Appendices

Appendix A (from chapter 5)

This material describes the details of the pattern and tactic data from the patterns and fault tolerance tactics studied.

PATTERN AND TACTIC DATA

For the architect and developer, the key information is the data itself. In this section we describe the fault tolerance tactics we studied. We follow it with the data for the Layers pattern.

Descriptions of the Tactics

1. Tactics for Fault Detection:

   a. Ping/Echo: A monitoring component issues a ping message to one or more components under scrutiny, and expects to receive an echo message back within a predetermined time. If a component does not respond within the time limit, the monitoring component considers that component to be in failure mode, and takes corrective actions. Implementation requires that a monitoring process be created or used, and that all components being watched must be modified to handle the echo messages.

   b. Heartbeat: A component emits a heartbeat message at regular intervals and a monitoring component listens for it. If no heartbeat message is received within a predetermined time, the originating component is assumed to have failed, and corrective actions are taken. This tactic requires a monitoring component, and all components must be modified to send heartbeats at the proper intervals.

   c. Exceptions: Raise and handle exceptions. Exceptions are usually handled in the components in which they occur. Most modern programming languages include built-in support for exception handling. Implementation usually only requires minor, sub-architectural structural changes, such as the add of an exception handling block.

2. Fault Recovery – Preparation and Repair

   a. Voting: Processes running on redundant processors each take equivalent input and compute a single out value that is sent to a voter. The voter component decides which of the results is correct using an algorithm such as majority rules. The strongest approach is to implement each voting component independently; otherwise you can only detect hardware faults, and not algorithm faults. (If the
voting components are running the same software, this tactic becomes very similar to Active Redundancy; see below.) To implement voting, create a voter component, and either replicate or write a new voting component.

b. Active Redundancy: redundant components receive events in parallel, thus they are always in the same state. If one component fails, the other can immediately take over. This tactic that the processing component(s) to be replicated. It usually requires a central arbitrating component, although it is possible to make the redundant components perform the arbitrating without a central component.

c. Passive Redundancy: One component is the active or primary component. It updates the state of one or more backup components. If the primary component fails, a backup component will be in approximately the same state, and will take over. This tactic requires that the primary component to be replicated to form the backup, and both are modified to implement the state update protocol between the primary and backup(s). A central arbitrating component may be needed.

d. Spare: A standby spare can replace different components, and is booted for the particular component that failed. This is most common where multiple components sharing load; imagine a system with multiple identical servers to service requests. Such a configuration is often called “N + 1 sparing.” This requires the component of interest to be replicated, and changed to support the sparing. An important consideration is that all components must save state so the spare can replace any of them. A central arbitrating component may be needed.

3. Recovery – Reintroduction of a Failed Component

a. Shadow: When a component is restored, it runs in shadow mode behind an active component until its integrity is fully established. Implementation of this tactic requires a component to monitor the health of the “shadow” component. The shadow itself is probably replicated from another component.

b. State Resynchronization: Before a component is returned to service, synchronize its state with the current operation. The state synchronizing messages must come from somewhere; it is probably easiest to have them come from an active component; then a controlling component isn’t needed.

c. Checkpoint/Rollback: Record consistent states and have a path to roll back to them if necessary. Each component in question must define its consistent states and implement a way to record the checkpoints and roll back to them. A component can usually do this without the need for a central component.
However, note that the ease of implementation is based on how easy it is to define sane checkpoints – some systems have little notion of state.

4. Fault Prevention

a. Removal From Service: Remove a component from service to repair potential problems. For example, a component might be periodically restarted to prevent memory leaks from causing a failure. This can be implemented by modifying a component to restart itself, or by using a central monitoring component. If a central component is used, this tactic looks like a proactive process monitor; see below.

b. Transactions: Bundle actions into sets that can be undone all at once. Transactions are highly compatible with checkpoints as described above. Components are modified for transactions.

c. Process Monitor: When a fault is detected, a monitoring process manages the deletion of the process and its replacement. This is typically used together with the fault detection tactics of Ping/Echo or Heartbeat. Implementation requires a central monitoring component. The components being monitored may or may not need any changes.

Pattern and Tactic Interaction Data
We have organized the data by pattern. Within each, we give a short summary of the data for that pattern, followed by the descriptions of the tactics’ impact.

The impact of each tactic is described as very positive, positive, neutral, negative, or very negative, as described earlier. They are abbreviated with the symbols $++, +, \sim, -, --$, respectively. We show the Layers and Pipes and Filters patterns.

Layers
Summary: The Layers pattern is very common in all systems. It provides good support for many fault tolerance tactics, and should be considered for highly reliable systems.

1. Fault Detection

a. Ping/Echo: $+$: The monitor component is added. If all the layers are implemented in a single process, then only the top layer must respond to the ping. If each layer is a separate process, then one approach is to create a hierarchy: the monitor pings the top layer, and each layer pings the layer below it before responding with an echo. (Add in the Pattern, Minor modifications)

b. Heartbeat: $+$: Similar to ping/echo, except the top layer sends the heartbeat based on time, rather than an echo in response to a ping. (Add in the pattern, minor modifications)
c. Exceptions: + +: Exceptions can be handled or propagated through layers with few if any changes to structure. (Implemented in the components)

