Architecture Evaluation

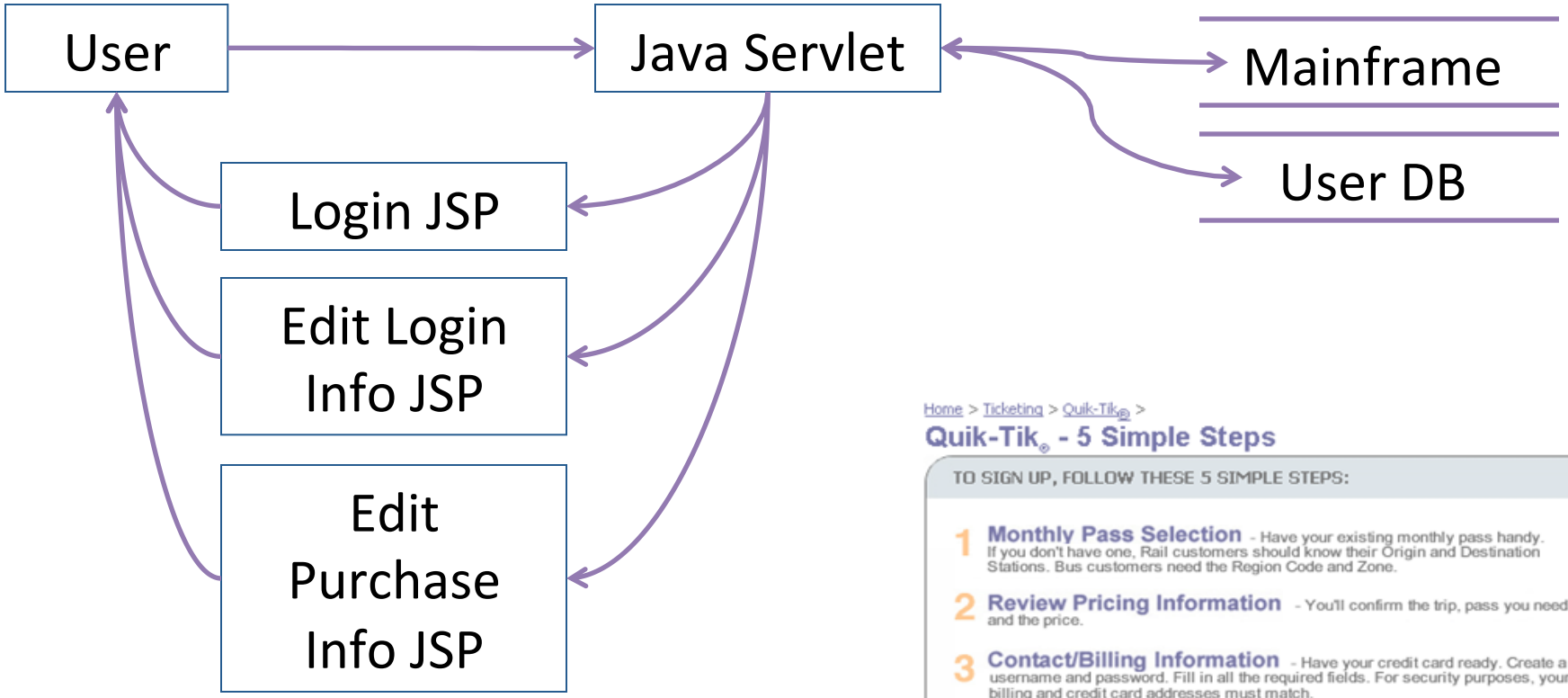
http://www.flickr.com/photos/bramkiforobrain/2704013961/

