Problem Frame Patterns

An Exploration of Patterns in the Problem Space

Contents

1 Problems and solutions

Software developers solve problems in code. It's a part of our identity to decompose, resolve, or drive toward a solution quickly and efficiently. We naturally gravitate to 'the solution space' where our architectures, designs and mechanisms 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 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 conceptualisation 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 (Jackson 95) 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 that interact with these elements.

In Jackson's problem frames, the problem context is described as consisting of the machine and an application domain. 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. 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.

Jackson suggests that a requirement should be expressed in terms of the *context* rather than in terms of the *machine*. By expressing a requirement in terms of the machine, we risk jumping to the solution space prematurely, and we are in danger of missing important aspects of specification and opportunities for redesign (and reuse) outside the existing experience or existing processes or machines.

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 some of Jackson's problem frames—the Control Frame, the Workpieces Frame, and the Connection Frame. The Control 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—anexample is an electronic thermostat for temperature control. The frame consists of three principal parts. The *machine* (the component to be built) shown as a double-lined box. The machine is associated with a single *domain* (the *Controlled Domain* depicted by a rectangle) by a line, representing a kind of of connection. The domain is in turn connected by another line to a named set of *properties* (the *Desired Behavior* requirement depicted by an ellipse).

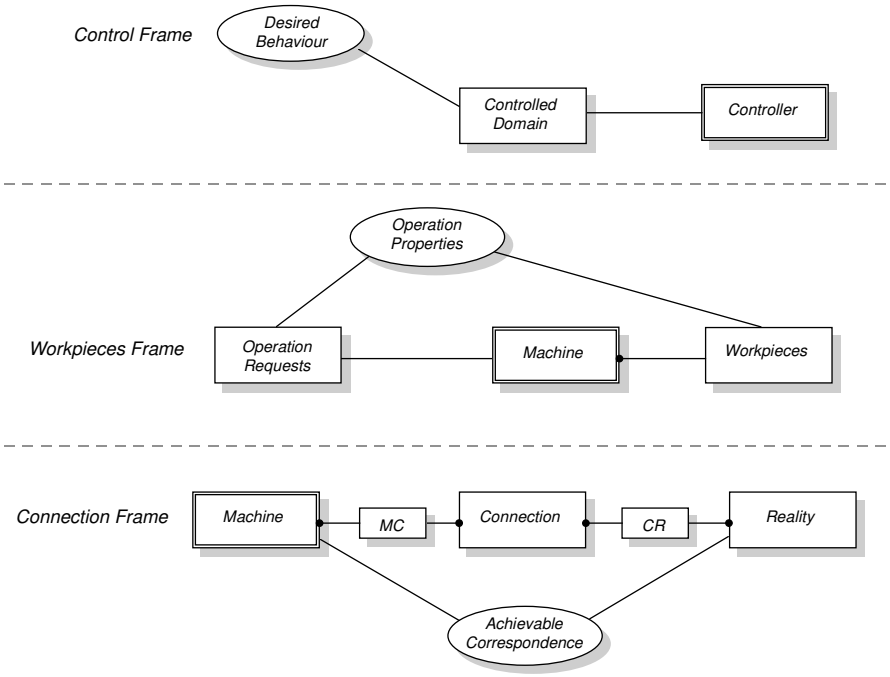


Figure 1: Three of Jackson's problem frames.

Although abstract, this problem frame generalizes one of a number of basic structures that underlie every system and architecture that solve problems *of this type*. Recognizing the structure of the *problem* and adopting a frame such as this helps us 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 later work [2] Jackson renamed Control Frame to Required Behavior Frame and the Workpieces Frame to Simple 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 Information Display Frame. Work is progressing toward elaborating these frames and identifying new ones [3,7,8,9].