2. Recovery – Preparation and Repair

a. Voting: +: Voting requires that the main processing components be rewritten to implement additional voting components. A new component, the voter, is added, and can be added as the top layer above all the voting components. Thus the pattern gives moderate support to this tactic. (Add in the Pattern, minor modifications)

b. Active Redundancy: +: To implement this tactic, replicate the layers without change. Add an additional layer above both layered systems to arbitrate and distribute messages to the redundant components. If it is a distributed system, this begins to look like a Broker. (Replicate, minor modifications, possibly add within the pattern)

c. Passive Redundancy: +: Replicate the layers, and then modify one layer to send and receive state updates (keep it to a single layer). It is likely the top layer, but may be a different layer. If needed, a monitor component can be added as a layer above both. This tactic does not fit quite as well as Active Redundancy, but is still a positive fit with Layers. (Replicate, minor modifications, possibly add within the pattern)

d. Spare: ~: One must replicate the layers, and modify some part of the replicated components to save their state. The spare must have a way to synchronize itself with the component that fails. A monitoring layer is probably needed.) In all this, the Layers pattern doesn’t help, but it doesn’t hurt, either. (Replicate, moderate modifications, add in the pattern)

3. Recovery – Reintroduction

a. Shadow: +: The system will have been duplicated, following one of the redundancy tactics. An additional layer is added to assess the health of the system coming back into service. If there is a monitoring layer for fault detection or redundancy, it will be the same layer. (Minor modifications, Add in the Pattern is in conjunction with another tactic.)

b. State Resynchronization: +: Layers provides some support for states – state information can be encapsulated in a single layer, which can help with resynchronization of a component returning to service. A monitoring layer may be needed, and a layer is modified to send state information to the other component. The layers will have been replicated in another tactic. (Minor modifications, Add in the Pattern is in conjunction with another tactic.)
c. Rollback: + +: Layers can provide checkpointing of data which can make it easier to recover the data or the state. A layer can hold a request to a lower layer that is pending. If the lower layer is unable to complete the transaction, the higher layer can readily undo the transaction. (Implemented in the components)

4. Prevention

a. Removal from Service: ~: A monitoring component is needed. If all the layers are implemented in a single process, then the monitor is added as the top layer (Add in the pattern, little or no modification needed.) However, if each layer is a separate process, either the monitor must control each layer, or each layer must control the layer under it (both are significant modifications.) So depending on the implementation of Layers, it is either positive or negative.

b. Transaction: + +: Layering encourages the packaging of actions into transactions; usually done at the highest layers. One can create commands to the

c. Process Monitor: ~: Implementation depends on how the layers are implemented. If they are all in a single process, then simply add the process monitor as a top layer (Add in the pattern, little or no modification needed.) However, if each layer is a separate process, then either the process monitor must monitor each layer individually, or each layer becomes the process monitor of the layer under it. Both these cases require significant modifications to components. So it is either positive or negative, depending on the implementation of the Layers.

Pipes and Filters

5. Fault Detection

a. Ping/Echo: - -: A central monitoring process must be added, which must communicate with each filter. Each filter must be modified to respond in a timely manner to the ping messages. This not only affects the structure of the pattern, but may conflict with realtime performance. (Add out of the Pattern, along with moderate changes to each filter component)

b. Heartbeat: - -: Similar to ping/echo. It is a bit easier to add the heartbeat generating code to the filters, because they don’t have to respond to an interrupt. However, each filter must still send the heartbeat in response to a timer. (Add out of the Pattern, along with moderate changes to each filter component.)

c. Exceptions: - -: The problem here is who should catch an exception when one is thrown. If the exception can be completely handled within a single filter
component, then there is no problem. However, if the filter cannot fully handle the exception, who needs to know? Depending on the application, it may be the subsequent filter (simple modifications needed to the filters), or a central process might need to know (Add out of the Pattern, along with moderate changes to each filter component.)

6. Recovery – Preparation and Repair

a. Voting: + +: To implement voting, create different filters as the voting components. Create the receiving component (the voter) as a filter. To distribute the input to the different voting filters, use a pipe that has one input and multiple outputs (e.g., the Unix “tee” command.) (Add in the Pattern, but the work besides using different algorithms in the different filters is very straightforward; almost trivial.)

b. Active Redundancy: + +: Replicate filter components. Send the same stimuli to redundant filters. Use a pipe or a final filter to receive the results from the redundant filters. You can arrange it so you take the first one finished, which improves performance; a common goal in Pipes and Filters. (Replicate, plus add a trivial pipe or filter to handle the results, as well as a distribution pipe, as noted in Voting. As in voting, the adds here are trivial.)

c. Passive Redundancy: -: Replicate the filter you are backing up. Then modify the primary filter to send occasional updates to the backup filter. The backup filter must be modified to receive the updates, rather than the normal input data. A pipe or trivial filter is needed to handle the results, just as in Active Redundancy. This can be done within the pattern, but Active Redundancy is generally a superior tactic, and it fits so well, that this pattern is not recommended with Pipes and Filters. (Replicate, plus significant changes to the filters.)

d. Spare: +: Set up a device as the spare, with the ability to run as any of the different filters. Create a new filter that handles distribution of work. It must detect when a filter does not respond to sending work (e.g., did the data write fail), and then initialize the spare as that kind of filter. (Add in the pattern, but the new filter is not trivial.)