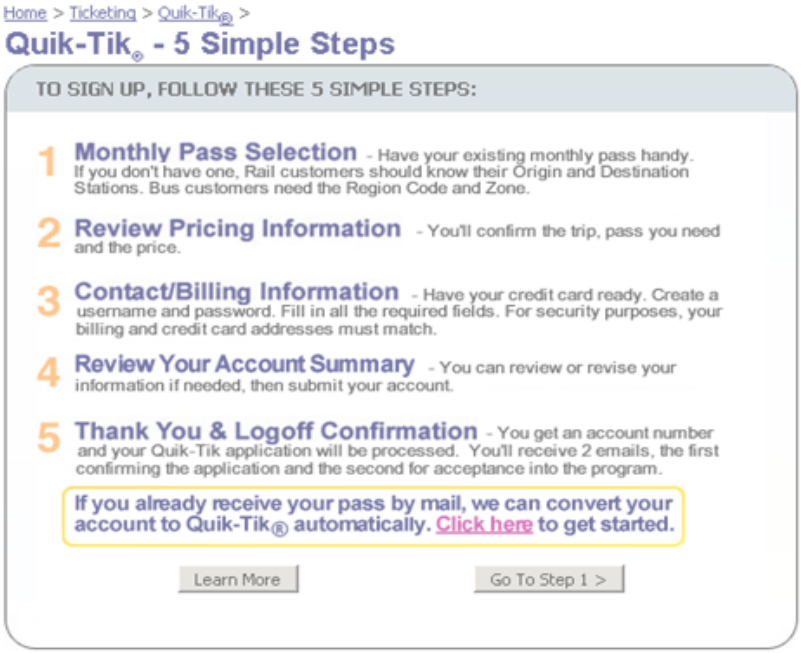
Four basic ways to evaluate architectural designs

1. Compare against desired **quality attributes**
2. Check for problematic **failure modes**
3. Walk through **use cases**
4. Verify conformance to **checklist** of principles

The goal is ***NOT*** to "prove" that the architecture is perfect, but rather to **identify opportunities** for improvement.

Example system from last lecture





Four basic ways to evaluate architectural designs

1. Compare against desired **quality attributes**
2. Check for problematic **failure modes**
3. Walk through **use cases**
4. Verify conformance to **checklist** of principles

The goal is ***NOT*** to "prove" that the architecture is perfect, but rather to **identify opportunities** for improvement.

Quality attributes of great software

- Reliability
- Efficiency
- Integrity
- Usability
- Maintainability
- Testability
- Flexibility
- Portability
- Reusability
- Interoperability

Checking against quality attributes

- **Maintainability** (includes modifiability)
  - How hard will it be to make anticipated changes?
- **Efficiency** (includes performance)
  - Can the system respond quickly, do a lot of work per unit time, and scale to high loads?
- **Reliability**
  - Will it perform properly under assumed conditions?
- **Integrity** (includes security)
  - Is it possible put the system into a bad state?
- **Usability**
  - Can real users complete their goals with the system?

Checking against quality attributes

- **Testability**
  - Can you (semi-)automatically test if the system is right?
- **Flexibility** (includes robustness)
  - How easily can the system adapt to unusual conditions?
- **Portability**
  - Could you get the system to run on a new platform?
- **Reusability**
  - What parts of the system could you use in a new system?
- **Interoperability**
  - Can the system talk to other relevant systems?

Checking example system against **key** quality attributes

- Integrity: security
  - Are all communications and databases encrypted?
  - Is credit card info stored in any risky location?
- Integrity: another consideration
  - What happens when a credit card expires?
- Efficiency
  - What platform is used to run the servlet & JSPs? What throughput and response time is likely?

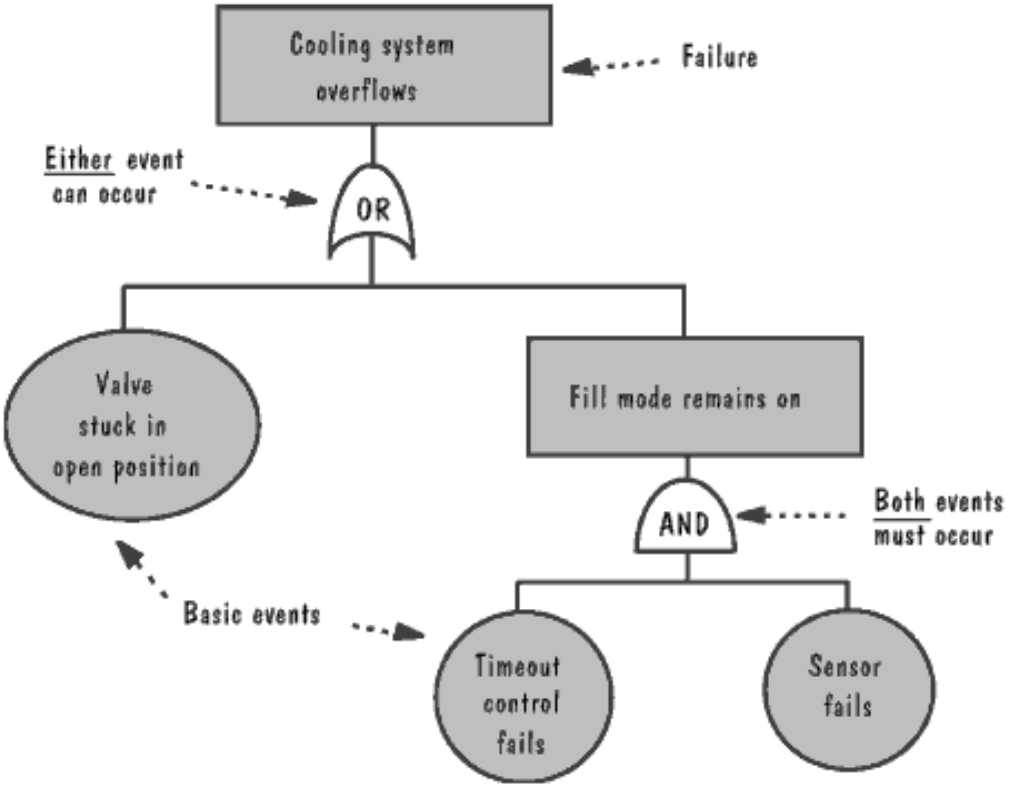
Four basic ways to evaluate architectural designs

