Problem Frame Patterns

An Exploration of Patterns in the Problem Space

Rebecca Wirfs-Brock, Paul Taylor, James Noble

Contents

1 **Problems and Solutions** 2
1.1 Introducing problem frames 2
1.2 Examples of problem frames 3
1.3 Using problem frames 4
1.4 Our approach 5
2 **Problem Frame Patterns** 6
2.1 Required Behavior Problem Frame 8
2.2 Commanded Behavior Problem Frame 13
2.3 Information Display Problem Frame 18
2.4 Simple Workpieces Problem Frame 23
2.5 Transformation Problem Frame 26
3 **Assessment and Conclusions** 30
4 **Glossary** 32
5 **Acknowledgments** 34
6 **References and Resources** 34

1 Problems and Solutions

Software developers solve problems in code. It's part of our nature to decompose, resolve, or drive toward a solution quickly and efficiently. We naturally gravitate to 'the solution space' where our architectures, designs, patterns and idioms combine to resolve the plaguing problems that our clients continually push on us. Patterns have in part been so successful because they expedite the journey from problem to solution—they make us look good by handing us a best-practice template that we can fill out to deliver a proven solution. So what good can come out of immersing ourselves in the problem space?

Patterns work like a ladder in the 'Snakes and Ladders' board game—given a known context and problem (square on the board) they give us a leg-up to a higher place. Design patterns fall squarely in the middle of the solution space and provide object-oriented fragments of structure to resolve solution space forces [1]. But they do assume that the problem and context are sufficiently well understood so that a sensible selection of the appropriate pattern can be made. So what if we don't yet have this orientation? What if we find ourselves washing around in the amorphous problem space, unable to get a foothold on anything to bear the weight of a pattern or to anchor a fragment of architecture? Is there another kind of pattern that helps to locate our thinking early in the analysis and conceptualization of systems and solutions? Do *patterns* in the problem space exist? If so, what kinds of patterns are they? How do they relate to design patterns? And how might consideration of problem structure help us produce better software architecture and design?

1.1 Introducing problem frames

In this paper, we work with 'problem frames', a problem space classification mechanism proposed by Michael Jackson [1] and further refined in [2]. Jackson's 'problem frames' are interesting because they build on a recognition of generic problem types, based on structures and relationships between domains and system elements. Problem frames are based on a philosophy of phenomenology, which firmly places us in a world of concepts, domains, phenomena and (software) machines—software mechanisms of our own design—that interact with these elements of the problem's enveloping context.

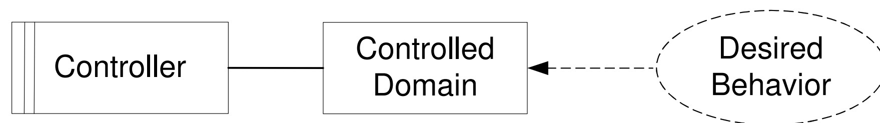
In Jackson's problem frames, a problem is described as consisting of the software machine and one or more application domains. The machine and application domain are connected, representing a domain of some shared phenomena in which both the machine and the application domain participate. The problem context provides us with the elements of a scene, but not the function. For example, a context may include a workbench, a box of hand tools, and some pieces of timber. What we are missing is the requirement that will dictate *but not describe* the function of the machine. Jackson suggests that a requirement should be expressed in terms of the *context* rather than in terms of the *machine*. One possible requirement in this example context is to produce a wooden container with removable lid to hold pens and pencils, while another may be to transform the timber into a big pile of wood shavings for combustion. On the other hand, requirements in terms of the machine in this example could be a series of detailed steps that tell you how to hand-saw and plane the constituent wood pieces, and how to use nails and glue to best hold the box together. Or how to shave the wood with a handheld plane until there is no solid wood left. By expressing a requirement only in terms of the machine and not in contextual terms, we risk jumping to the solution space prematurely. As a consequence, we risk missing important aspects of specification and opportunities

for use and reuse outside the existing experience or existing processes or machines. Continuing the example, when the craftsperson ignores the workshop, the possibility of using the high-powered wood-chipping machine sitting in the corner might get overlooked.

1.2 Examples of problem frames

A problem frame is a generic, abstract problem structure, proposed by Jackson using the problem solving techniques of Polya [3]. A problem frame consists of 'principal parts', a structure, and a solution task. Figure 1 illustrates two of Jackson's problem frames—the Required Behavior Frame and the Simple Workpieces Frame. The Required Behavior Frame deals with a simple problem class so let's start there. This frame is a simple generalization of the structure of a class of problems that involve automated control—an example is an electronic thermostat for temperature control. The frame consists of three principal parts. The *machine* (the component to be built) is shown as a double-hatched box. The machine is associated with a single *domain* (the *Controlled Domain* depicted by a rectangle) by a line, representing an interface of shared phenomena. The domain is in turn connected by a dashed line to a named set of *properties* (the *Desired Behavior* requirement depicted by a dashed ellipse).

Required Behavior Frame



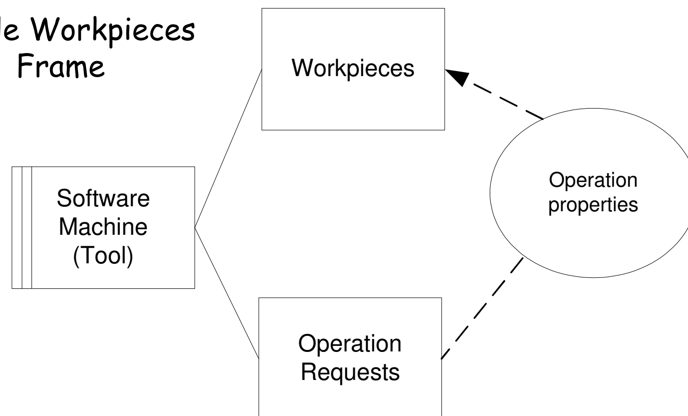


Figure 1: Two of Jackson's problem frames.

The Simple Workpieces Frame deals with a class of problems where a user interacts with a software tool to create and manipulate computer-processable text or graphic objects, or similar structures.

Although abstract, these problem frames generalize one of a number of basic structures that underlie every system and architecture that solve problems *of their type*. Recognizing the structure of the *problem* and

adopting a frame such as this helps to structure the specification and analysis, as each element of the frame is specified and documented. Each fact (predicate, assertion, invariant, observation, classification, relationship and behavior) that will be addressed at some stage during specification, analysis, design or implementation, now has a principle part in the problem frame with which to be associated. A further benefit is that all extraneous elements of the problem can be recognized for what they are earlier in the analysis and development process.

In earlier work [1] Jackson named the Required Behavior Frame a 'Control Frame' and the Simple Workpieces Frame a 'Workpieces Frame'. In addition to the frames we have briefly mentioned, we present three other problem frames in pattern form—the Commanded Behavior Problem Frame, the Transformation Problem Frame, and the Information Display Frame. Work is progressing amongst problem frame adherents toward elaborating these frames and identifying new ones [3,7,8,9].

1.3 Using problem frames

Using a problem frame involves selecting a candidate frame from a catalog of problem frames and identifying a mapping between elements of the specific problem with the principal parts of the selected problem frame. Practically, a real world problem frequently maps to several or many problem frames, so a simple best-fit analysis is needed to select the most appropriate frame or frames. Generally, analysts decompose a complex problem into a number of smaller problems, and then focusing on the requirements and the concerns of each sub-problem. As you progress through the process of fitting one or more problem frames to the problem at hand, the frames guide you in what to specify and what questions to ask. In effect, each problem frame comes with its own micro-method in the form of a descriptive template to be completed. But to get value from using frames you do not have to 'go formal'. In practice, the frame helps you to know what questions to ask and what issues are commonly encountered in particular problem frames. Once you have framed a problem, you can start asking questions. Or conversely, as you are asking questions you find yourself exploring what frames seem to fit and push harder to gather appropriate requirements. In this early analysis period, we find ourselves working in both directions at the same time—finding a frame that fits and executing its associated micro-method to evaluate the fit occur simultaneously.

Each problem frame also describes a *frame concern*. The frame concern, illustrated in Figure 2, characterizes the domains making up a frame and describes how they must be interrelated (that is, how the operation of the machine must interact with the various domains) to produce a stylized argument that an eventual implementation will be correct. As well as helping you convince yourself your requirements are a correct analysis of the problem you are studying, frame concerns can be useful to check that you have characterized your problem with the correct frame. If you've selected the wrong frame for a particular sub-problem, the frame will suggest descriptions that don't make sense and will leave out other necessary ones—if you're fitting your problem into the wrong frame, it will be difficult to construct a convincing argument that your specification can meet that frame's concern.