7. Recovery – Reintroduction

a. Shadow: +: In Pipes and Filters, Shadow is implemented similarly to Voting. In this case, the receiving component checks the results from the shadow against the results of the primary filter to see if they are correct. It may be necessary to communicate the state of the shadow filter back to the filter that distributes the work. Note that the shadow filter itself should need no changes. (Duplication with simple Add in the Pattern, plus possible small Modify to two filter components.)
b. State Resynchronization: - -: The biggest problem with this tactic is filters should not have states except within processing of one piece of data. And in that case, it usually makes most sense to restart processing of that data from the beginning. If you must to implement this tactic, define states for each filter and create a mechanism to restore a filter to the proper state when it comes back up. That may require a monitoring process. (Major changes to components, plus possible Add out of the pattern.)

c. Rollback: - -: Checkpointing is easy to do. However, once the data passes to the next filter, it is extremely hard to undo it -- it is gone. If you must use it, use a monitoring process and a protocol of checkpoints to ensure the integrity of the data at the end of each filter. (Add out of the pattern, plus major changes to components.)

8. Prevention

a. Removal from Service: - -: Use a monitoring process to decide when to remove a filter from service. (Add out of the pattern. Minor changes may be needed for reconfiguration.)

b. Transaction: ~: The first filter might create the transactions. However, filters work more naturally on streaming data than on transaction-oriented data.

c. Process Monitor: - : Use a monitoring process to detect when a filter fails, and reconfigure the system. (Add out of the pattern. Minor changes may be needed for reconfiguration.)

Broker

1. Fault Detection

a. Ping/Echo: + +: The Broker component can implement a ping/echo and can even have a message serve double duty as a request and as a ping message. This is both efficient and easy to implement. (Implemented in the Pattern)

b. Heartbeat: +: As in ping/echo, the Broker is the monitor component. The other components must implement a heartbeat mechanism, with messages independent of the normal control messages. (Minor modifications to components needed.)

2. Recovery – Preparation and Repair

a. Voting: + +: The broker distributes work to multiple servers, acts as the arbiter among them. (Implemented in the pattern. Different voting components are implemented as server components.)

b. Active Redundancy: + +: The broker component provides a natural mechanism for distributing the same messages to redundant servers, and management of
the swap to the redundant server. Servers might be replicated, or that might be done already. (Implemented in the Pattern.)

c. Passive Redundancy: +: The Broker component manages updating the spare with the active’s state. Active Redundancy is such a good fit that it is recommended over this tactic. However, it is a good fit for duplicating the Broker component. (Minor modifications to the active and backup components.)

d. Spare: + +: If the system has multiple servers with different responsibilities, then the broker component manages bringing in a spare for any of the failed servers. (Implemented in the pattern; a few minor modifications may be needed.)

3. Recovery – Reintroduction

a. Shadow: + +: The broker component is a natural place for monitoring servers that return to service. The broker can keep track of the health of the servers, and mark a server as a shadow until it returns to full operation. (Implemented in the pattern)

b. State Resynchronization: + +: When a server comes back into service, the Broker component can send it state information to synchronize it. (Implemented in the pattern.)

c. Rollback: + +: Broker-controlled systems tend to be transaction-oriented. The actual rollback happens within the server components, while the control of rollback may be within the server, or in the Broker, depending on the application. (Implemented in the pattern.)

4. Prevention

a. Removal from Service: + +: The broker component can act as an arbiter for the server, and remove it from service if it detects error conditions. (Implemented in the pattern.)

b. Transaction: + +: Broker systems tend naturally to be transaction-oriented. The units of transaction would be defined in terms of work done on the server. (Implemented in the pattern.)

c. Process Monitor: + +: It is natural for the broker component to detect if a server fails and restart it. (Implemented in the pattern.)
Appendix B, from Chapter 6

This appendix contains detailed information about fault tolerance tactics and their impact in multi-pattern architectures.

Tactics’ Impact on Architecture Components

Here we examine all the reliability tactics from Bass, et al. [15]. For each, we examine whether it impacts some components, or whether it impacts all components. If it impacts all components, we determine whether it utilizes a central component. We determined that all tactics could be classified into these three categories. The three categories are (repeated from the main text):

1. Impacts all components
2. Impacts all components, plus a central component
3. Impacts some components

Analysis of the tactics follows:

1. Tactics for Fault Detection:
   a. Ping-Echo: This requires that each component must respond to the ping messages. In addition, one component must initiate the ping messages, and manage the responses. (Category 2: impacts all components, plus a central component.)
   b. Heartbeat: Each component must emit a periodic heartbeat to a central component, which takes corrective action if a component fails to do so. (Category 2: impacts all components, plus a central component.)
   c. Exceptions: Generally, exception raising and handling should be done consistently across the application, which means that each component must raise exceptions as well as respond to other exceptions. (Category 1: There is no explicit central component, but all exceptions must be appropriately handled. This may hint at a central “handler of last resort”.)

2. Fault Recovery – Preparation and Repair
   a. Voting: While the critical components are implemented multiple times (in different ways), only a few actually implement the voting process. In fact, it may be contained in a single component that gathers the votes and selects the winner. (Category 3, impacts some components. Which components must vote depends on the system requirements.)
   b. Active Redundancy: Similar to voting in that a single component can manage the redundant components. Some systems are entirely redundant, while others may have a few critical components that are
redundant; perhaps the data store or communication infrastructure components. Which components must be duplicated depends on the system requirements (Category 3.)

c. Passive Redundancy: This impacts more than one component, because the passive component must receive state updates from the active. It is likely that the modifications can be confined to a few components, though. (Category 3, impacts some components; who is duplicated depends on the requirements; same rationale as for Active Redundancy.)
d. Spare: Same as Passive redundancy in terms of impact. (Category 3, impacts some components; who is duplicated depends on the requirements; same rationale as for Active Redundancy.)

