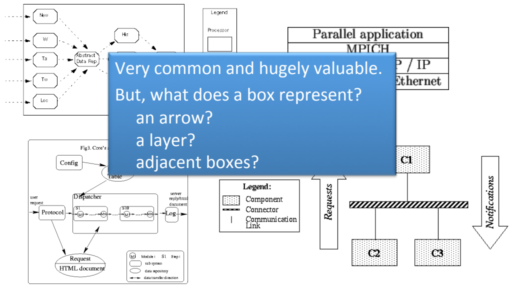
## A better answer



Provides a high-level framework to build and evolve the system

## What does an architecture look like?

## Box-and-arrow diagrams



Very common and hugely valuable.

But, what does a box represent?

- an arrow?
- a layer?
- adjacent boxes?

## An architecture: components and connectors

- *Components* define the basic computations comprising the system and their behaviors
  - abstract data types, filters, etc.
- *Connectors* define the interconnections between components
  - procedure call, event announcement, asynchronous message sends, etc.
- The line between them may be fuzzy at times
  - Ex: A connector might (de)serialize data, but can it perform other, richer computations?

## A good architecture

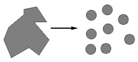
- Satisfies functional and performance requirements
- Manages complexity
- Accommodates future change
- Is concerned with
  - reliability, safety, understandability, compatibility, robustness, …

## Divide and conquer

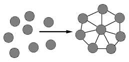
- Benefits of decomposition:
  - Decrease size of tasks
  - Support independent testing and analysis
  - Separate work assignments
  - Ease understanding
- Use of abstraction leads to modularity
  - Implementation techniques: information hiding, interfaces
- To achieve modularity, you need:
  - Strong cohesion within a component
  - Loose coupling between components
  - And these properties should be true at each level

## Qualities of modular software

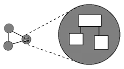
- decomposable
  - can be broken down into pieces



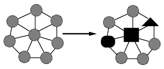
- composable
  - pieces are useful and can be combined



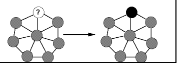
- understandable
  - one piece can be examined in isolation



- has continuity
  - change in reqs affects few modules



- protected / safe
  - an error affects few other modules



## Interface and implementation

- **public interface**: data and behavior of the object that can be seen and executed externally by "client" code
- **private implementation**: internal data and methods in the object, used to help implement the public interface, but cannot be directly accessed
- **client**: code that uses your class/subsystem

Example: *radio*

- public interface: the speaker, volume buttons, station dial
- private implementation: the guts of the radio; the transistors, capacitors, voltage readings, frequencies, etc. that user should not see

## UML diagrams

- UML = universal modeling language
- A standardized way to describe (draw) architecture
- Widely used in industry

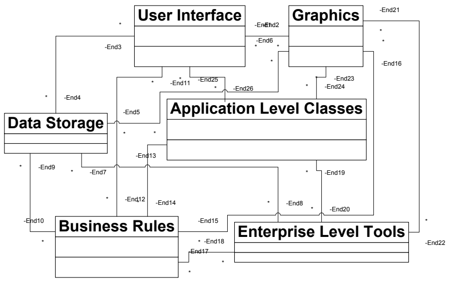
## Properties of architecture

- Coupling
- Cohesion
- Style conformity
- Matching
- Errosion

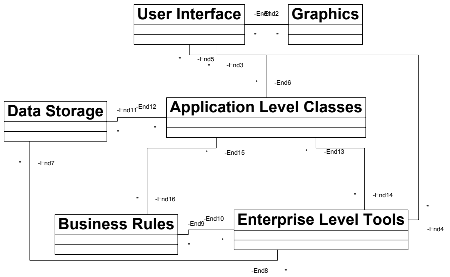
## Loose coupling

- *coupling* assesses the kind and quantity of interconnections among modules
- Modules that are loosely coupled (or uncoupled) are better than those that are tightly coupled
- The more tightly coupled two modules are, the harder it is to work with them separately

## Tightly or loosely coupled?



## Tightly or loosely coupled?



## Strong cohesion

- *cohesion* refers to how closely the operations in a module are related
- Tight relationships improve clarity and understanding
- Classes with good abstraction usually have strong cohesion
- No schizophrenic classes!