1. Compare against desired **quality attributes**
2. Check for problematic **failure modes**
3. Walk through **use cases**
4. Verify conformance to **checklist** of principles

The goal is ***NOT*** to "prove" that the architecture is perfect, but rather to **identify opportunities** for improvement.

Checking against failure modes

What does it take to cause a "very bad thing"?

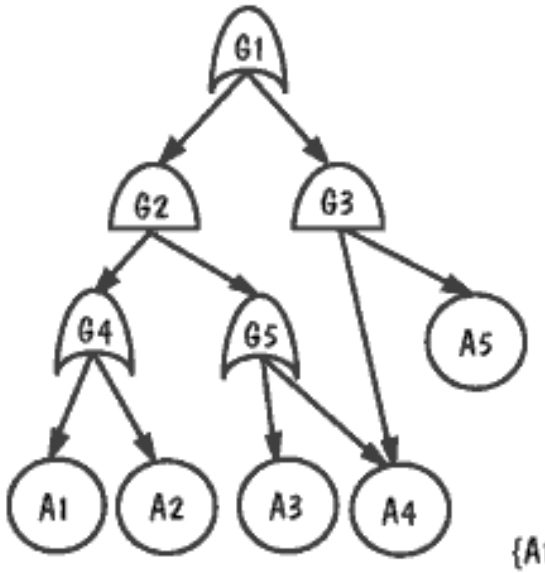


Identifying failures

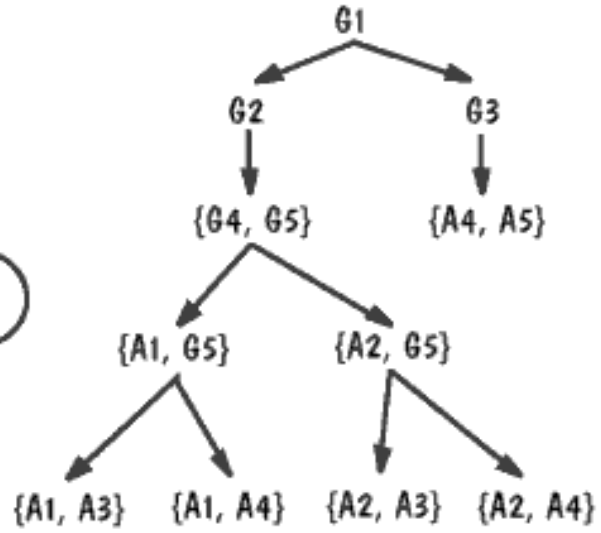
- List main failures that you expect from your system
- Work backwards identifying what can cause that failure
- Use fault trees and cut-set trees to help you identify where failures arise

Checking against failure modes

Trace the basic events through the "AND" and "OR" gates of the fault tree.



