Dynamically Adaptive System (DAS)

- Increasingly difficult to build software systems
  - continuously satisfy functional and non-functional requirements.
  - changing system and environmental conditions.
  - disallowed downtimes

Dynamically adapt in response to changing requirements and environmental conditions [McKinley04].
**DAS Design Challenges**

- Challenges in designing self-adaptive systems
  - high complexity.
  - difficult to specify, design, verify, and validate.
  - current lack of reusable design expertise.
  
  Need for a systematic approach to building adaptive systems with assurance!

**DAS Design Strategies**

- **Middleware**
  - abstracts complex and low-level reconfiguration tasks.
  - highly domain-specific.
  - Examples: MADAM [Mikalsen06], ACT [Sadjadi04], TRAP [Yang et al 02, Sadjadi et al DOA-05, DEAS-05]

- **Frameworks**
  - reusable code and infrastructure for supporting reconfiguration tasks.
  - design decisions already built into the framework.
  - Rainbow [Garlan04, Garlan06]
Designing Adaptive Systems

• **Reconfiguration Design Patterns** [Gomaa04]
  - Provides UML state diagram templates.
  - Identify when it is safe to reconfigure a system based on the application’s architecture.
  - Focus on behavior, rather than structural design decisions (patterns).

Our Objectives

• Discover recurring problems in self-adaptive system design.
  - Identify common solutions applied in practice.
  - Catalog general solutions to facilitate reuse across domains.
  - Identify common interactions between monitoring, decision-making, and reconfiguration.
  - Emphasize assurance throughout process.
Roadmap

- Harvesting Adaptation Design Patterns
- Adaptation Pattern Template
- Table of Adaptation Patterns
- Sample Adaptation Pattern
- Case Study
- Summary and Future Work

Approach

- Studied over 30 adaptive and autonomic systems.
  - Literature
  - Open Source

Discover existing solutions to recurrent problems!
Overview

Adaptation Patterns

Case Study

Discussion

References

Approach

- Studied over 30 adaptive and autonomic systems.
  - Literature
  - Open Source

Discover existing solutions to recurrent problems!

Pattern Template

- Each adaptation pattern’s contents are organized in a template similar to Gamma et al. [Gamma95]
Pattern Classifications

- Adaptation patterns are organized based on their main objective:
  - **monitoring**
    - systematically observe system and environmental conditions that may warrant reconfiguration.
  - **decision-making**
    - determine when and how to reconfigure an adaptive system.
  - **reconfiguration**
    - safely perform structural and behavioral changes in an adaptive system.
### Table of Adaptation Design Patterns

<table>
<thead>
<tr>
<th>Type</th>
<th>Pattern Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Sensor-Factory</td>
<td>Deploy sensors across a distributed infrastructure and probe components.</td>
</tr>
<tr>
<td></td>
<td>Reflective-Monitoring</td>
<td>Introspect on a component and dynamically alter a sensor’s behavior.</td>
</tr>
<tr>
<td></td>
<td>Content-based Routing</td>
<td>Route monitoring information based on the content of the message.</td>
</tr>
<tr>
<td>Decision-Making</td>
<td>Case-based Reasoning</td>
<td>Rule-based approach to selecting a reconfiguration plan.</td>
</tr>
<tr>
<td></td>
<td>Adaptation Detector</td>
<td>Interpret monitoring data and determine when an adaptation is required.</td>
</tr>
<tr>
<td></td>
<td>Architecture-based</td>
<td>Select reconfiguration plans based on system-model constraints.</td>
</tr>
<tr>
<td></td>
<td>Divide &amp; Conquer</td>
<td>Decompose a reconfiguration plan into simple reconfiguration plans.</td>
</tr>
<tr>
<td></td>
<td>Utility Theory-based</td>
<td>Select a reconfiguration plan that best balances multiple objectives.</td>
</tr>
<tr>
<td>Reconfiguring</td>
<td>Component Insertion</td>
<td>Safely insert and initialize a component at run time.</td>
</tr>
<tr>
<td></td>
<td>Component Removal</td>
<td>Safely remove a component at run time.</td>
</tr>
<tr>
<td></td>
<td>Server/Client Reconfig</td>
<td>Safely reconfigure a server component architecture at run time.</td>
</tr>
<tr>
<td></td>
<td>Decentralized</td>
<td>Safely insert and remove components from a decentralized component architecture at run time.</td>
</tr>
</tbody>
</table>

### Case Study

- **ZAP.com**
  - Adaptive News Web Server.
  - leverage adaptation to address unpredictable environment conditions.
    - “slashdot” effect.