3. Recovery – Reintroduction of a failed component
a. Shadow: Impact depends on whether “shadow” mode is a visibly different way of running, or whether it is just that the shadow is running in the background, and must be verified to be running correctly. (The second is the more common interpretation of the tactic.) The second case is more desirable, and means changes are limited to one or a few components. (Category 3, impacts some components; who is shadowed depends on the requirements. Rational is similar to that of Active Redundancy.)
b. State Resynchronization: This tactic is that when a failed component is brought back into service, its state must be synchronized with the rest of the system. It is typically used with active and passive redundancy. The components affected are the redundant components and any component that controls them. (Category 3, impacts some components; which components depends on the requirements.)
c. Checkpoint/Rollback: Likely to affect all components that deal with the data/transactions/state that must be checkpointed and rolled back. This impacts some or all of the components that are directly involved with the state of the data being processed. (Category 3, because not all components are directly involved in the state of the system)

4. Fault Prevention
a. Removal from Service: This likely has some impact on all the components which can be removed from service, in that there must be a way to do so gracefully. In other words, when a component is removed from service, it must do something as it exits. Therefore, the component that is removed from service must change, and there must be a component that manages the removal. (This works out to be Category 3, impacts some components; one must analyze the requirements to determine which components.)
b. Transactions: Processing is bundled into transactions which can be undone all at once, if necessary. Transactions make checkpointing and rollback easier. Affects those components that deal with the details of the transactions. (Category 3: impacts some components. The requirements help shape which components deal with transactions.)

c. Process Monitor: Similar to Removal from Service, above. It depends on how many processes are monitored. If it is all, this tactic resembles Ping-Echo structurally. It is likely that the number of processes is limited to a few critical processes. In this case, it has a similar structure to Removal from Service. (Category 3, impacts some components; one must analyze the requirements to determine which components.)

Analysis of Tactics with Pairs of Patterns

In our studies of software architectures [62], we found that certain patterns were often found together in the same architectures. We identified six pairs of patterns that were particularly common. We examined how the reliability tactics from [15] would be likely to be implemented in these pattern pairs. We also considered how the effort to implant the tactic compared to the effort of implementing the tactic in each individual pattern.

Broker-Layers (the most common pair we saw)

1. Tactics for Fault Detection:
   a. Ping-Echo: (All components plus control) Layers: +, Broker: ++. Therefore, overall score should be +, or better. Analysis: Each component must respond to the ping message to a central component. This is a good fit for both patterns, but is particularly good for Broker, because the broker component is a natural central component, and has connections to all components. In a system with both patterns, the broker component will be the central component, thus, the fit is very good; ++.

   b. Heartbeat: (All components plus control) Layers: +, Broker: ++. Analysis: The same rationale as for Ping-Echo results in a very good fit; ++.

   c. Exceptions: (All components) Layers: ++, Broker: +. Therefore, overall score should be +. Analysis: all components must deal with exceptions. The hierarchy of layers in the Layers pattern is particularly good for handling exceptions, but not all components in the Broker pattern will be in the Layers, so the overall impact is +, not ++.

2. Fault Recovery – Preparation and Repair
a. Voting: Voting (Some components) Layers: +, Broker: ++. Therefore, the overall score should be up to ++, depending on system requirements. Analysis: Consider Layers by itself: the layers structure fits well with voting, but you need another layer to handle the voting and select the winner. If you have Broker too, though, you already have a component that handles multiple independent components, and can easily implement the voting control. So the tactic inherits the better fit, from Broker. However, this depends on exactly what components must vote; if they are components of the Layer pattern, then the score will be just +. The decision about which components are voting components depends on the requirements: what processing must always be correct.

b. Active Redundancy: (Some components) Layers: +, Broker: ++. Therefore, the overall score should be up to ++, depending on requirements. Analysis: If the components that are to be duplicated are controlled by the broker component, then there is an excellent fit; otherwise, they will be managed by a Layer; a good fit. Requirements will dictate which component(s) should be duplicated.

c. Passive Redundancy: (Some components) Layers: +, Broker: ++. Therefore, the overall score should be up to ++, depending on requirements. Analysis: Same arguments as for active redundancy.

d. Spare: (Some components) Layers: ~, Broker: ++. Therefore, the overall score should be up to ++, depending on requirements. Analysis: Same arguments as for active redundancy.

3. Recovery – Reintroduction of a failed component

a. Shadow: (Some components): Layers: +, Broker: ++. Therefore, the impact should be up to ++, depending on requirements. Analysis: Same arguments as for Active Redundancy.

b. State Resynchronization: (Some components): Layers: +, Broker: ++. Therefore, impact should be up to ++. Analysis: Consider a layered architecture where state resynchronization is being added. See SERENE paper for details. Now consider the same system with Broker added. Would the Broker components pick up the state resynchronization work, or would it still have to be done within the layers? It depends: the business logic; i.e., the significant states probably resides in the layers, so the state resynchronization may be within the Layers, increasing the impact to +. So it is possible that the impact is +, although perhaps the Broker component may pick up the work.
c. Checkpoint/Rollback: (Some components): Layers: ++, Broker: ++. Therefore, the impact should be ++. Analysis: Checkpointing can easily be implemented in either Layers or Broker.

4. Fault Prevention

a. Removal from Service: (Some components) Layers: ~, Broker: ~. Therefore, impact should be ~. Analysis: The requirements dictate which components are to be removed from service. The impact will match the impact of the pattern where the component is removed from service, and the impact is about the same.

b. Transactions: Layers: ++, Broker: ++. Therefore, the impact should be ++. Analysis: Layers provides a natural way to package transactions. On the other hand, Broker is set up to naturally handle transactions. So the result is strong support of this tactic.