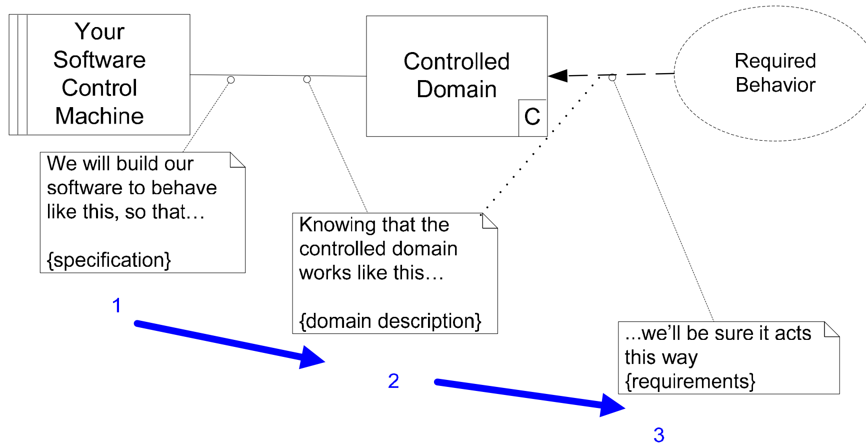


Figure 2: The Required Behavior Frame Concern

By helping you to ask the right questions, frames improve specification quality. There is another benefit—frames encourage you to separate the concerns of the problem into demarcated sub-spaces (domains or problem parts) in the overall problem space, and then to treat each in turn according to its specific needs. This form of 'separation of concerns' helps you to develop minimal and contextual descriptions, and ensures that the machine you specify sits properly in its real-world context. This can result in well-integrated and minimal software solutions that are more likely to deliver quality—fitness for purpose—because you have understood the purpose better.

1.4 Our approach

Consistent with patterns community ethos, we make no claims on originality for much of the material presented in this paper, other than the idea of bringing problem frames and patterns together, the identification of implementation issues discussed for each pattern, the running example, and our slight shift away from Jackson's insistence on real world phenomena and domains to one that accommodates computers and computational domains. The problem frames and their definitions are taken nearly verbatim from Jackson's books. The fitting of each frame into a pattern template, and the assessments and conclusions are all our own work.

We welcome engagement from anyone interested in developing the idea and look to the patterns community for feedback and stimulating discussion.

2 Problem Frame Patterns

In this section we present five patterns, one for each of Jackson's problem frames. The names of the five problem frame patterns, and the classes of problem that each address, are as follows:

- **Required Behavior Problem Frame pattern**—there is some part of the world whose behavior is to be controlled so that it satisfies certain conditions… the problem is to build a machine that will impose that control.
- **Commanded Behavior Problem Frame pattern**—there is some part of the world whose behavior is to be controlled in accordance with commands issued by an operator… the problem is to build a machine that will accept the operator's commands and impose the control accordingly.
- **Simple Workpieces Problem Frame pattern**—a tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analyzed or used in other ways… the problem is to build a machine that can act as this tool.
- **Information Display Problem Frame pattern**—there is some part of the world about whose states and behavior certain information is needed… the problem is to build a machine that will obtain this information and present it at the required place in the required form.
- **Transformation Problem Frame pattern**—there are given data which must be transformed to give certain required output. The output data must be in a particular format and it must be derived from the input data according to certain rules… the problem is to build a machine that will produce the required outputs from the inputs.

Each problem frame is presented using a simple pattern form. The pattern begins with a short definition of the problem the frame addresses, taken verbatim from Jackson's corresponding frame definition. Then our patterns present a brief example, showing the various domains that comprise the structure of the problem. The pattern examples are drawn from a running example that depicts the specification of an email client. This client can exchange messages with email servers, detect junk messages, allow users to compose new messages, and display encoded multimedia objects. A client with this amount of functionality necessitates the use of a number of different problem frames, and the process of fitting, then combining frames together illustrates how this kind of analysis can yield simple yet highly definitive and formal descriptions.

Returning to the problem frame pattern descriptions, each pattern then describes which abstract problem frame the example can be fitted to, shows the generalized structure of that frame, and describes the 'participants' (or principal parts)—that is, each domain that is a part of the frame. Each pattern next describes the abstract 'frame concern', that is, the overall condition the machine must satisfy if it is to embody a correct solution meeting the requirements of the frame. In some ways, this is similar to the 'collaboration' section of an object-oriented design pattern, in that it shows how different domains are interrelated within the frame. The pattern shows how the frame resolves the email client example, sketching an argument to show how the example's specific concerns can be resolved. Each problem frame

pattern includes a brief list of analysis, design, or implementation considerations that often arise with this frame, and (for a few of the frames) briefly lists common variants or closely related problem frames.

Domain Diagrams:

Both specific problems and abstract frames are drawn using Jackson's most recent [2] problem frame diagram notation:

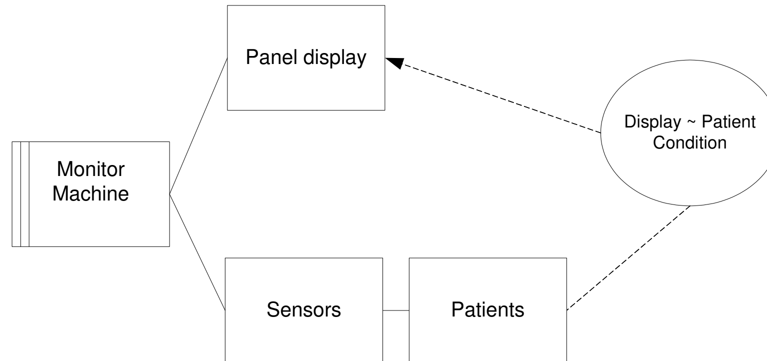


Figure 3: An example of Jackson's problem frame notation (for a medical monitoring machine).

These diagrams show *domains* (rectangles) and *requirements* (dashed ellipses). Domains generally represent sets of phenomena in the real world—in the example shown in Figure 3, medical patients, sensors, display panels, and a 'monitor machine' that connects sensors to panels. An important point is that the *solution*—in problem frame terminology, the *Machine*—is considered a domain like any other. This emphasizes a practical consequence of Jackson's phenomenological stance—the 'machine' that we have to build is a domain in the real world (at least once the software 'machine' is built and installed) and can be treated from a specification perspective just like any other domain, with characteristics and phenomena of its own. Since we have to build it, we identify it with stripes on the left-hand side of its rectangle.

Domains are linked by lines representing *shared phenomena* between them; that is, phenomena that occur in each domain. In Figure 3's example, sensors are physically attached to patients and monitor their pulse, blood oxygen levels, blood pressure, etc. Sensors are similarly attached to the monitor machine via an instrumentation bus, and the machine is attached to a display panel via some graphics drivers. *Requirements* are constraints on the states or operations of various domains. In Figure 3's case, they are linked (by dashed lines) to the domains they constrain. The requirement that the patient's state of health must be reflected accurately in the Panel display is expressed as an assertion in the Patients domain, but acts as a *specification* for the Panel Display domain (thus the arrowhead).

2.1 Required Behavior Problem Frame

2.1.1 Problem

There is some part of the world whose behavior is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.

2.1.2 Example

The most basic requirement for an email client (the most fundamental problem that it has to solve) is to send emails from the client program to some external Mail Server, and to get emails back from that server. The problem frame diagram in Figure 4 illustrates how the Required Behavior problem frame fits this simple description of the client's most basic function. The Machine (called the Email Client) must interact with the immediate endpoint of the emails (the Mail Server) and that interaction must satisfy the requirement that emails are correctly exchanged between the two.

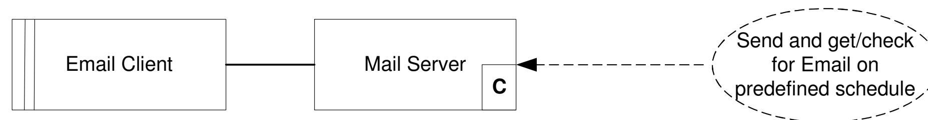


Figure 4: Basic email client operation mapped onto the Required Behavior problem

The Mail Server domain will consist primarily of entities representing email messages. The Mail Server domain is linked to the Email Client domain by shared phenomena—obviously email messages, but also events by which the Email Client can send or request emails to and from the server. This is two-way communication, since the Email Client can inspect the Mail Server domain (i.e. to find any emails) and affect it (by sending emails to the server). Since the key problem the frame presents is to design the Email Client, *the problem gives no more details about that client directly*—because those details are precisely the space we will fill in our design and implementation: our specification says nothing more about the machine. There are three basic Required Behaviors: the Email Client periodically issues Send events to send emails, it initiates Check events to determine if any new emails have arrived, and it issues Get events to retrieve each of those emails.

The "C" in the lower corner of the Mail Server domain indicates that it is a causal domain, one which predictably responds to events. Note that although the server is internally aware of the arrival of new email messages, it never signals out to Client when one arrives—rather, it waits for the Client to initiate a Check event. Because of this, it is always responsive as far as the Client is concerned.