1.3 Reasons for liking problem frames

There are some good reasons to like problem frames. As an architect and developer, your goal is to design and build software that will behave appropriately and solve the customer's problem. Jackson advocates that you convince yourself and your customer that your proposed software will tackle the right problem by

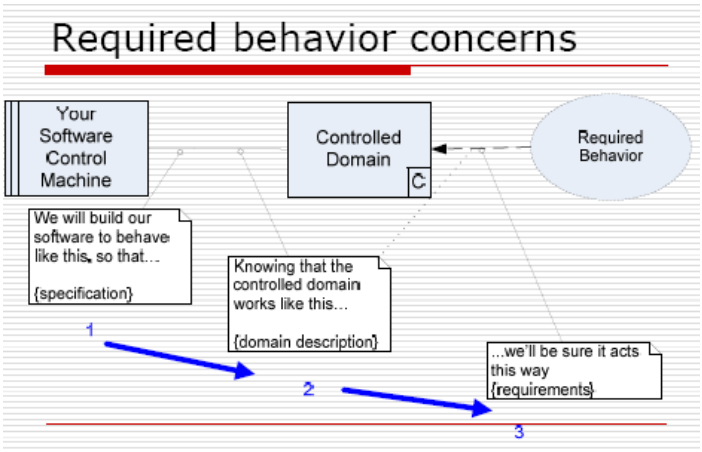
writing an appropriate set of descriptions about the problem domains. As a problem framer, your central task is to investigate and describe problem domain properties. Each type of problem frame has a different set of concerns that are typically addressed. You can think of each problem frame as having its own micro-methodology. So when you choose a problem frame, you choose a problem-structuring template that encourages you to *ask the right questions* and *specify the right things*.

Because problem frames provide a structure for analysis and description, the analysis effort can be correctly targeted and the most appropriate method for the solution task is implied. In practice the need for analytical judgement and decision making is not eliminated, rather it is transferred from the detail of analysis to selection of a problem frame and creation of the mapping between problem elements and frame's principal parts. This is consistent with the goals of the pattern movement—a practitioner should be able to reuse the community's time-proven collective knowledge of problem analysis by simply fitting the pattern (problem frame) to the problem at hand.

1.4 Using problem frames

Using a problem frame involves selecting a candidate frame from a catalogue 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. 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 both occur simultaneously.

<< Rebecca, here's another of your slides that nicely illustrates how to use the frame … >>



Each problem frame also describes a *frame concern*. The frame concern 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 correct analyses of the problem you are studying, frame concerns can be useful to check that you have characterized your problem with the correct frame. Basically, if you are 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.

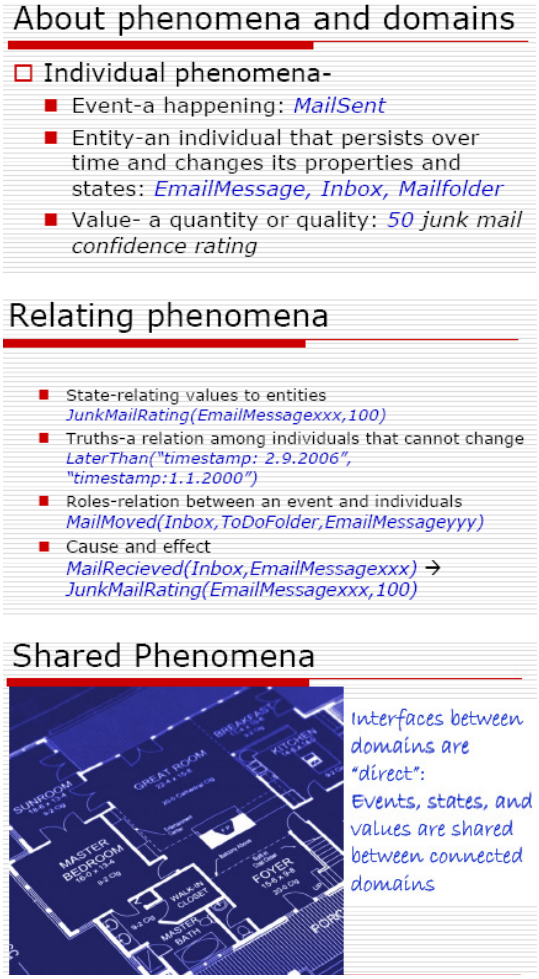
By helping you to ask the right questions, frames improve specification quality. There is another benefit—frames encourage you to separate the concerns of the problem into demarcated sub-spaces (domains or problem parts) in the overall problem space, and then to treat each in turn according to its specific needs. This form of 'separation of concerns' helps you to develop minimal and contextual descriptions, and ensures that the machine you specify sits comfortably in its phenomenological world. 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.