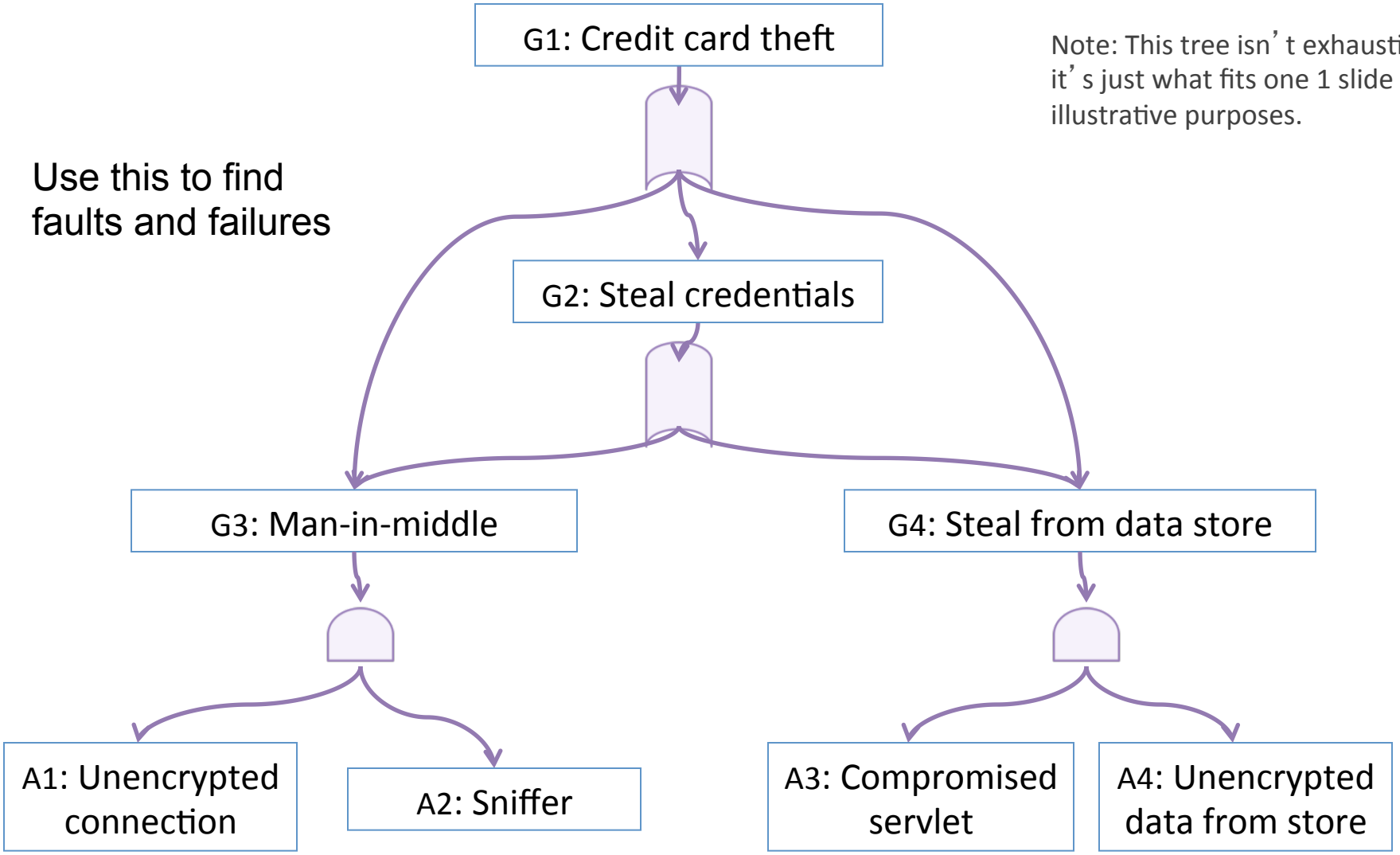
Fault tree



Cut-set tree

Checking against failure modes

Use this to find faults and failures



Note: This tree isn't exhaustive; it's just what fits one 1 slide for illustrative purposes.

Errors, faults, and failures

- **Error**: discrepancy between intended behavior and actual behavior, usually caused by faults
  - Prevent errors by establishing requirements up-front, by doing design up-front, by pair programming, …
- **Fault**: defect in the system, may or may not lead to failure
  - Prevent errors from causing faults by using unit testing, system testing, & similar techniques
- **Failure**: undesirable event caused by fault
  - Prevent faults from causing failures through redundancy, transactions, graceful degradation, etc

Strengths and weaknesses of alternate architectures

- Would the risk of a security breach failure mode have been higher in a peer-to-peer architecture that extended beyond our intranet? Maybe.
  - But for a file-sharing system, total system blackout might be the failure mode of concern, in which case peer-to-peer might *reduce* the risk of failure
- Quality attributes: Would performance have been improved by a system fully implemented in EJBs?
  - But would interoperability have been reduced?