2.1.3 Structure

This problem fits into the **Required Behavior** problem frame, shown in Figure 5.

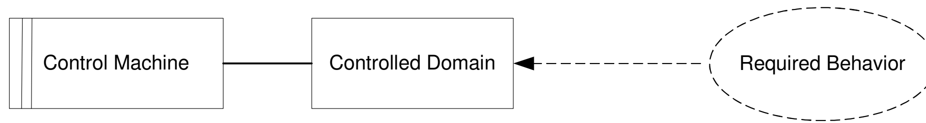


Figure 5: General form of the Required Behavior problem frame.

The Required Behavior problem frame is comprised of these participating elements:

- Control Machine (Email Client in the example)—this is the part that we know we have to build, and its purpose is to exert control on the Controlled Domain.
- Controlled Domain (Mail Server in the example)—this domain defines just the part of the world that needs to be 'controlled' by the machine.
- Required Behavior (Send and Get Emails in the example)—this part describes how the domain must be controlled by the machine.

2.1.4 Frame Concern

They key concern of the Required Behavior problem frame is that the machine must ensure that the Controlled Domain exhibits the required behavior.

The frame concern relates the frame's domains in the following way:

1. The behavior of the Control Machine
2. AND the properties of the Controlled Domain
3. ENSURE the Required Behavior.

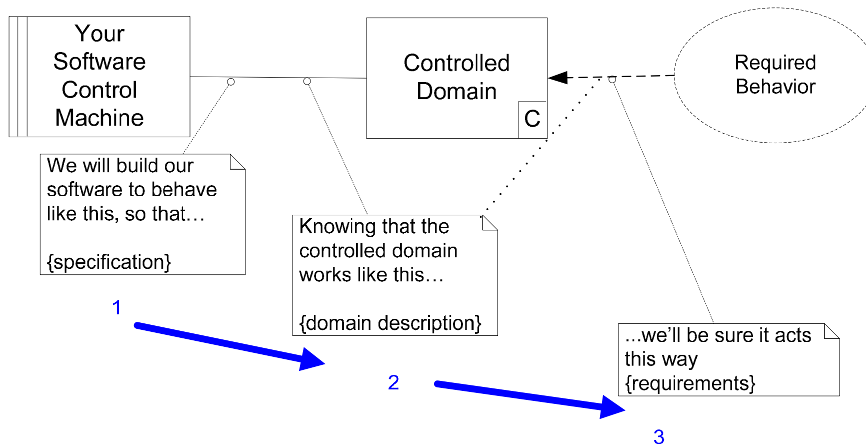


Figure 6: The Required Behavior Frame Concern

Concern Resolved:

Referring to the example, we can say that the behavior of the Email Client AND the properties of the Mail Server ENSURE that email will be sent and received according to the predefined schedule.

That is, we must be able to make a convincing argument that:

1. The behavior of the Email Client
2. AND the properties of the Mail Server
3. ENSURE the emails will be exchanged between Client and Server.

Addressing the frame concern adequately means making sure that descriptions of the requirement, the specification of your machine at its interface to the controlled domain, and the description of the controlled domain's reactions to events all work together consistently. In the Email Client application, the requirement must describe how emails should be sent and received. The domain description must show how emails are received, stored, and transferred by the mail server, and the machine's specification must show how it behaves at its interface with the Mail Server. To deal with receiving emails, for example, the Email Client must issue Check events periodically to determine if there are any incoming emails, and then a series of Get events (one for each pending email). Assuming the Mail Server behaves as expected —listing all pending emails in response to a Check event, and moving one Email message in response to each Get event, the system as a whole will satisfy the required behaviour—that incoming emails are periodically transferred from server to client.

2.1.5 Discussion

The task of analyzing a problem that fits the Required Behavior frame is to analyze how the controlled domain works and specify the behavior your machine must have so that it exerts the proper control over the Controlled Domain. To fit a problem to this problem frame, you need to match each of the frame's parts to a portion of the problem at hand. As you proceed, you assess the quality of the fit by working with each part in turn to write down a mapping between the frame part's characteristics and the corresponding phenomena in that part of the problem space. When you do this, the frame will guide you, prompting you with questions. For example, to fit the frame's Controlled Domain part, you will have to answer the following kinds of questions—what external state in the Controlled Domain must be controlled? What are the natural states of these objects or phenomena and how do transitions come about? And which of these transitions must your Machine command? And how and when does your software machine decide what actions to initiate?

With an unreliable connection interspersed between the machine and the Controlled Domain, there is an increased probability that the state of things as "expected" in the Controlled Domain won't match up with your machine's intended effects on the Controlled Domain. If this indeed is a valid concern, then your descriptions must address how your machine will detect when things get 'out of synch' and what your machine must do in this case.

More fundamental is the question of whether or not your Software Machine needs find out whether its actions have had the intended effect. A question to ask about any Required Behavior problem is whether the machine needs to know for certain, or whether it can just react later (when the state of something in the

Controlled Domain is not as expected). A simple example from our Email Client application is the case where the user wishes to know whether sent emails have been received by the recipient. One way to accomplish this is to tag an email as requiring a 'confirmation of receipt' reply when it is read by its recipient.

2.1.6 Variants

Connection Domain:

The part of the Required Behavior problem frame that varies most across different problems is whether or not a Connection Domain is part of the problem. In an ideal world your software machine directly shares phenomena with the Controlled Domain and a rich interface gives it access to all the phenomena it needs to detect or control. If you can convince yourself that it is safe to view the connection between your software Machine and the thing under control as being direct (with no complicating connection properties that have to be specified and managed by the Machine) your software solution will be considerably simpler. On the other hand, when software system designers assume the simple case and overlook the complexities of this connection (which happens often in the desire to selectively ignore complexity) the reliability of the Machine can be dramatically reduced.

So, the reality is that often software isn't able to directly and simply affect the Controlled Domain. A Connection Domain lies between the two which interposes its own properties and behavior. If, as is often the case, you decide that this connection can cause quirky or interesting behavior then you may need to understand the properties of this Connection Domain. You'll then need to describe the properties of this domain and how it interacts with your software Machine and the Controlled Domain. In the example, a separate Connection Domain is most likely needed as part of the elaboration of the frame since the internet (a not entirely reliable connection) lies between our Email Client and the Mail Server (Figure 7).

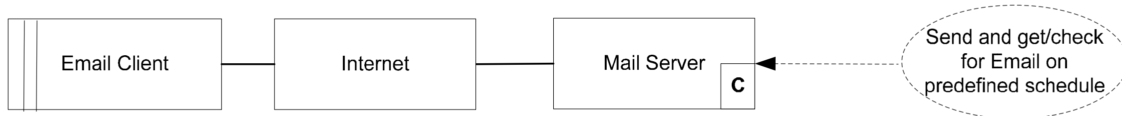


Figure 7: The Required Behavior Frame showing the internet connection domain.

Configuration Domain:

Sometimes it is useful to specify the configurable aspects that drive the controlling machine's behavior. For example, in the case of the Email Client, the schedule for when to send and check for incoming mail might be represented in a designed description domain, shown as an oval with a single vertical line in Figure 8—the Email Transfer Schedule. While this may indeed be a simple lexical description, it is interpreted by the Email Client, and its values can be set by the user in yet another problem frame (a Commanded Behavior frame). Domains in one problem frame can be represented in other frames; if this is so, then care must be taken to ensure that the requirement from one frame doesn't conflict or contradict another.

Similarly, it can be useful to model other domains that the Control Machine interfaces to as it performs its control functions. For example, in the email example, incoming and outgoing emails are transferred between the Mail Folder Domain and the Email Client. A more complete picture of the use of the Required Behavior frame for specifying the Email Client example is shown in Figure 8.

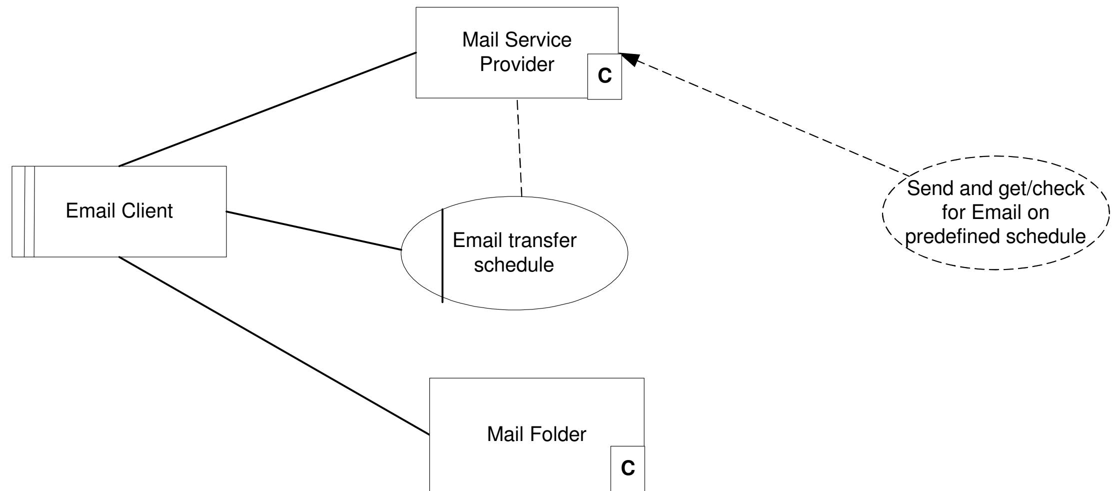


Figure 8: Required Behavior problem frame for the Email Client, with added 'Mail Folder' and 'Email transfer schedule' domains.