1.5 The *phenomenology* thing

Earlier, we stated that problem frames assume a phenomenological philosophical stance, which is different from the underlying philosophy of conventional information systems design methodology. These are big words but their meaning for problem framers can be stated with small ones. Conventional methods such as functional decomposition assume that there is an objective reality that the analyst must discover and understand objectively. The analyst's task is to describe the implications of the domain's invariant characteristics for the proposed system as input to a system design activity. The phenomenological position holds that the only meaningful assertions that can be made are from observed behavior. Phenomenologists construct theories and descriptions from observed behaviors, paying close attention to contextual and situational factors and forces. A problem framer does just this by asking specific questions and studying actual phenomena within tightly defined domains. Frames dovetail with this philosophy by promoting

definition and description of small, targeted domains at the expense of high-level architectural structure and process.

<< Rebecca, here's three clippings from your Powerpoint – if you think these are relevant you could elaborate on the key messages in this section… >>



1.6 Our motivations

Problem frames were first published by Jackson in 1995 [1], with a follow-up book in 2001 [2]. Two international workshops have been conducted for researchers (IWAAPF 2005 and 2006) and while the idea appears to be gathering broad academic acceptance, its use in industry is some way off. So why have we decided to pursue problem frames, and what benefits do we expect to see by expressing them in pattern form?

We find ourselves intrigued by the promise of problem frames and their similarity with patterns. On face value, problem frames and patterns may be kissing cousins:

- 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 for the problem
- Once fitted, they both dictate a process (a 'micro-method')
- They can both be combined, with parts that can overlap or be shared and parts that must remain separate

However, when we look closely, we can also see a number of important differences that make problem frames and patterns look more like members of feuding families:

- 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

We think it is about time the patterns community took a look at problem frames. For one thing, we think that problem frames look just too intriguingly similar to patterns to be ignored, and we want to understand the relationship between the two. This paper is an attempt to expose these similarities and differences. Another reason is that the patterns community is (arguably) some way down the track of making patterns mainstream, and some of our lessons might be applicable in the development and promotion of problem frames. Finally, we think that those interested in patterns will be similarly intrigued by problem frames. Many other questions are suggested:

- Are problem frames an early attempt at applying patterns to specification? If so, are they useful?
- Are problem frames 'patterns in the problem space'? If so, what does this mean for patterns?
- Is there value in treating problem frames as patterns? If so, where are the forces? Where is the solution?
- Is it meaningful to associate a given problem frame with a set of design patterns? In other words, can design patterns instrument a problem frame's micro-method?
- Do Alexanderian concepts such as pattern language, generativity and emergence have an analog in problem frames?
- 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].

While we obviously don't expect to answer many of these questions, we include them to share a sense of

the possibilities. We should also declare our own personal motivations for pursuing this chimera—our motivations for promoting problem frames in the patterns community and for attempting to write problem frames as patterns include:

- to investigate and explore the relationship between problem frames and patterns in the hope of finding common ground
- to learn for ourselves about problem frames by making associations back to familiar concepts from pattern theory and practice
- to identify options for *informal* and *agile* approaches to problem frames that take the approach out of its methodological origins and ground it in the techniques and language of system and software constructors.

1.7 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, and the running example. The problem frames and their definitions are taken verbatim from Jackson's books, and the frame diagrams from Stephen Feng's Wikipedia entry on Problem Frames. 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 problem frame patterns, one for each of Jackson's problem frames. The names of the five problem frames, and the classes of problem that each address, are as follows:

- Required Behavior Problem Frame—there is some part of the physical 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—there is some part of the physical 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—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—there is some part of the physical 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.
- Transformation Problem Frame—there are some given computer-readable input files whose data must be transformed to give certain required output files, 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, beginning with a brief example, drawn from a running example that depicts the specification of an email client. This client can exchange messages with email servers, display junk messages, allows 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 shows 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. Finally, 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.

Both specific problems and abstract frames are drawn using Jackson's problem frame diagram notation.