Layers-Shared Repository (the second most common pair we saw)

1. Tactics for Fault Detection:

a. Ping-Echo: (All components plus control) Layers: +, SR: ~. Therefore, overall score should be ~ or better. Analysis: A Shared Repository would usually reside at a low level in a layers hierarchy. So it inherits the behavior of the Layers pattern. So it might actually be +. On the other hand, if the SR is shared by any component outside the Layers hierarchy, then it’s probably necessary to ping the SR, whether or not the individual layers are pinged. So it is probably not quite a +, and back down to a ~.

b. Heartbeat: (All components plus control) Layers: +, SR: ~. Same rationale as for Ping-Echo.

c. Exceptions: (All components) Layers: ++, SR: ++. Therefore, overall score should be ++. Analysis: the exception handling provided as part of commercial databases should fit very naturally into a layered hierarchy. This should be a very good fit.

2. Fault Recovery – Preparation and Repair

a. Voting: Voting (Some components) Layers: +, SR: ~. Therefore, the overall score should be +. Analysis: According to the scores, SR shouldn’t care about voting, but Layers can implement the voting control in an upper layer. As the SR is a low layer, implementation of Voting should happen in
layers above it, and it shouldn’t care. So we get the impact on Layers, which is +.

b. Active Redundancy: (Some components) Layers: +, SR: ++. Therefore, the overall score should be up to ++. Analysis: It depends on what you want to replicate. If it is the database, then most commercial databases do this for you. However, if it is a different part of the system, or more of the system, then you involve the Layers, and the impact is +, not ++. NOTE: This is an example of needing to analyze the application’s requirements.

c. Passive Redundancy: (Some components) Layers: +, SR: ++. Therefore, the overall score should be up to ++. Analysis: Same arguments as voting.

d. Spare: (Some components) Layers: ~, SR: -. Therefore, the overall score should be up to ~. Analysis: The analysis for Shared Repository says that to implement a Sparing scheme, one needs to add a layer above the database, thus creating a small Layers pattern. We have that already with this composition, so we can usually pick up the impact on Layers, rather than SR. This depends on the application, requirements of course, namely what part of the application needs to run nonstop.

3. Recovery – Reintroduction of a failed component

a. Shadow: (Some components): Layers: +, SR: ~: Therefore, the impact should be up to +. Analysis: similar to voting.

b. State Resynchronization: (Some components): Layers: +, SR: ++. Therefore, impact should be up to ++. Analysis: It depends on the application’s requirements. If there are requirements only to keep the database in synch, then the impact matches that of the Shared Repository pattern. If more synchronization is needed, then the impact is that of Layers.

c. Checkpoint/Rollback: (Some components): Layers: ++, SR: ++. Therefore, the impact should be ++. Analysis: For layers, see above. Data in a database is often checkpointed for recovery, which is a natural fit.

4. Fault Prevention

a. Removal from Service: (Some components) Layers: ~, SR: ~, Therefore, impact should be ~. Analysis: Nothing in either pattern either contributes to or hinders implementation of this tactic.

b. Transactions: Similar to checkpoint/rollback, above.

**Pipes and Filters-Blackboard** (the third most common pair we saw)

1. Tactics for Fault Detection:
   a. Ping-Echo: (All components plus control) P&F: --, Blackboard: ~. Therefore, overall score should be - - or better. Analysis: Adding a Blackboard to a P&F architecture does not relieve all the P&F components from making significant changes. Could it be made easier if the Blackboard becomes the ping controller? Yes, but that is going outside the usual role of the Blackboard component.

   b. Heartbeat: (All components plus control) P&F: --, Blackboard: ~. Same rationale as for Ping-Echo

   c. Exceptions: (All components) P&F: --, Blackboard: ~. Therefore, overall score should be --. Analysis: whatever needs to change with the P&F components must also change in the same manner with the addition of Blackboard.

2. Fault Recovery – Preparation and Repair
   a. Voting: Voting (Some components) P&F: ++, Blackboard: +. Therefore, the overall score should be up to ++. Analysis: The logical way to do this seems to be to have the voting components be different filters, and the Blackboard be the deciding component. That is an extremely nice fit, and would require little work for either. Of course, that assumes you divide your application appropriately, but that should be doable. It depends on what must be voted upon, although it is likely that it involves calculations associated with the blackboard component.

   b. Active Redundancy: (Some components) P&F: ++, Blackboard: +. Therefore, the overall score should be up to ++. Analysis: The situation is similar to voting, and the same arguments can be made.

   c. Passive Redundancy: (Some components) P&F: -, Blackboard: ~. Therefore, the overall score should be up to ~. Analysis: The blackboard might be the arbitrator among redundant components, but the components still might have to change. The most appropriate approach is likely determined by the context of the application.

   d. Spare: (Some components) P&F: +, Blackboard: ~. Therefore, the overall score should be up to +. Analysis: Same arguments as for voting.
3. Recovery – Reintroduction of a failed component

a. Shadow: (Some components): P&F: +, Blackboard: -. Therefore, the impact should be up to +. Analysis: Actually, it is difficult to imagine this tactic within a P&F-Blackboard architecture. You would probably shadow Filters, and then the Blackboard would do some sort of arbitration. The Filters provide the plumbing already, so the architecture should be reasonably compatible with this tactic. Thus, the overall impact would be somewhat positive; +.

b. State Resynchronization: (Some components): P&F: --, Blackboard: +. Therefore, impact should be up to +. Can you get away with only synchronizing the state in the Blackboard? If so, the impact is positive. Otherwise, we have to examine and keep the state of all components, including the P&F components. And that becomes a --.

c. Checkpoint/Rollback: (Some components): P&F: --, Blackboard: --. Therefore, the impact should be --. Analysis: Neither pattern is well adapted for rolling back. Putting them together won’t help things.