2.2 Commanded Behavior Problem Frame

2.2.1 Problem

There is some part of the world whose behavior is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.

2.2.2 Example

An Email Client cannot only communicate with the Mail Server: it must also provide an interface to its users to compose and send, receive, and read email The frame diagram below shows this problem: the Machine, the Email Client, must exchange emails with the Mail Server with the requirement that the interactions with the Mail Server are commanded by the client.

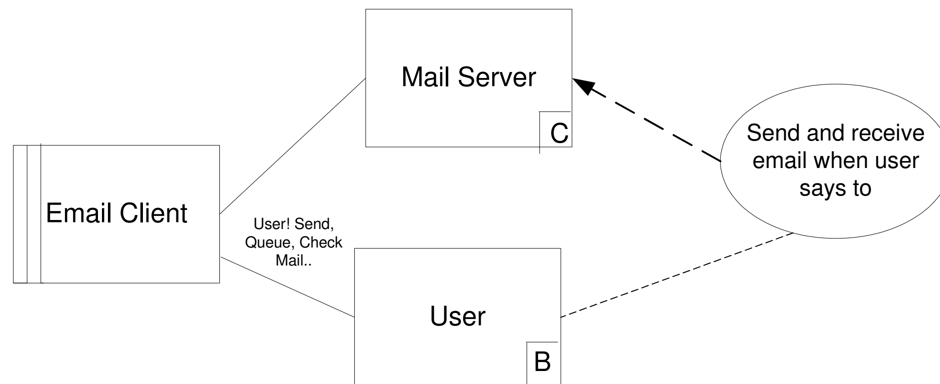


Figure 9 User's control of the email client mapped onto the Commanded Behavior problem frame

The Mail Server domain, once again, will consist primarily of entities representing email messages, and is linked to the Email Client domain by shared phenomena—Control Events—Email messages, and Send, Get, and Check events. The User domain is also a source of *events*, in this case Commands from the user to Queue (i.e. edit to send later), to Send queued emails, and to Check Mail. The Commanded Behaviors are as follows: in response to User Send commands, the Email Client must send all queued Email Messages; in response to User Check events it must check Email and use Get events as necessary to retrieve them.

As explained earlier, the "C" in the lower corner of the Mail Server domain indicates that it is a causal domain, one which predictably responds to events. The User domain, marked with a "B", is a biddable domain—one which isn't guaranteed to respond predictably. This is because it is impossible to predict how (when, and in what sequence) the user will issue events to the Email Client.

2.2.3 Structure

This problem fits into the Commanded Behavior problem frame shown in Figure 10:

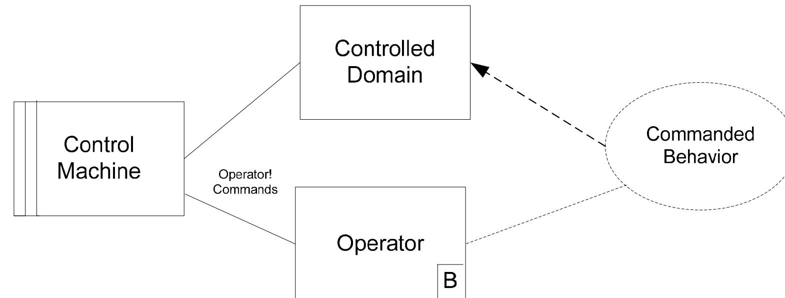


Figure 10: General form of the Commanded Behavior problem frame.

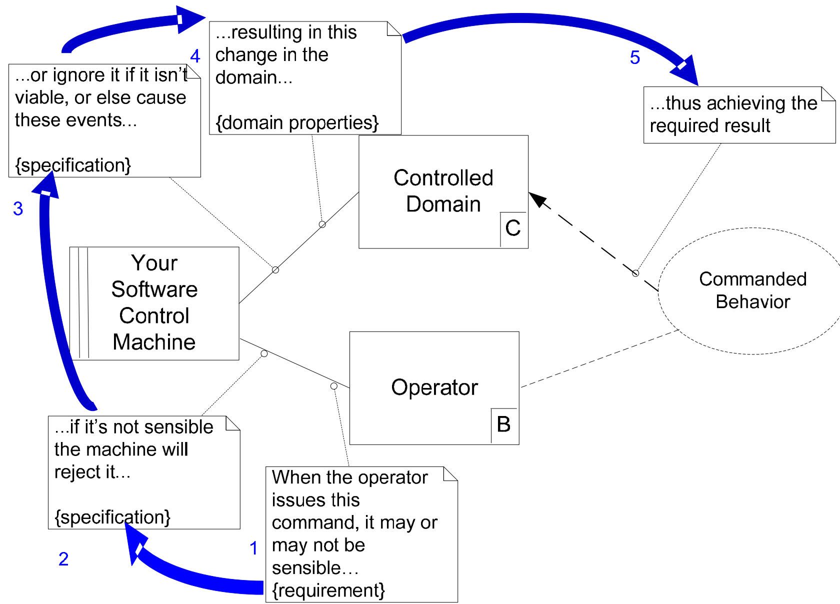
Participants:

- Control Machine (Email Client)—the part that must be built in software, it controls the Controlled Domain using Operator Commands as commanded by the Operator domain.
- Controlled Domain (Mail Server)—the part of the world to be controlled via Control Events issued by the Control Machine.
- Operator (User)—autonomously active domain that issues Commands (Operator Commands) directly to Control Machine.
- Commanded Behavior (Send and Receive Emails)—describes how the Controlled Domain must be controlled in response to user commands.

Frame Concern:

The key concern of the Commanded Behavior problem frame is that the Control Machine must produce the Commanded Behavior in the Controlled Domain in response to the Operator's commands. The frame's concern, illustrated in Figure 11, can be stated as follows:

1. When the Operator issues a Command
2. AND the Machine rejects invalid Commands
3. AND the Machine either ignores it if unviable, OR issues Control Events
4. AND the Control Events change the Controlled Domain
5. ENSURE the changed state meets the Commanded Behavior *in every case*.


...resulting in this change in the domain...
{domain properties}
4
...or ignore it if it isn't viable, or else cause these events...
{specification}
5
...thus achieving the required result
Controlled Domain
C
3
Your Software Control Machine
Commanded Behavior
Operator
B
...if it's not sensible the machine will reject it...
{specification}
When the operator issues this command, it may or may not be sensible...
{requirement}
1
2

in order to “command” the machine to do things. And although you may specify permissible actions in detail (because the Operator domain is biddable) you cannot rely on the operator following operating instructions at all times. As a result, the Control Machine can’t be required to respond to every command. Some commands may make no sense in the context of previously issued ones. Certain commands may not be viable because they are inappropriate or not permitted given the current state of the Machine or the Controlled Domain. For example, a “send email” command doesn’t make sense if there is no unsent email. This leads you to ask what commands need to be inhibited based on the current state of the Machine or the Controlled Domain. One way to inhibit “send mail” would be to disable the ‘send’ menu item when there is no unsent mail.

For more complex Commanded Behavior frames, it may be appropriate to ask if a sequence of actions makes sense, or whether a lag between issuing a command and the machine performing the action could cause the operator to mistakenly believe that a command has been ignored. In some circumstances, it is legitimate to ignore certain sequences of commands—repeated presses of the elevator ‘call’ button are a good example.

In any case, it is always appropriate to ask what should happen when a command fails. Should the operator be involved in “steering” the Control Machine through a recovery procedure? Do commands need to be reversible, logged, monitored or otherwise tracked? What kinds of feedback (if any) should the Machine give the Operator to indicate when commands have been successfully processed?

Because of the interplay between the Operator’s actions, reasons for disobeying or failing to execute commands, and the required properties of the Controlled Domain, there are more subtle relationships among the descriptions in a Commanded Behavior problem than those in a typical Required Behavior problem.