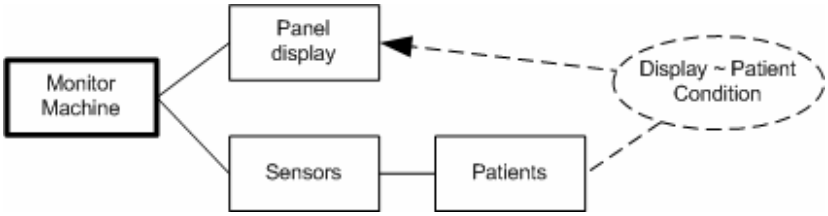


Figure 2: An example of Jackson's problem frame notation (for a medical monitoring machine).

These diagrams show *domains* (rectangles) and *requirements* (dashed ellipses). Domains generally represent sets of phenomena in the real world—in this example, 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 is built and machine 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 either with a dark border or stripes on the left-hand side of the rectangle.

Domains are linked by lines representing *shared phenomena* between them; that is, phenomena that occur in each domain. In Figure 2'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 2'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 Class

There is some part of the physical 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 of Use

The most basic requirement for an email client (the most fundamental problem that a client has to solve) is to send emails from the client program to some external Internet Service Provider, and to get emails back from that provider. The problem frame diagram in Figure 3 illustrates how the Required Behavior problem frame fits this simple description of the client's most basic function. The Machine (called the 'Automatic Email Controller') must interact with the immediate endpoint of the emails (the Internet Service Provider) and that interaction must satisfy the requirement that emails are correctly exchanged between the two.

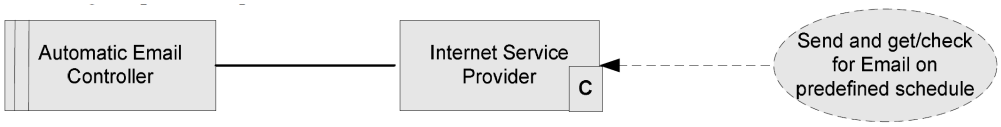


Figure 3: Basic email client operation mapped onto the Required Behavior problem frame.

<< what does the 'c' in the box mean? >>

<< would it make more sense if we replaced the ISP with a mail server? >>

2.1.3 General Form

The generalized form of the R**equired Behavior** problem frame is shown in Figure 4.

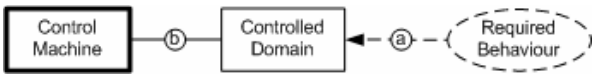


Figure 4: General form of the Required Behavior problem frame.

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

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

<< what are ‘b’ and ‘a’ ? >>

The problem frame’s parts are related in the following way:

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

Referring to the example, we can say that the behavior of the email client AND the properties of the Internet Service provider ENSURE that email will be sent and received according to the predefined schedule.

2.1.4 Application

To fit the 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 questions:

Firstly, 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?

This moves you on to considering the connection domain between the machine and the Controlled Domain, where you’ll need to describe the complexity of interactions between your software and the thing under its control, and whether they require or imply sequence or have other dependencies.

Another Connection Domain issue is how your software Machine will find out whether its actions have had the intended effect. Many machines equipped with actuators require sensors to create a corrective feedback loop. Does your Machine need to know for certain, or can it just react later (when the state of something in the Controlled Domain is not as expected)? How will the machine know when things get ‘out of synch’ between your software and the thing it is supposedly controlling, and what should your machine do at this point?

<< need to cover remaining frame parts >><< check this pattern coverage against Jackson’s description >>

2.1.5 Variants

The part of the Required Behavior problem frame that varies most across different problems is the Connection Domain. 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. 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 and specify them fully. When software

system designers assume the simpler case and overlook Connection Domain complexities (which happens often in the desire to selectively ignore complexity) the reliability of the Machine can be dramatically reduced.

2.1.6 Discussion

<< it feels unfinished at this point -- do we want to add a discussion on each pattern?

Some things that we might drop in include:

So what are the strong points and weak points if the frame as described?

How did the frame help us in specifying the email client?

What kinds of specification and design problems would this frame help us to avoid?

>>

2.2 Commanded Behavior Problem Frame

2.2.1 Problem Class

There is some part of the physical 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 of Use

An email client cannot only communicate with a service provider: it must also allow users to send and receive email The frame diagram below shows this problem: the Machine, the Email Client, must exchange emails with the Internet Service Provider with the requirement that the interactions with the service provided are commanded by the client.