4. Fault Prevention

a. Removal from Service: (Some components) P&F: -, Blackboard: -. Therefore, impact should be -. Analysis: Same issues as for the individual patterns; they haven’t gone away.

b. Transactions: P&F: -, Blackboard: -- (Some components), therefore the impact should be between - and --. Analysis: It depends on where transactions are needed. If one has this architecture, transactions would be more likely to be handled by the Pipes and Filters part of the system, but it depends on the particular application.


Client-Server -- PAC (the fourth most common pair we saw)

1. Tactics for Fault Detection:

a. Ping-Echo: (All components plus controller) CS: +, PAC: ~. Therefore, overall score should be ~ or better. Analysis: All components must be notified and respond. CS components have natural communication built in. The PAC display components can easily respond, but it’s not their primary job, so some response code must be added to them. This should not be difficult, so the overall score is probably +.
b. Heartbeat: (All components plus controller) CS: +, PAC: ~. Same rationale as for Ping-Echo

c. Exceptions: (All components) CS: ~, PAC: ~. Therefore, overall score should be ~. Analysis: It isn’t a natural fit for any components in either pattern to throw and catch exceptions. But it would not require horribly disruptive changes. Putting the two patterns together doesn’t change anything.

2. Fault Recovery – Preparation and Repair

a. Voting: Voting (Some components) CS: +, PAC: +. Therefore, the overall score should be +. Analysis: The client-server architecture provides a nice common point for collection and interpretation of different voters’ results. On the PAC side, the voting probably happens under the level of the abstraction, meaning few changes to this pattern.

b. Active Redundancy: (Some components) CS: +, PAC: +. Therefore, the overall score should be +. Analysis: Reasonable support in CS for active redundancy. Active redundancy will probably be implemented below the Abstraction component of the PAC pattern, resulting in only minor changes. The requirements will dictate where redundancy must be implemented, but either place is good.

c. Passive Redundancy: (Some components) CS: +, PAC: -. Therefore, the overall score should be up to +. Analysis: This tactic is by nature more involved than Active Redundancy, so one can expect it is more difficult to implement. However, it does not have a greater impact on the CS pattern. On the PAC side, the tactic should be able to be implemented below the Abstraction level, because the work goes over in the CS components. This is more difficult, but it is more likely to be implemented in the CS pattern than the PAC pattern.

d. Spare: (Some components) CS: +, PAC: ~. Therefore, the overall score should be +. Analysis: Same arguments as for voting.

3. Recovery – Reintroduction of a failed component

a. Shadow: (Some components): CS: +, PAC: +. Therefore, the impact should be +. Analysis: Shadowing is well supported by either pattern.

b. State Resynchronization: (Some components): CS: +, PAC: -. Therefore, impact should be up to +. Analysis: The issue with PAC is that presentation agents must have their states resynchronized. The problem
doesn’t go away with the introduction of the CS pattern; the agents still need to handle the state resynchronization. But the key issue is where which state needs to be resynchronized. It may well be underneath the PAC – inside the abstraction. In that case, it will be quite positive. So it ultimately depends on the particular application.

c. Checkpoint/Rollback: (Some components): CS: ++, PAC: -. Therefore, the impact should be between - and ++. Analysis: CS has natural support, but synchronization of presentations in the PAC pattern is a bit problematic. The overall effort depends on whether the presentations must be resynchronized, which would depend on how often the interface must be updated – a requirements issue.

4. Fault Prevention

a. Removal from Service: (Some components) CS: ~, PAC: ~, Therefore, impact should be ~. Analysis: CS would be a good fit except it needs a controlling component; see Broker. PAC doesn’t really support this tactic, nor does it hinder it. So regardless of where this tactic is implemented, the effort is moderate ~.

b. Transactions: CS: ++, PAC: + (Some components), therefore the impact should be between + and ++. Analysis: CS supports transactions, but they are generally created in the PAC components. This is likely to lean closer to the impact on PAC than on CS.


Layers -- PAC (the fifth most common pair we saw)

1. Tactics for Fault Detection:

a. Ping-Echo: (All components plus control) Layers: +, PAC: ~. Therefore, overall score should be ~ or better. Analysis: All components must be notified and respond. Layers provide some support for this type of communication, but what about the components for the interface? They must also respond to pings. The PAC display components can respond, which will make the impact better, but it’s not their primary job.

b. Heartbeat: (All components plus control) Layers: +, PAC: ~. Same rationale as for Ping-Echo

c. Exceptions: (All components) Layers: ++, PAC: ~. Therefore, overall score should be ~. Analysis: Layers’ hierarchical organization provides a natural
path for exception handling and reporting. But the PAC components, particularly the P and A are likely not in the Layers’ hierarchy. So there must be a way implemented for them to respond. This is therefore no better than the single pattern.)

2. Fault Recovery – Preparation and Repair
   
a. Voting: Voting (Some components) Layers: +, PAC: +. Therefore, the overall score should be +. Analysis: Layers provides a nice hierarchy for handling voting, and it probably happens all under the PAC components, not affecting them.

b. Active Redundancy: (Some components) Layers: +, PAC: +. Therefore, the overall score should be +. Analysis: Same rationale as for voting: the redundancy can be supported by Layers, and PAC can stay out of the picture.

c. Passive Redundancy: (Some components) Layers: +, PAC: -. Therefore, the overall score should be up to +. Analysis: Same as Active redundancy.

d. Spare: (Some components) Layers: ~, PAC: ~. Therefore, the overall score should be ~. Analysis: Same arguments as for voting, although it isn’t quite as easy in Layers as voting is. So the overall score is neutral.