## Strong or weak cohesion?

```
class Employee {

public:
  …
  FullName GetName() const;
  Address GetAddress() const;
  PhoneNumber GetWorkPhone() const;
  …
  bool IsJobClassificationValid(JobClassification jobClass);
  bool IsZipCodeValid (Address address);
  bool IsPhoneNumberValid (PhoneNumber phoneNumber);
 …
  SqlQuery GetQueryToCreateNewEmployee() const;
  SqlQuery GetQueryToModifyEmployee() const;
  SqlQuery GetQueryToRetrieveEmployee() const;
 …
}
```

## An architecture helps with

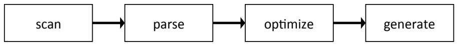
- System understanding: interactions between modules
- Reuse: high-level view shows opportunity for reuse
- Construction: breaks development down into work items; provides a path from requirements to code
- Evolution: high-level view shows evolution path
- Management: helps understand work items and track progress
- Communication: provides vocabulary; pictures say $10^3$ words

## Architectural style

- Defines the vocabulary of components and connectors for a family (style)
- Constraints on the elements and their combination
  - Topological constraints (no cycles, register/announce relationships, etc.)
  - Execution constraints (timing, etc.)
- By choosing a style, one gets all the known properties of that style (for any architecture in that style)
  - Ex: performance, lack of deadlock, ease of making particular classes of changes, etc.

## Styles are not just boxes and arrows

- Consider pipes & filters, for example (Garlan and Shaw)
  - Pipes must compute local transformations
  - Filters must not share state with other filters
  - There must be no cycles
- If these constraints are violated, it's not a pipe & filter system
  - One can't tell this from a picture
  - One can formalize these constraints



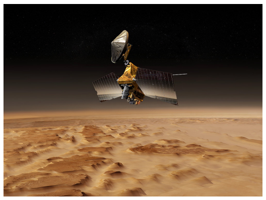
## The design and the reality

- The code is often less clean than the design
- The design is still useful
  - communication among team members
  - selected deviations can be explained more concisely and with clearer reasoning

## Architectural mismatch

- Mars orbiter loss

  NASA lost a 125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation



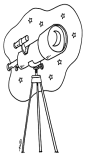
## Architectural mismatch

- Garlan, Allen, Ockerbloom tried to build a toolset to support software architecture definition from existing components
  - OODB (OBST)
  - graphical user interface toolkit (Interviews)
  - RPC mechanism (MIG/Mach RPC)
  - Event-based tool integration mechanism (Softbench)
- It went to hell in a handbasket, not because the pieces didn't work, but because they didn't fit together
  - Excessive code size
  - Poor performance
  - Needed to modify out-of-the-box components (e.g., memory allocation)
  - Error-prone construction process
- Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software 12*, 6 (Nov. 1995)
- Architecture should warn about such problems (& identify problems)

## Views

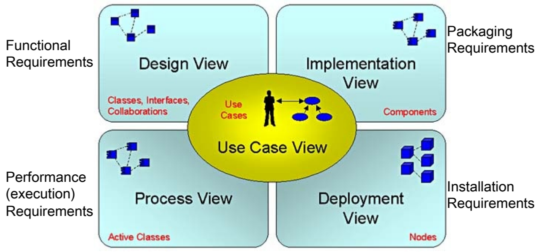
A <u>view</u> illuminates a set of top-level design decisions

- how the system is composed of interacting parts
- where are the main pathways of interaction
- key properties of the parts
- information to allow high-level analysis and appraisal