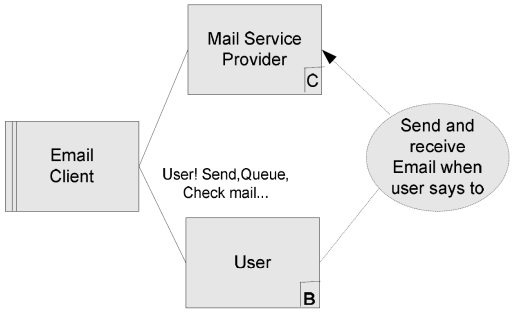


Figure 5: User's control of the email client mapped onto the Commanded Behavior problem frame.

2.2.3 General Form

This problem fits into the commanded behavior problem frame:

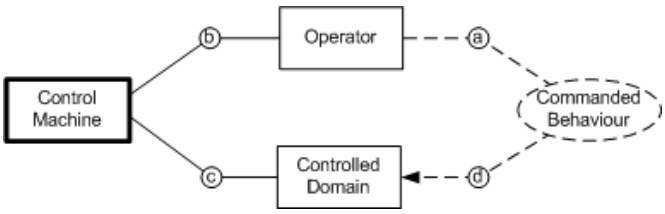


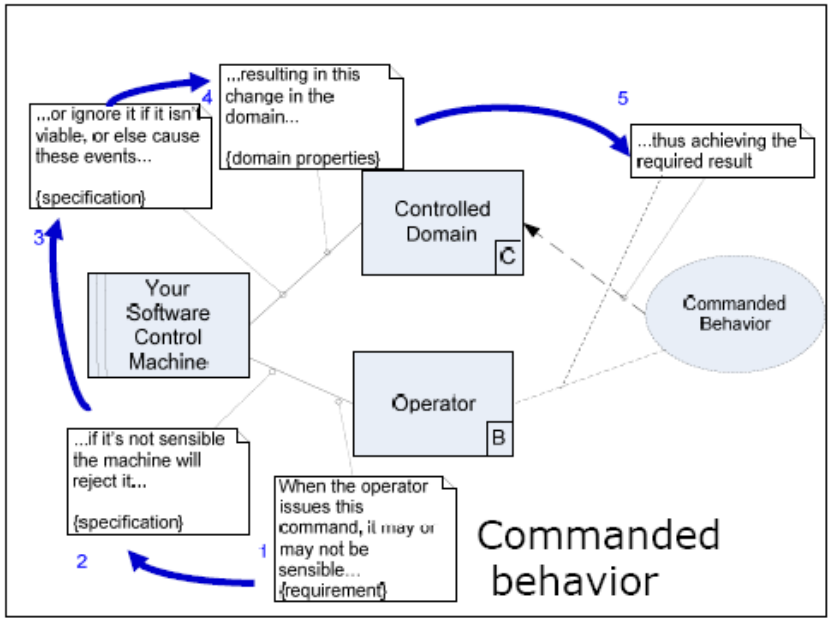
Figure 6: General form of the Commanded Behavior problem frame.

Participants:

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

Frame Concern:

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*



Concern Resolved:

1. When the User issues a command
2. AND the machine accepts that command only if it is valid
3. AND the machine issues email requests only if they are viable
4. AND the email requests exchange email with the mail service provider
5. ENSURE that mail is exchanged only when the user says so

2.2.4 Application

- What's a good model of user-system interaction?
- What does the user need to know in order to "command" the system to do things?
- Do certain commands need to be inhibited based on the current state of the system? Do they always make sense? Does a sequence of actions make sense?
- Is there a lag between issuing a command and the system performing the action? Is that a
- problem?
- What happens when a command fails? How should users be involved in "steering" the software when a command fails?
- Should certain commands be ignored (e.g. how many times do you need to press the elevator button to call the elevator to your floor)?
- Do commands need to be reversible? logged? monitored or otherwise tracked?

2.2.5 Variants

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 user issued commands and required behavior? If so, then a specification of frame concern priorities may be needed.

2.3 Information Display Problem Frame

2.3.1 Problem Class

There is some part of the physical 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 of Use

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 users receive. The frame diagram below shows this problem: the Machine, the Junk Mail Filter must inspect the Incoming Mail and then produce a report of all junk mail.