2.2.5 Variants

Designated Domain:

Is it possible that commands may be specified to the Machine that do not immediately take effect? If so, then a "Designed Domain" may need to be specified that describes commands, their parameters and when they take effect. Are there possible conflicts between Operator-issued commands and required behavior in a problem where operator commands modify or overrides required default behavior (as shown in Figure 12)? If so, a specification of frame concern *priorities* may be needed. This would simply state relative priorities (possibly in the form of rules) to resolve conflicting commands and situations.

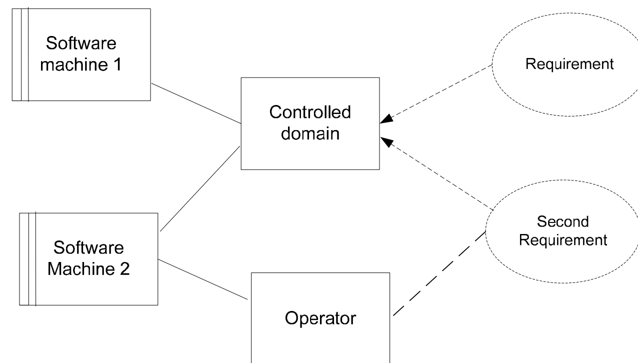


Figure 12: A composite frame with both commanded and required behaviours.

2.3 Information Display Problem Frame

2.3.1 Problem

There is some part of the world about whose states and behavior certain information is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.

2.3.2 Example

These days, email clients need to do more than allow users to edit emails, and exchange those emails with mail servers. They also need to identify the large amount of junk mails that most email users receive. The frame diagram below (Figure 13) shows this problem: the Machine (a 'Junk Mail Filter') must inspect Incoming Mail and then produce a report which assigns a junk mail rating to each email based on a Bayesian algorithm.

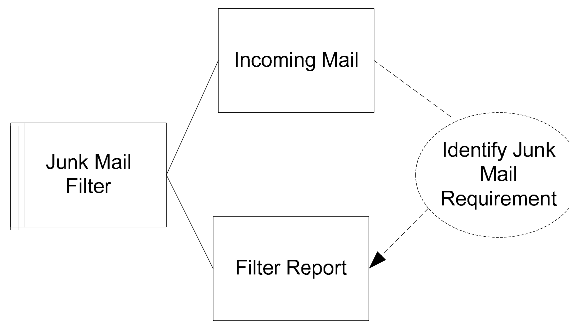


Figure 13: Junk mail detection problem mapped onto the Information Display problem frame.

2.3.3 Structure

This problem fits into the Information Display problem frame:

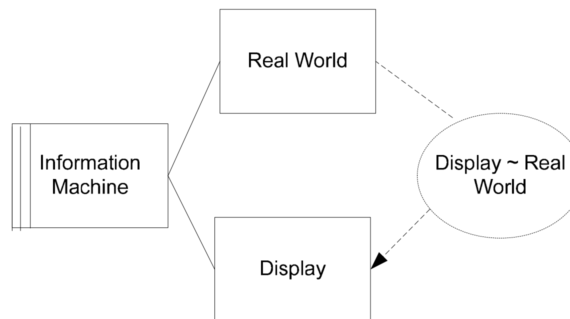


Figure 14: General form of the Information Display problem frame.

Participants:

- Information Machine (Junk Mail Filter)—the part to be built, the Information Machine displays information from the real world—something that (relative to our problem) is not under our software's control.
- Real World (Incoming Mail)—an active and autonomous domain containing the information that needs to be displayed. Nothing in the problem context can affect the Real World.
- Display (Junk Mail Report)—the part of the world where information is to be presented.
- Display~Real World (Identify Junk Mail Requirement)—the requirement that relates the domains (the Display must show true information about the Real World).

2.3.4 Frame Concern:

The key concern of the Information Display problem frame is that the Information Machine must ensure the Display's output is derived from the values in the Real World. The frame's concern, illustrated in Figure 15, can be stated as follows:

1. When the Real World is in a particular state
2. THEN because the Real World domain contains particular values
3. AND the Machine will detect those values from the Real World domain
4. AND it causes events to the Display domain
5. AND the Display domain produces some output in response to those events
6. ENSURES the Display can be interpreted as corresponding (as required) to the Real World.

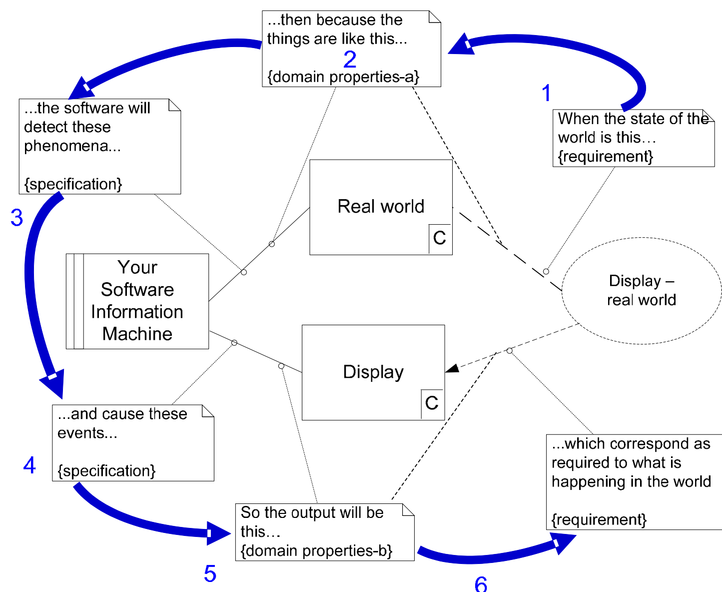


Figure 15: Information Display Frame Concern

Concern Resolved:

Referring to the example, we can say that the Information Machine always ensures that the Display responds to the state of the Real World according to the Display~Real World requirement:

1. When the user is receiving spam
2. THEN the Incoming Mail includes junk mail messages
3. AND the junk mail filter will detect those junk messages and assign each a junk mail rating value
4. AND it sends the title, junk mail rating value, and 'From:' line of those messages to the junk mail filter report
5. AND the junk mail filter report can be interpreted as listing junk messages
6. ENSURES the junk mail filter report lists the junk mail messages within the Incoming Mail stream.

2.3.5 Discussion

To address the frame concern for an Information Display problem, you describe the requirement of how information should be presented to the Display domain, the properties of the Real World domain, and the phenomena that are available at the Machine-to-Real World domain interface. In essence, you must ask, what is the form of "observation" that the Machine must make about some event, fact or thing? Indeed, it may be difficult to ascertain when an event has occurred in the Real World simply because the Real World domain-to-Machine interface is a narrow view onto the real world. For example, if your software is trying to record how many vehicles passed over sensors placed on the road it may be difficult to characterize what constitutes a vehicle—is it two axles passing within a time period? What about motorcycles, backed-up slow traffic, etc?

An Information Display problem is often characterized by a significant gap between the real world phenomena and the ability of your Machine to make an accurate interpretation of "reality" based on limited phenomena available at its interface to the Real Word domain. Although a human recipient can answer quickly by scanning email whether it is junk or not, it is much harder for a machine to make an accurate discrimination. When considering the specification of your Machine, it is often important to ask, how much computation does your software have to do to come to an observation? For example, most spam mail detection is based on analysis of the email contents compared to "known" junk, as well as matching an email's properties with other known junk mail characteristics (such as where the email originated from). A "junk mail probability rating" can be assigned to an email, based on Bayesian analysis of the contents of a message based on sample data currently loaded into the junk mail box.

This leads to consideration of how precise or accurate your information display requirement is. Is it sufficient to assign a junk mail confidence rating value (i.e. 50%) to an incoming email, or is a more precise (but potentially less accurate) "yes" or "no" answer satisfactory?

Although the basic Information Display frame only describes the problem of representing a transitory value on the Display, information display requirements are often more complex. For example, historical information may be important, and users may need to query, organize and manipulate information. If so, this will lead to further analysis of the requirements for display, querying, and retention of information.

2.3.6 Variants

Model Domain:

Sometimes, to simplify the workings of your Display Machine, it is useful to include a “Model domain” of the phenomena being observed in order to answer questions about it (Figure 16). When you do this, you’ve essentially decomposed an Information Display problem into two sub-problems: one that observes the real world and creates model of it (called the Model domain), and another that displays the information based on the more accessible phenomena in the model. It is important to realize that a Model domain is completely distinct from the Real World domain, but it is introduced when events that are unavailable in the Real World would be useful to drive the display. In essence, a Model domain is part of a solution—and not an intrinsic part of the problem.