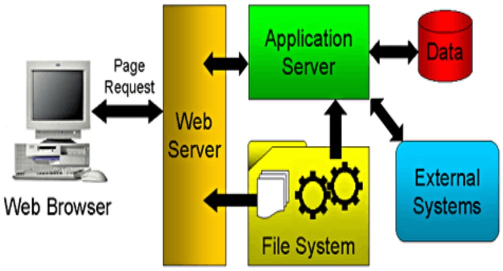
## Importance of views

Multiple views are needed to understand the different dimensions of systems

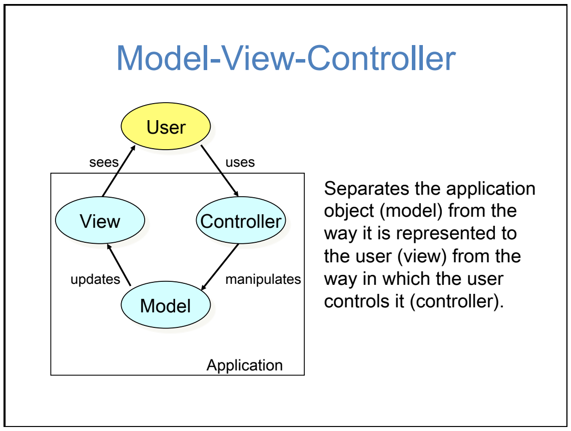


Booch

## Web application (client-server)

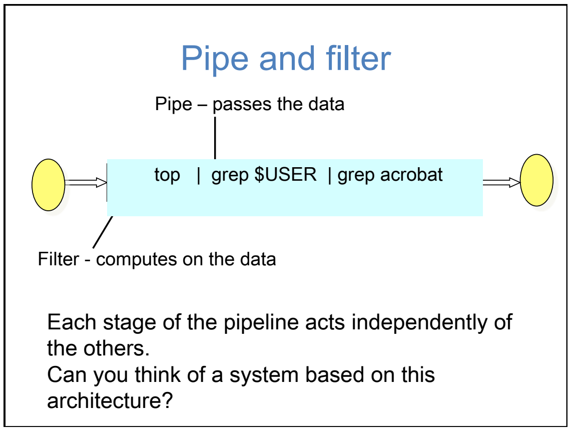


Booch



## Model-View-Controller

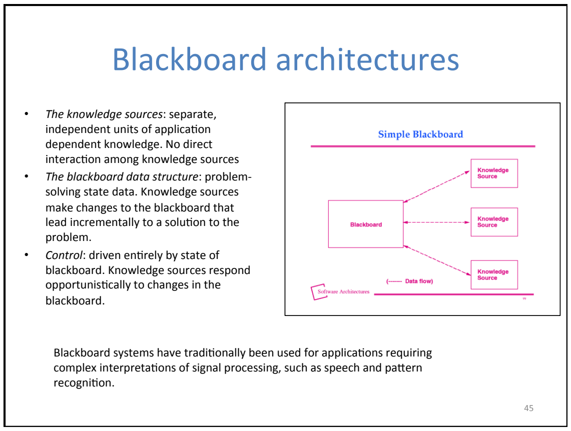
Separates the application object (model) from the way it is represented to the user (view) from the way in which the user controls it (controller).



## Pipe and filter

Pipe – passes the data
top | grep $USER | grep acrobat
Filter - computes on the data

Each stage of the pipeline acts independently of the others.
Can you think of a system based on this architecture?

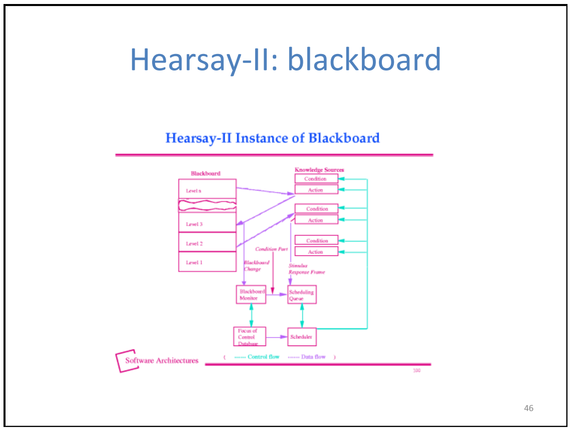


## Blackboard architectures

- *The knowledge sources*: separate, independent units of application dependent knowledge. No direct interaction among knowledge sources
- *The blackboard data structure*: problem-solving state data. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
- *Control*: driven entirely by state of blackboard. Knowledge sources respond opportunistically to changes in the blackboard.

Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition.

45



## Hearsay-II: blackboard

46