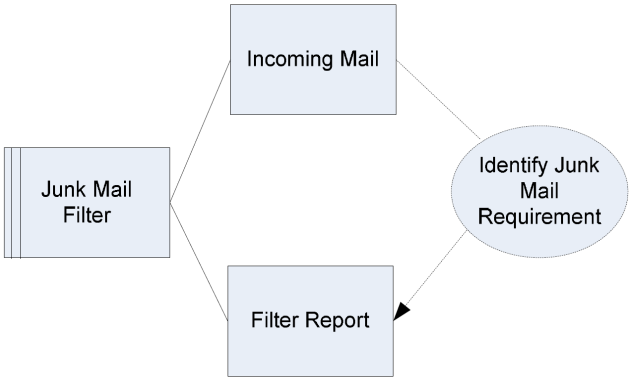


Figure 7: Junk mail detection problem mapped onto the Information Display problem frame.

2.3.3 General Form

This problem fits into the information display problem frame:

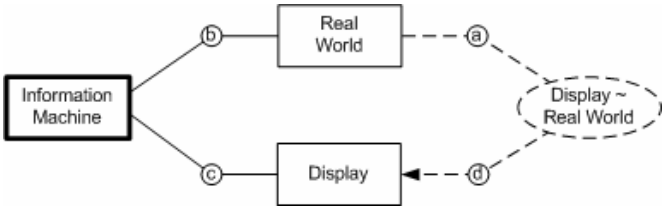


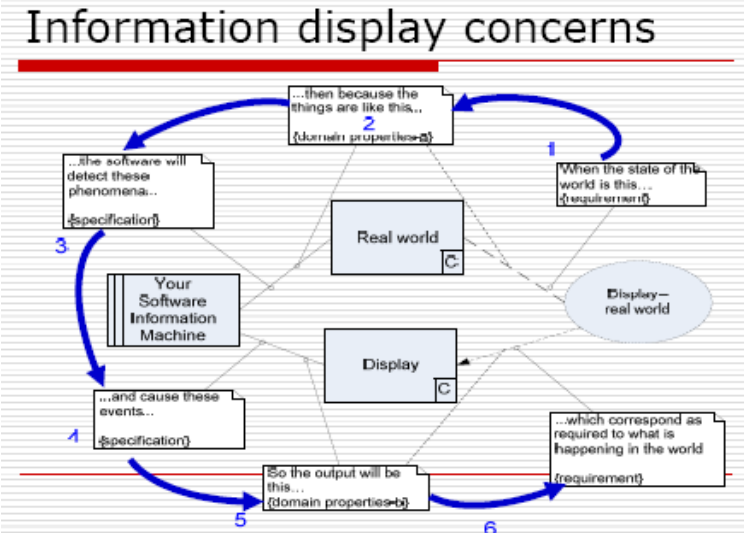
Figure 8: General form of the Information Display problem frame.

Participants:

- Information Machine (Junk Mail Filter) — to be built, displays information from the real world (e.g. 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 Display must show true information about the Real World.

Frame Concern:

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
5. AND the Display produces some output in response to those events
6. ENSURES the Display can be interpreted as corresponding (as required) to the real world



Concern Resolved:

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
4. AND it sends the title 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 their Incoming Mail stream

Implementation

2.3.4 Application

- What is the form of “observation” that the software must make about some event or fact or thing?
- Is it difficult to ascertain when an event has occurred? (For example, if your software is trying to record how many “vehicles” passed over sensors place on the road it may be very difficult to characterize what constitutes a vehicle—is it two axles passing within a time period, but what about motorcycles, backed up slow traffic, etc., etc.)?
- How precise does the information need to be? Is the information “fuzzy”?
- How much computation does your software have to do to come to an observation? (For example,
- consider assigning a “junk mail rating” to an email, based on Bayesian analysis of the contents of the current message based on sample data currently loaded into the junk mail box?
- Is the user only interested in current information? Or is historical information important?
- Are there questions that the user may want to ask about the information? What are they? How easy are they to accurately answer?

2.3.5 Variants

Do you need to include a “model domain” of the phenomena being observed in order to answer questions about it?