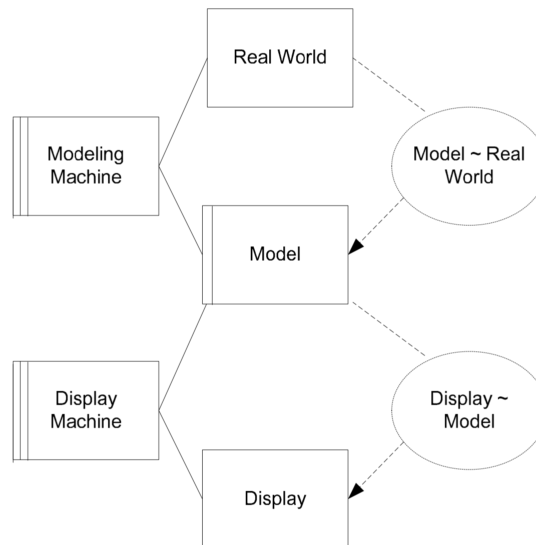


Figure 16: Information Display frame with an added Model domain—an example of a composite frame.

Commanded Information Frame:

In the basic Information Frame, the choice of information to be displayed is fixed in the requirement. But sometimes it is useful to have a kind of information problem where the machine answers questions of a user. Jackson calls this variant a Commanded Information frame. The operator is called the Enquiry Operator whose enquiries are regarded as commands to the Answering Machine. The machine produces its information outputs in the Display domain. In the case of our junk mail rating machine, a Commanded Information frame, shown in Figure 17, would let the user query and view a junk mail report based on specific threshold values.

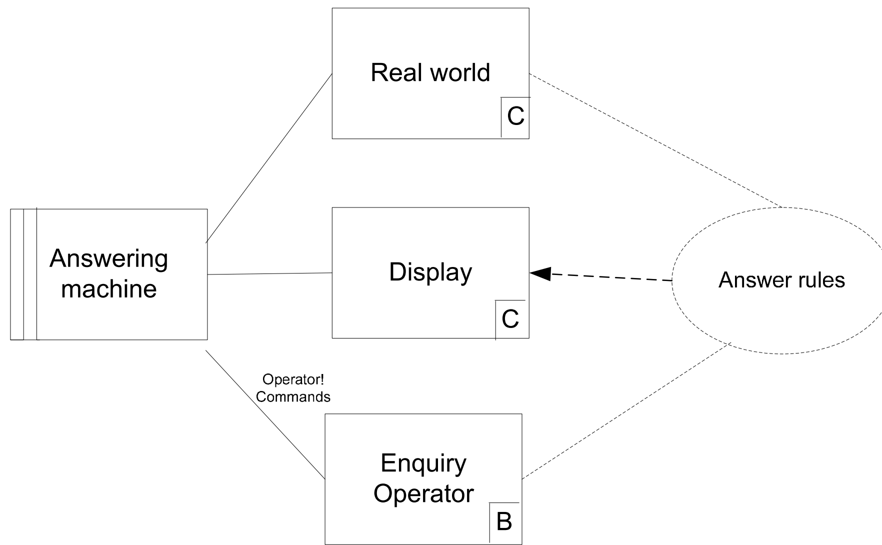


Figure 17: Commanded Information frame.

2.4 Simple Workpieces Problem Frame

2.4.1 Problem

A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analyzed or used in other ways. The problem is to build a machine that can act as this tool.

2.4.2 Example

In order to have email messages to send, an email client must allow users to compose emails. The frame diagram below (Figure 18) shows this problem: this Machine (the Email Editing Tool) must support Users editing a set of email messages. The Email messages (annotated 'X') comprise a lexical domain, that is, a set of symbolic objects rather than a part of the world external to the system. The User domain (annotated 'B'), which interacts with the Email Editing Tool domain, is biddable—that is, in most situations it's impossible to compel a person to initiate an event (your machine can ask, but their response is never guaranteed).

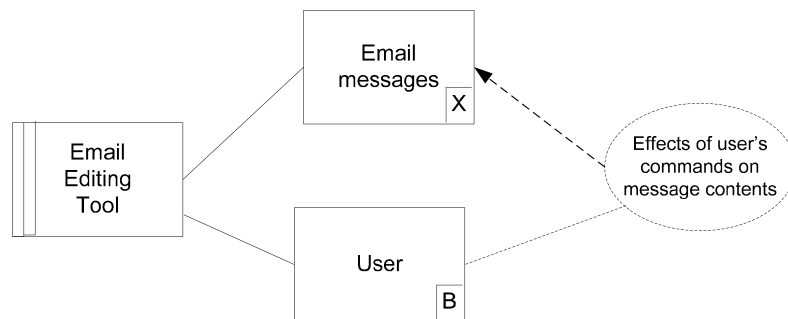


Figure 18: Email editing function mapped onto the Simple Workpieces problem frame.

2.4.3 Structure

This problem fits into the Simple Workpieces problem frame:

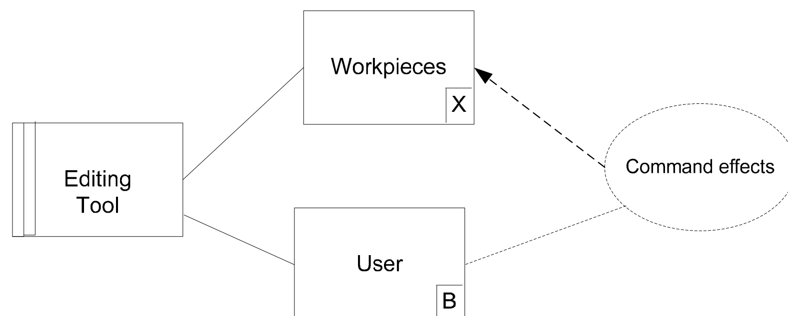


Figure 19: General form of the Simple Workpieces problem frame.

Participants:

- Editing Tool (Email Editing Tool)—the part to be built, this domain issues Operations on the Workpieces in response to User's commands.
- User (User)—autonomously (actively) issues Commands to the Editing Tool to manipulate Workpieces.
- Workpieces (Email Messages)—an inert, lexical (symbolic) domain containing materials to be worked on.
- Command Effects (User's Commands on Message Contents)—the requirement that describes how the User's commands should affect the Workpieces.

2.4.4 Frame Concern

The key concern of the Workpieces frame is that the Machine correctly changes the Workpieces in response to editing Commands. The frame's concern, illustrated in Figure 20, can be stated as follows:

1. When the User issues a Command
2. AND the Machine rejects invalid Commands
3. AND the Machine either ignores a Command if unviable, OR invokes editing Operations
4. AND the editing Operations result in changes of Workpiece values and states
5. ENSURE the changed state meets the Commanded Behavior *in every case*.

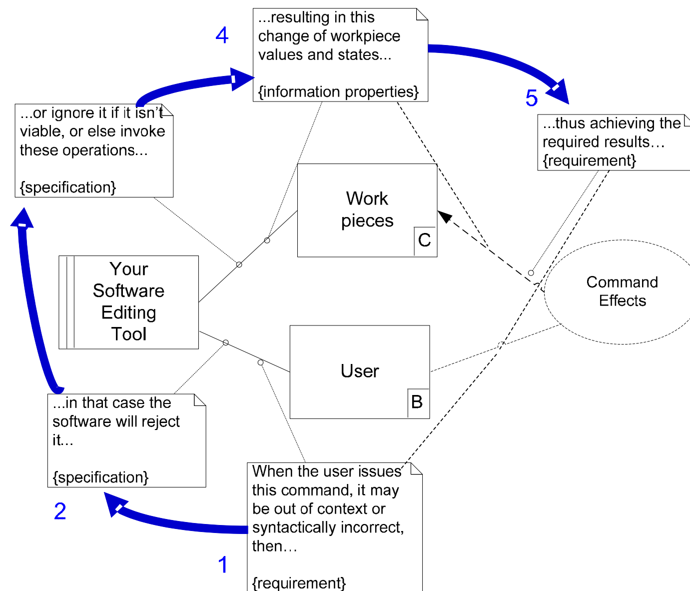


Figure 20: Simple Workpieces frame concern

Concern Resolved:

Referring to the example, we can say that the Editing Tool ensures that commands have the effect on the Workpieces according to the Command Effects requirement:

1. When the User issues an editing gesture (e.g. keystroke, mouse click)
2. AND that command is syntactically correct
3. AND the command is semantically correct
4. AND that command changes the email message being edited
5. ENSURE the message is edited correctly.

2.4.5 Discussion

The Simple Workpieces frame's concern bears a strong resemblance to that of the Commanded Behavior frame. The User has a role and characteristics very similar to the Operator in a Commanded Behavior problem. The chief difference is that Workpieces is a lexical domain whose contents can be manipulated by user commands. It is usually a designed domain (or, in the case of email message contents, a given domain whose correctly formed contents are defined by email message standards).

It is important to identify both the structural elements of the workpiece and the commands that operate upon them. Sometimes a workpiece can take on different forms, or may need to be published or printed. One question to ask is whether a workpiece has an interesting lifecycle, or whether it is just changed and then treated as "static" after each user command. Consideration of lifecycle questions lead to asking whether a workpiece can be shared, and if so, how? It may be that a workpiece is passed around between various users, for example a document requiring approvals or a meeting appointment whose attendees must confirm their attendance. In cases like these, there may be a more complex workflow associated with changes to the workpiece.