- modeled after Rainbow’s Z.com [Garlan06] (Exemplar)
  - different design/implementation
  - same observable reconfiguration behavior
ZAP.com

- Functional Requirements
  - provide basic HTML functionality to clients.
  - operational costs may never exceed allocated budget.
  - serve “best” quality content whenever possible.
    - graphical content.
  - avoid losing customers due to high response times.
    - textual content.

- Adaptation Objective
  - minimize operational costs and latency while providing graphical news content whenever possible.

- Possible Adaptations [Garlan06]
  - increment server pool if response time is high and does not exceed budget.
    - otherwise, switch to textual mode.
  - decrement server pool if response time is low and near budget limit.
  - switch to graphical delivery mode if response time is low.
ZAP.com

• Functional Logic Implementation
  - object-oriented multi-threaded server-client architecture built in Java.
  - services basic HTML requests
  - servelet queries workloads and dispatches HTML requests to the server pool.

ZAP.com

• Selected 6 Adaptation Patterns based on ZAP.com’s adaptation requirements.
  1. **Sensor Factory:** periodically monitors average latency of servers.
  2. **Adaptation Detector:** process monitoring data and detect when a reconfiguration is required.
  3. **Case-based Reasoning:** select reconfiguration plans based on specific conditions or events.
    - We leverage the utility-theory analysis performed by Garlan et al. for Z.com [Garlan06]
• 3 of 6 Adaptation Patterns handle reconfiguration

4. **Component Insertion**: safely insert and initialize servers.

5. **Component Removal**: safely remove servers from server pool.

6. **Server Reconfiguration**: queues incoming client requests while servers are reconfigured.

---

### Comparing Z.com and ZAP.com

**a) Z.com**
- Rainbow Core
- GaugeCoordinator
- AbstractGauge
- RegularPattern Gauge
- Model Manager
- Rainbow Solver
- Adaptation Manager

**b) ZAP.com**
- Sensor-Factory
- Resource Manager
- Abstract Sensor
- Registry
- Trigger
- Server
- RequestQueue
- LoadBalancer Servlet
- ServerPool
- AdaptationDriver
- Sensor Factory
- Adaptation Detector
- Case-Based Reasoning

**Key associations**
- Monitoring
- Decision-Making
- Reconfiguration

**Key implementations**
- Sensor Factory

**Key aggregations**
- Component Insertion, Component Removal, Server Reconfiguration
Discussion

• Advantages
  - encourage design reuse across different domains.
  - no steep learning curve.
  - developers select which adaptation mechanisms to include.
    - reduce code-bloat.
  - instantiated models can be verified to ensure they satisfy key properties [McUmber01].

Discussion

• Disadvantages
  - does not provide any code reuse.
  - does not automatically support a wide range of adaptation mechanisms.
  - developers are responsible for preserving safety during reconfiguration (using patterns).
Conclusions

• Catalogued 12 adaptation design patterns to support monitoring, decision-making, and reconfiguration.
  - result of studying and generalizing over 30 different adaptive and autonomic systems.
  - promotes separate development of functional logic and adaptive logic.

• Customized the pattern template by Gamma et al. [Gamma95] for adaptation concerns.

• Applied a set of adaptation patterns to re-engineer an adaptive news web server.

Future Work

• Identify and integrate additional adaptation design patterns.

• Refine current set of adaptation design patterns.

• Leverage aspect-oriented techniques to integrate adaptation design patterns with functional logic.

• Explore automatic refactoring to incorporate adaptation design patterns [GECCO-2010]

• Analyze resulting behavior of designs for adherence to assurance properties with AMOEBA (modular model checker) [AOSD-2009]
References


[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements ofReusable Object-Oriented Software. Addison-Wesley Professional, 1995.


Acknowledgements

- Students and faculty in the Software Engineering and Networking (SENS) Lab

- This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, CNS-0854931, CNS-0751155, CNS-0915855, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University.
Questions?

Contact Information:

Betty H.C. Cheng (chengb@cse.msu.edu)
Andres J. Ramirez (ramir105@cse.msu.edu)
Website: http://www.cse.msu.edu/~ramir105/