3. Recovery – Reintroduction of a failed component
   
a. Shadow: (Some components): Layers: +, PAC: +. Therefore, the impact should be +. Analysis: Shadowing is well supported by Layers, and would be implemented in the abstraction component of PAC.

b. State Resynchronization: (Some components): Layers: +, PAC: -. Therefore, impact should be up to +. Analysis: It really depends on what needs to be resynchronized. It is likely to be within the layers, making the impact closer to +, but it depends on the application’s requirements.

c. Checkpoint/Rollback: (Some components): Layers: ++, PAC: -. Therefore, the impact should be from – to ++. Analysis: Similar to the Broker-PAC combination discussed above. If the rollback does not need to be reflected in the user presentation, then the transaction is likely able to be implemented wholly within the Layers, which is a very good match. Otherwise, implementation becomes somewhat troublesome, as all presentations must be updated and synchronized.

4. Fault Prevention
a. Removal from Service: (Some components) Layers: \(\sim\), PAC: \(\sim\). Therefore, impact should be \(\sim\). Analysis: Neither pattern gives any support for this tactic, but they don't hinder the implementation, either.

b. Transactions: Layers: ++, PAC: + (Some components), therefore the impact should be + to ++. Analysis: Layers basically already supports a transaction-based model. PAC is more removed, but some changes may be required to get the interface components (the presentation component in PAC) to operate in a transaction mode. Overall, it is expected to be relatively easy.


**Layers -- MVC** (the sixth most common pair we saw)

1. Tactics for Fault Detection:

a. Ping-Echo: (All components plus control) Layers: +, MVC: \(\sim\). Therefore, overall score should be \(\sim\) or better. Analysis: All components must be notified and respond. Layers provide some support for this type of communication, but what about the components for the interface? They must also respond to pings. The MVC display components can respond, but it's not their primary job.

b. Heartbeat: (All components plus control) Layers: +, MVC: \(\sim\). Same rationale as for Ping-Echo

c. Exceptions: (All components) Layers: ++, MVC: \(\sim\). Therefore, overall score should be \(\sim\). Analysis: Layers' hierarchical organization provides a natural path for exception handling and reporting. But the MVC components, particularly the M and C are likely not in the Layers’ hierarchy. So there must be a way implemented for them to respond. Note that MVC is more tightly coupled than PAC, but shouldn’t affect the implementation of several tactics, such as this one.

2. Fault Recovery – Preparation and Repair

a. Voting: Voting (Some components) Layers: +, MVC: +. Therefore, the overall score should be +. Analysis: Layers provides a nice hierarchy for handling voting, and it probably happens all within the Model component, not affecting the others.

b. Active Redundancy: (Some components) Layers: +, MVC: +. Therefore, the overall score should be +. Analysis: Same rationale as for voting.
c. Passive Redundancy: (Some components) Layers: +, MVC: -. Therefore, the overall score should be up to +. Analysis: Same as Active redundancy, except the MVC doesn’t have to worry about keeping the replicated component updated; Layers can do it. It is likely that the replicated components are in the Layers, rather than in the MVC pattern, which means that the impact is likely to be +.

d. Spare: (Some components) Layers: ~, MVC: -. Therefore, the overall score should be up to ~. Analysis: Same arguments as for voting, although it isn’t quite as easy in Layers as voting is. So the overall score is likely to be neutral.

3. Recovery – Reintroduction of a failed component

   a. Shadow: (Some components): Layers: +, MVC: -. Therefore, the impact should be up to +. Analysis: Shadowing is well supported by Layers, and is likely to not involve the user interface (M and C components). So the impact is more likely to be closer to + than ~.

   b. State Resynchronization: (Some components): Layers: +, MVC: ~. Therefore, impact should be up to +. Analysis: If it can all happen in the Layers, the impact will be low. But some applications may be such that the Model, and the View have to be synchronized as well. It will depend on the application’s requirements.

   c. Checkpoint/Rollback: (Some components): Layers: ++, MVC: +. Therefore, the impact should be between + and ++. Analysis: Checkpointing and rollback can be implemented in the Model. Layers has very good support for checkpointing, but is it enough that the whole implementation becomes ++? It likely depends on what data or processing must be reliably checkpointed.

4. Fault Prevention

   a. Removal from Service: (Some components) Layers: ~, MVC: ~, Therefore, impact should be ~. Analysis: Neither pattern gives any support for this tactic, but they don’t hinder the implementation, either.

   b. Transactions: Layers: ++, MVC: ~ (Some components), therefore the impact should be from ~ to ++. Analysis: Layers basically already supports a transaction-based model. MVC provides neither support nor distraction, but will have to be involved, so it isn’t just letting Layers handle it.

Appendix C, From Chapter 8

The following is a report from an actual PBAR session

System Description and Requirements
A local company provides discount magazine subscriptions to its customers, and has found a rather lucrative niche market for their business. Inmates in prison have a lot of time on their hands, and appreciate magazines. So this business mainly targets prisoners.

The main part of the business consists of receiving orders, and then placing the orders with the magazine companies. The service they provide is one-stop shopping for magazine subscriptions at a very good price. (I assume they get good volume prices from the magazines themselves.)

At this time, all their orders are received on paper through the mail. This system allows employees to enter the customer information and the order information from the paper orders into a database, where it can be used later.