Four basic ways to evaluate architectural designs

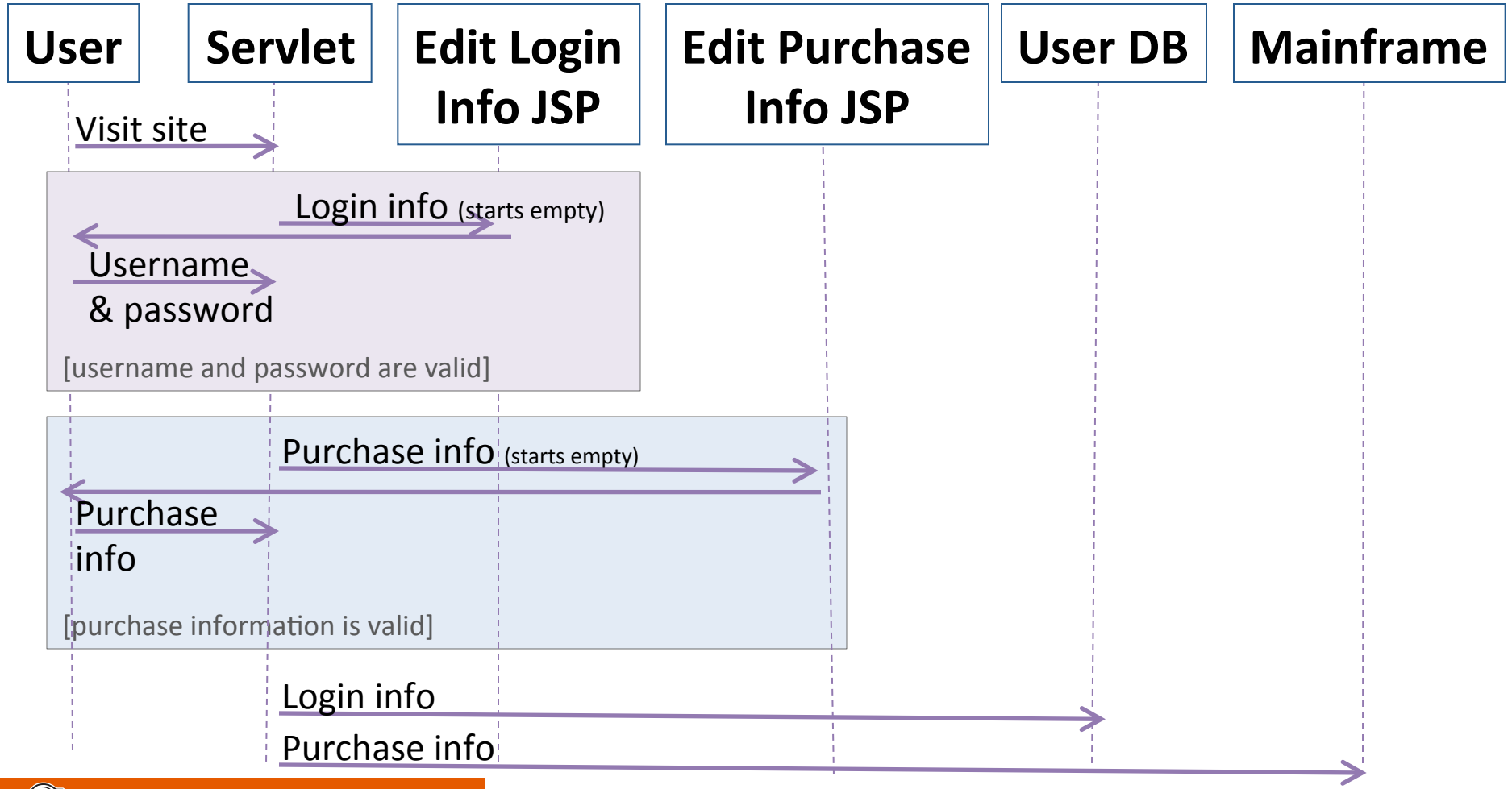
1. Compare against desired **quality attributes**
2. Check for problematic **failure modes**
3. Walk through **use cases**
4. Verify conformance to **checklist** of principles

The goal is ***NOT*** to "prove" that the architecture is perfect, but rather to **identify opportunities** for improvement.