2.4.6 Variants

Command File:

Sometimes a Command File can take the place of a User. Instead of the User domain controlling edit events as it does in the Simple Workpieces frame, a Command File (a passive lexical domain) is substituted as shown in Figure 21.

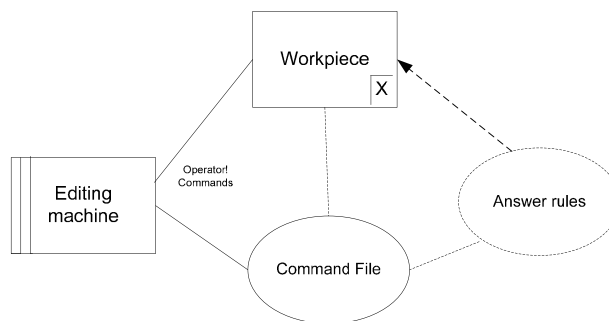


Figure 21: Command File Workpieces problem.

2.5 Transformation Problem Frame

2.5.1 Problem

There are some given inputs which must be transformed to give certain required outputs. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.

2.5.2 Example

Consider multimedia messages, that is, email encoded in some particular way. The frame diagram in Figure 22 shows this problem: the Machine (the Email Decoder) must transform Encoded Email messages into Viewable Email messages, according to some Decoding Requirements.

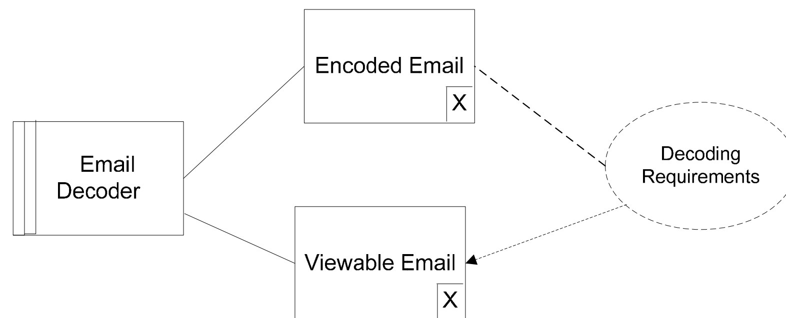


Figure 22: Multimedia decoding problem mapped onto the Transformation problem frame.

The “X” in the lower corner of the Encoded Email and Viewable Email domains indicates that they are lexical domains—physical representations of structured data.

2.5.3 Structure

This problem fits into the Transformation problem frame:

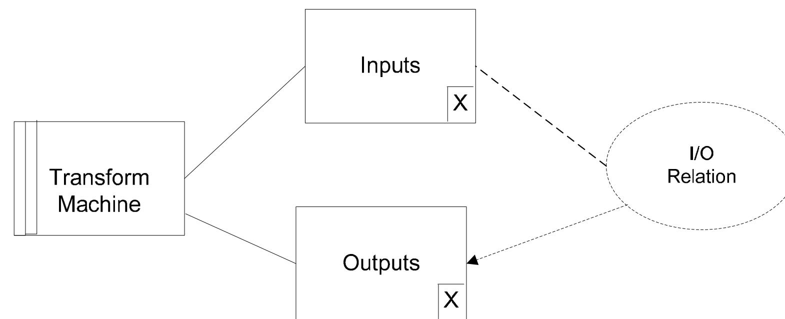


Figure 23: General form of the Transformation problem frame.

Participants:

- Transform Machine (Email Decoder)—the part to be built, this domain transforms inputs into outputs without changing inputs.
- Inputs (Encoded Email)—a static lexical domain containing inputs.
- Outputs (Decoded Email)—a static lexical domain that is to be made by the machine.
- I/O Relation (Decoding Requirements)—a description of the desired relationship between inputs and outputs.

2.5.4 Frame Concern:

The key concern of the Transformation problem frame, illustrated in Figure 24, is that the input is correctly transformed into the output:

1. BY traversing the input in sequence, and simultaneously traversing the outputs in sequence
2. AND finding values in the input domain, and creating values in the output domain
3. AND that the input values produce the correct output values
4. ENSURES the I/O relation is satisfied.

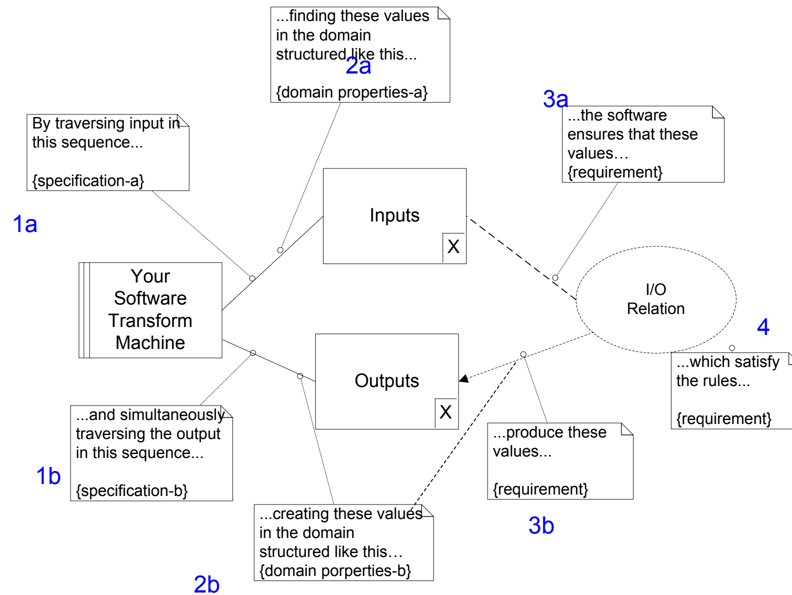


Figure 24: Transformation Problem Frame Concern

Concern resolved:

1. BY traversing an encoded email, and the decoded representation
2. AND finding values in the encoded emails, and creating values in the decoded emails
3. AND that the encoded values produce the correct decoded values
4. ENSURES the email message is decoded correctly

2.5.5 Discussion

In a transformation problem, the two problem domains are lexical. The transform machine traverses over the Input domain, accessing the data values it needs by visiting the places in the domain where they are to be found. In the same way it simultaneously traverses the Output domain, creating data values and depositing them at places where they are required. The frame concern for the Transformation Frame is to show that as the Machine traverses the Input and Output domains that it correctly calculates the values to be written to the proper places in the Output domain.

If the transformation is complex, or the input domain's size isn't well-known or bounded, there are other considerations. For example, the analyst might need to consider what speed, space, or time tradeoffs exist for performing any particular transformation. Is the transformation "lossy", i.e. is it permissible to lose certain information when space and speed tradeoffs must be made? And does the transform need to be reversible?

Efficiency of the Machine and its traversal algorithms is a common concern. A practical efficient traversal tries to avoid multiple visits to the same data or unnecessary visits to irrelevant data. If the transformation is complex, then algorithmic descriptions may be part of the Machine specification.

Another question to ask is whether a transformation will always work. What should happen when your transform machine encounters anomalies in the Input or unknown data in the Input domain? For example,

what should happen when a particular encoded element cannot be read by the Email Decoder? Should it ignore it, put it in the Output as some distinguished (uninterpretable item) and continue, or terminate?

2.5.6 Variants

Description Domain:

A more flexible way of treating a transformation problem is to add a description domain that guides the behavior of the machine. For example, the definitions of tokens and their types are encoded in a description domain that a Lexical Analyzer interprets during its traversal of the Input stream (Figure 25). The Token definitions domain describes the relationship between the characters or values in the input domain and tokens. The requirement is that the Output domain should contain token records corresponding to the Input domain tokens.

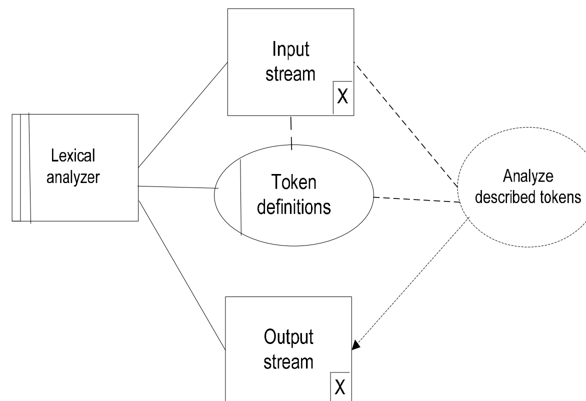


Figure 25: Transformation problem description variant.