There are several key requirements:

1. Customers may have multiple addresses, and the addresses change often. It is important to keep former addresses around (at least one former address), because customers sometimes move, and their magazines don’t catch up to them. So they complain to this company.

2. In the future, customers will be able to be entered directly from the customer website.

3. In the future, orders might be taken via the web.

4. One order may contain orders for several different magazines.

5. Orders arrive with payments, which can be cash, check, or credit card. Payments may be of two or more types (e.g. two different checks).

6. Customer complaints are common, and the company spends considerable time and money resolving them. Besides the forwarding address problem noted above, a common complaint is that an order is placed, and the subscription never starts. So tracking order information and status is desirable. (Note that the original vision of the system included support for handling customer problems, but that has been deferred to a later release.)
7. Data entry errors are a fact of life. In the future, the system should support a double-entry system, where two different employees enter the same information, and it is compared for accuracy.

8. Data entry employees may work remotely.

Key Quality Attribute Requirements
1. Reliability (accuracy and fault tolerance): The consequences of inaccurate data are serious: Customers get angry because they don’t get the magazines they paid for. And it costs time and trouble to hunt down the error, even to the point of finding the original paper order and verifying it. This is a key quality attribute.

2. Capacity: The amount of data being handled is quite large for a system of this size. There are currently around 90 thousand customers, and roughly one million orders. There are around 1000 different magazine titles. However, because each magazine can come from multiple suppliers, the real number of magazines is three to four times that number. The system must handle not only this load, but larger amounts of data as the business expands. This is a key quality attribute.

3. Extensibility: There are numerous important capabilities that will be needed in the future, so the system must be easily extended. This is a key quality attribute.

4. Security (authentication and authorization): It is important to keep the system safe from unauthorized use. In particular, data entry employees must be authenticated, and may have different privileges. This quality attribute is important.

5. Performance: The system has no hard or soft real time needs, but operations on this large database should be quick.

System Architecture
This system is a student team project, and as such is rather small, and has work left to be done. The architecture relies heavily on existing software packages. These packages, and the way they are designed, play a major role in shaping the architecture. Prominent patterns include the following:

1. Repository: The database is central to this application.

2. MVC: MVC is really central to the human interface of the application. The framework they used for implementing the UI is based on MVC, and even identifies its components as such. However, see the note in Layers, below
3. Layers: The system is also highly layered. The Views, Controllers, and Model form (almost) the top three layers, which is really not a true use of the MVC pattern. Underneath that, there is another layer associated with the Model layer, and then the database.

There is another layer that is sometimes present on top of (or in) some of the Views. It is called “widgets.” Its purpose is to go nearly directly to the database for certain actions (e.g., large queries) to improve performance. It bypasses intermediate layers.

4. Broker: There is some possibility for a Broker to manage efficient and reliable access to the database. However, at this time, it is simply potential.

**Patterns and Quality Attributes**

In general, the patterns are compatible with the important quality attributes of the system. Of note:

1. (Performance) The most notable liability of the Layers pattern is its performance. Indeed, this architecture uses the common “wormhole” variation of Layers to circumvent some layers in order to improve performance. However, this variant has some issues:
   a. It’s a bit harder to understand.
   b. It’s a bit more problematic for extensibility, in that as new capabilities are added, one must always consider whether or not to use the “wormhole” in the new features. This will almost certainly be an issue in this application. That said, the “wormhole” appears to be worth the tradeoff.
   c. In this case, it required a bit of data to be replicated.

2. (Security) The Layers pattern is so compatible with authentication and authorization that they are normally a slam-dunk in a Layered architecture. Unfortunately, it isn’t quite so easy here. The “wormhole” causes security checks to be done in two different places. So it isn’t quite as clean as we might like. It doesn’t appear to be a big problem, and is likely not worth doing differently.

3. (Extensibility) MVC will make it easy to add future web capabilities, such as adding customers or orders. This is a key strength of this architecture.

4. (Capacity) The use of a commercial database package takes care of any capacity worries. In addition, one of the commercial layers can easily become a Broker, thus adding further support for capacity, if needed. No concerns.
5. (Reliability) The commercial database package also takes care of most reliability concerns. It provides the ability to group actions into transactions which can be committed as a unit. We discussed the possibility of race conditions that might allow a record to be lost and appear to the employee as if it had been entered. Upon further reflection, I suspect that the easiest way to deal with this (if it even it possible) is simply to implement the double-entry system that will be put in place for catching erroneous data entry. That should take care of it. We also discussed running audits on the data to ensure that customer and order records are not missing anything. Given the importance of correct data, it still might be a good idea.

Summary Notes
1. There was no architecture documentation on the project, yet the review still worked. The architect was very knowledgeable.

2. Overall assessment: Successful: Few issues were raised, but that was because the system appears to be well-designed.

3. Statistical assessment:
   a. Time: 1 hour, 15 minutes
   b. Size: small (I didn’t get a size estimate. It’s bigger than the first one I did.)
   c. Team size: 4 people (students)
   d. Domain experience: 1 person – high, 3 people – moderate by this time
   e. Project phase: mid implementation, first release.
   f. Patterns found: 3 (Broker doesn’t count, because it is not present)
   g. Issues found: 3
      i. Reliability; in particular issues about potential loss of data in rare conditions. (Discussed during first part)
      ii. Extensibility, with respect to the Layers “wormhole” (discussed during the second part – strong pattern discussion) (Major Issue)
      iii. Security, with respect to the Layers “wormhole” (discussed during the second part)
   h. Pattern awareness: The team knew all the patterns we discussed.