Walk through use cases

- Methodically step through each use case
  - Are all the necessary components in the system?
  - Are they connected correctly?
  - Are all your arrows pointing the right direction?
  - Does the system enact the right state changes?
  - Does performing a use case prevent subsequently performing any other use cases?

Having a sequence diagram really helps to prevent surprises



Other ways to walk through use cases

- Use your requirements document and specification to your advantage
- Compare your architecture documentation to what you've already created to identify any gaps you may not have seen

Four basic ways to evaluate architectural designs

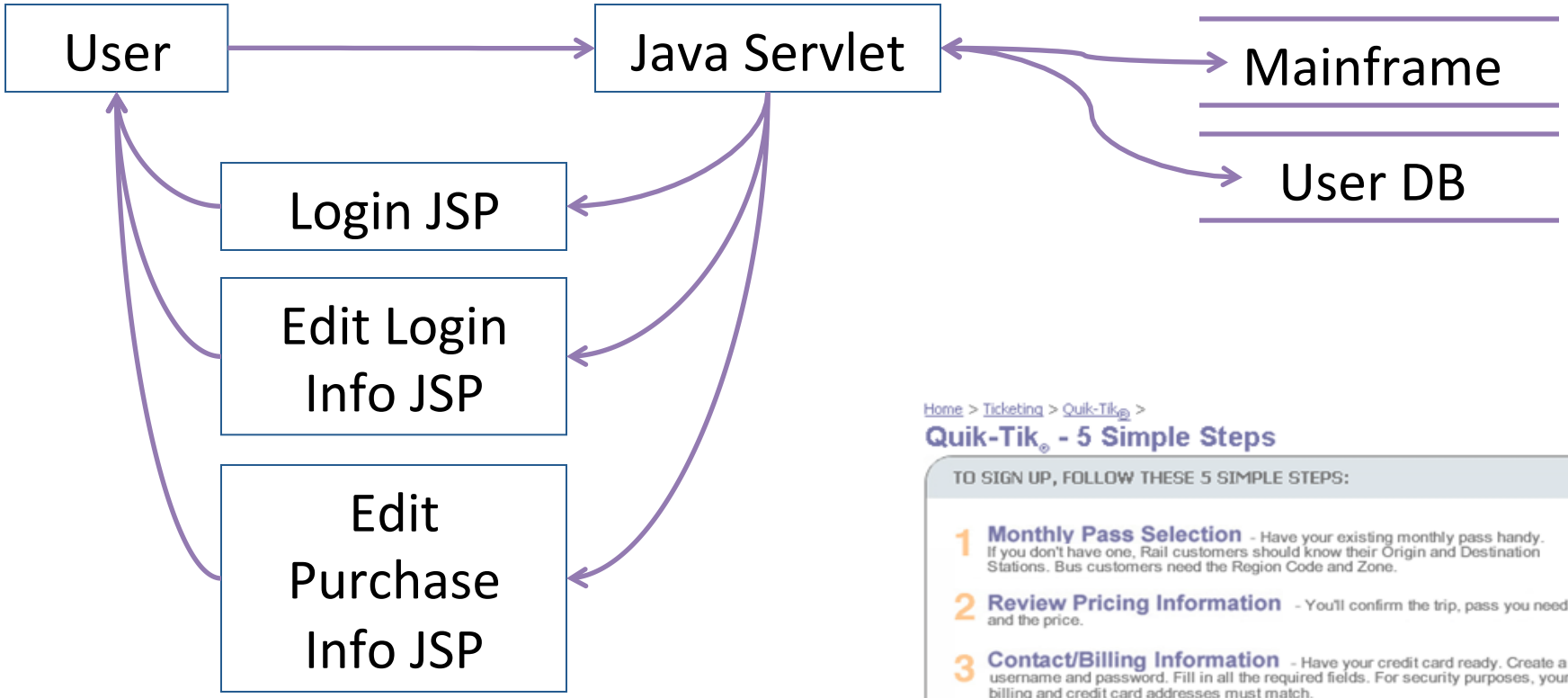
1. Compare against desired **quality attributes**
2. Check for problematic **failure modes**
3. Walk through **use cases**
4. Verify conformance to **checklist** of principles