3 Assessment and Conclusions

Now that Jackson's problem frames have been elaborated in pattern form, we can assess just how their expression as patterns has helped us understand and use problem frames. One intriguing question is the similarities and differences between problem frames and patterns. On face value, the two have much in common. They are both based on a structural classification and require decomposition or abstraction of aspects of the problem at hand. They both require selection, fitting and interpretation of a reference structure (a pattern or problem frame). Once fitted, they both dictate a highly specific process (a 'micro-method'). And both of them can be combined, with parts that can overlap and other parts that must remain separate. However, when we look closely, we can also see a number of important differences. Problem frames produce descriptions, whereas patterns produce architecture or code structure. Problem frames are top-down (they have their roots in formal specification) whereas patterns are bottom-up (they are rooted in practitioner experience) and emergent design. A problem frame is a template that arranges and describes phenomena in the problem space, whereas a pattern maps forces to a solution in the solution space. And finally, problem frames are method-centric (frames are subordinate to methodology), whereas patterns are artifact/asset-centric—they focus on particular designs (i.e. the patterns) and those designs are useful across a wide range of development methodologies, from UML Design Up Front to Extreme Agile Hacking [10].

Patterns are about designing things. The fact that we put problem frames into pattern form demonstrates that when people write specifications, they are designing too—they are designing the overall system, not its internal structure. And while problem frames are firmly rooted in the problem space, to us they also suggest solutions. When solving translation problems it seems reasonable to check out patterns about how to write parsers, or to consider the Command pattern when designing a solution to a Commanded Behavior problem (or most frames involving a user-operator domain). Required Behavior problems suggest investigating event and event handling patterns, finite state machines, or reactive system design patterns. And query-report patterns come to mind when solving the Commanded Information Frame variant. Likewise, when designing a Model domain, inspiration can come from considering the nature of that model and various patterns that may apply based on its necessary behavior and intrinsic structure. So it appears that problem frames usefully suggest patterns. But the link seems tenuous.

Connections between problem frame concerns and potential design pattern solutions certainly exist. It seems fruitful to view framing as one way of guiding the exploration for potential solutions. However, stronger connections between frame concerns and architectural or design patterns, other than those we have mentioned, don't appear so obvious. This may be because certain design patterns resolve tensions that are intrinsic to the solution, not the problem. Design patterns as a whole need to be better organized before more connections can become clear.

One recurring question that arises from our view of problem frames as patterns is how they help the analyst—specifically, the question of how people are supposed to use problem frame patterns. We suggest that as you look at a problem to be solved ask, "Is there a workpiece here? Or a transformation or a required behavior? What problem frames seem to predominate?" You'll apply a frame and see whether it

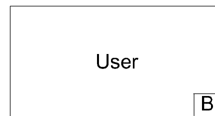
fits. And this leads to meaningful questions to ask. As you explore a problem, you will discover ancillary problems and decompose larger problems into sub-problems. You'll try to write requirements. While a phenomenological world view can lead to formal descriptions of events, states, statements about truths and cause and effects, we haven't found ourselves "going more formal" just because we know problem frames. Instead, we find that knowing problem frames leads us to ask deeper questions and distinguish requirements from assertions, wishes, and technology constraints. Framing also helps us spot the need for developing more rigorous state models, event descriptions or behavioral rules.

We find that analysts steeped in other forms of analysis descriptions and models find problem frames to be interesting but not immediately applicable. Problem framing doesn't seem to supplant other analysis activities nor do phenomenological descriptions replace other analysis artifacts. Integration of problem framing with other analysis activities needs further investigation. We hope that by writing about problem frames as patterns, we can expose problem frames to a wider audience who in turn integrate framing activities with their other analysis activities and report on their experiences.

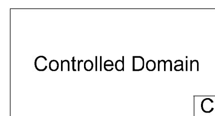
4 Glossary

Problem Frames have their own specialized terminology. Here's a brief illustrated glossary, shamelessly culled indirectly from the Problem Frames Book.

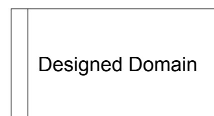
Biddable Domain—a domain that a Machine can tell what to do (although the outcome of such a mutation is not necessarily reliable or predictable). 'B' marks a biddable domain.



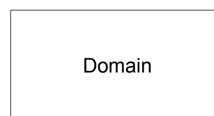
Causal Domain—a domain that a Machine can tell what to do, and where the outcome is perfectly predictable. 'C' marks a causal domain.



Designed Domain—a realization of a description or model that the developer is free to design. A box with a single stripe is a designed domain.



Domain—a collection of phenomena. A domain is designated by a box.



Domain Dependency—two domains may be linked by shared phenomena.

Entity—an individual phenomenon that persists over time, changing properties and state. In an email application, emails or mail folders and their contents are entities.

Event—an individual phenomenon representing an indivisible, instantaneous happening taking place at some point in time. In an email client problem, "email sent" or "email received" are events.

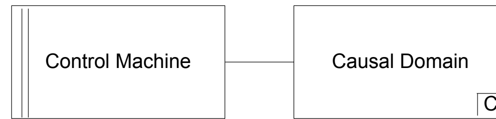
Frame Concern—an argument (or argument schema) that describes how the Machine Domain must interact with other Domains within a Problem Frame if the specification accurately fits that Frame.

Given Domain—a domain that is given or fixed in a particular problem, that is, it is not subject to change (it is pre-established).

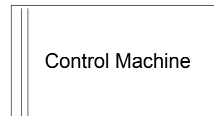
Individual Phenomena—individual elements of a domain that may be observed. Classified into states,

truths, and roles.

Interface—a connection among domains consisting of phenomena that they share. On a frame diagram, a connection is represented by a solid line between two domains.



Machine (or **Machine Domain**)—the software program you are specifying. A machine is drawn as a box with double stripes.

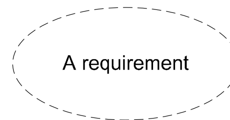


Lexical Domain—a domain that is a set of data with a deterministic structure. An 'X' marks a lexical domain.

Phenomena—something that may be observed; a part or quality of a domain. Classified into Individual Phenomena and Relationships.

Problem Frame—a set of Domains and Interconnections that describes a recurring problem structure.

Requirement—a condition on one or more domains of the problem context that the machine must bring about, for example a stipulated correspondence between an information display value and the reality it concerns. A dashed oval is a requirement.



Role—a relation between an event and individuals that participate in it such as Sendmail(Emailxxx,Outbox).

State—a relationship phenomenon (or predicate) among two or more individual phenomena that can be true at one time and false at another. So, for example Temperature(myOffice, 72) is a state as is Sent(Emailxxx).

Truth—a relationship among two or more individuals that cannot change that is either true at all times or false at all times. LaterThan("timestamp: 9.9.2007", "timestamp: 9.1.2007") is a truth just as GreaterThan(5,3) is.

Value—an immutable individual phenomenon existing outside of time and space—such as numbers or characters.

5 Acknowledgments

Many thanks are due to Susan Kurian, who ably shepherded this paper for PLoP 2006.

Thanks also to John Schwartz for his discussion of meaningful frame questions with Rebecca, to Jim Holt and Nathan Ward for their development of a working frame example with Rebecca, and to Nathan for his inspired leading of a Problem Frames reading and study group I (Rebecca) participated in.

Finally, we'd like to acknowledge Michael Jackson for his contribution of problem frames to the software community.

6 References and Resources

[1] *Software Requirements and Specifications*, Michael Jackson, Addison-Wesley, 1995.

[2] *Problem Frames: Analyzing and structuring software development problems*, Michael Jackson, Addison-Wesley, 2001.

[3] http://www.ferg.org/pfa/ —A website devoted to problem frames and their application.

[4] http://mcs.open.ac.uk/mj665/ —Jackson's home page.

[5] http://www.wirfs-brock.com/rebeccasblog.html —Rebecca's Blog (including some entries about problem framing).

[6] http://homepage.mac.com/jon_hall/Academic/IWAAPF06/ —The 2nd International Workshop on Advances and Applications of Problem Frames.

[7] http://csdl2.computer.org/comp/proceedings/re/2001/1125/00/11250306.pdf —Geographic Frames, Maria Nelson, Donald Cowan, and Paolo Alencar, Proceedings of the Fifth International Symposium on Requirements Engineering, 2001.

[8] http://csdl2.computer.org/comp/proceedings/re/2003/1980/00/19800371.pdf —Introducing Abuse Frames for Analysing Security Threats, Luncheng Lin, Bashar Nuseibeh, Darrel Ince, Michael Jackson, Jonathon Moffett; Proceedings of the Eleventh International Symposium on Requirements Engineering, 2003.

[9] http://mcs.open.ac.uk/mj665/ArchDrvn.pdf —Architecture-driven Problem Decomposition Lucia Rapanotti, Jon G. Hall, Michael Jackson and Bashar Nuseibeh; Proceedings of the 2004 International Conference on Requirements Engineering.

[10] http://www.xpuniverse.com/2001/pdfs/Edu02.pdf —Adapting Problem Frames to eXtreme Programming, James Tomayko.