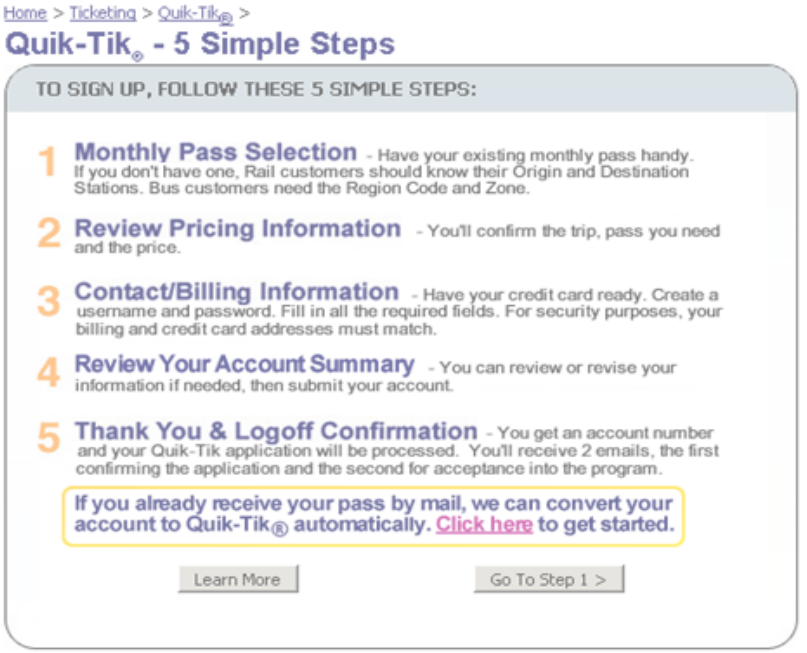
The goal is ***NOT*** to "prove" that the architecture is perfect, but rather to **identify opportunities** for improvement.

Verify conformance to checklist (Has a lot in common w/ quality attrs!)

- Is the architecture **modular**, well structured, and easy to **understand**?
- Can we improve the structure and understandability of the architecture?
- Is the architecture **portable** to other platforms?
- Are aspects of the architecture **reusable**?
- Does the architecture support ease of **testing**?
- Does the architecture maximize **performance**, where appropriate?
- Does the architecture incorporate appropriate techniques for handling **faults** and preventing **failures**?
- Can the architecture accommodate all of the expected design **changes** and extensions that have been documented?

Example system from last lecture





Checking example system vs checklist

- Modular?
  - "User DB" is a big black box. May need data-oriented decomposition.
  - "Mainframe" may need decomposition, too.
- Understandable?
  - Interaction among servlet and JSP is probably unclear to some programmers. May need some textual specification.
- Testable?
  - It depends on how fancy the JSP's HTML is. (DHTML?)
  - Mainframe is fully testable.
- Performance?
  - See earlier discussion regarding quality attributes.

Checking example system vs checklist

- Portable?
  - Not at all. Do we care? It depends.
- Reusable?
  - Probably only the data. Do we care? It depends.
  - Further decomposition might identify more reusable parts.

Checking example system vs checklist

- Fault handling & failure prevention?
  - Need to keep people from entering invalid data.
  - May need server redundancy.
  - All data should be backed up.
  - Encrypt communications + data in databases.
    - Security audit of system?
  - More consideration will be needed after the implementation is designed.

Checking example system vs checklist

- Maintainable?
  - Separating user interface code into JSPs generally improves maintainability.
  - But adding new fields requires changes in many places (e.g.: adding another field to purchase info, requires updating the servlet, JSP, and mainframe)
    - Object-oriented decomposition could have avoided this particular problem in this situation.

Take note of opportunities for improvement

- Encrypt database and connections
- Improve use case & sequence diagram to cover situations where credit card expires
- Decompose the database and mainframe
- Decompose servlet, looking for reusable parts

Consider tradeoffs

- Use DHTML or AJAX in HTML from JSP?
  - Might improve usability
  - Might hinder testability
- Specify server redundancy?
  - Might improve fault tolerance and performance
  - Might hinder integrity (keeping servers in sync)
- Move to object-oriented decomposition?
  - Might improve maintainability
  - Might hinder interoperability (legacy mainframe)

Takeaways

- Choices made at this stage have identifiable trade-offs
  - It is up to the stakeholders and developers to determine which benefits outweigh the drawbacks
- If I were to ask your team why you selected one architecture over another, I want to hear reasons that pertain both to the customer and the developers