2.4 Simple Workpieces Problem Frame

Formatted: Bullets and Numbering

2.4.1 Problem Class

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 of Use

In order to have email messages to send, an email client must allow users to compose emails. The frame diagram below shows this problem: this Machine, the Email Editing Tool must support Users editing a set of email messages. The Email messages (annotated 'X') are a lexical domain, that is, a set of symbolic objects rather than a part of the world external to the system.

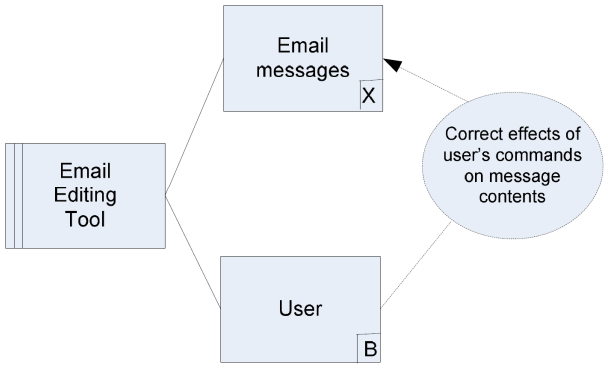


Figure 9: Email editing function mapped onto the Simple Workpieces problem frame.

2.4.3 General Form

This problem fits into the simple workpieces problem frame:

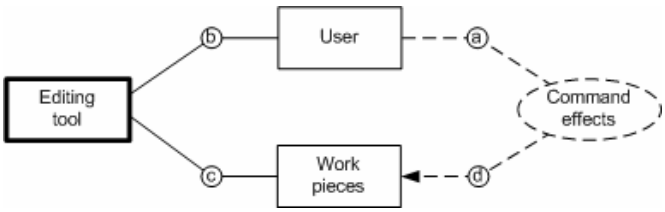
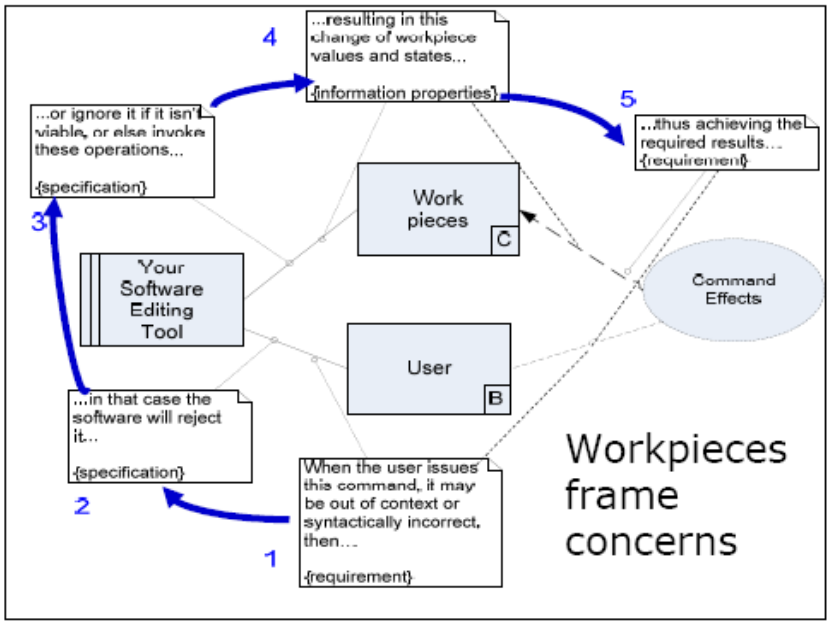


Figure 10: General form of the Workpieces problem frame.

<< Rebecca, again, from your Powerpoint, a nice annotated version of the frame… >>



Participants:

- Editing Tool (Email Editing Tool) — to be built, issues Operations on the workpieces in response to User's commands
- User (User) — autonomously (actively) issues Commands to edit Workpieces.
- Workpieces (Emails Messsages) — inert, lexical (symbolic) domain containing materials to be worked on.
- Command Effects (User's Commands on Message Contents) — describes how the User's commands should affect the Workpieces

Frame Concern:

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*

Concern Resolved:

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 <<check wording>>
4. AND that command changes the email message being edited
5. ENSURE the message is edited correctly

2.4.4 Application

- What are the basic elements of the workpiece?
- Will it take different forms?
- Does it need to be shared? If so, how?
- Does it have an interesting lifecycle (or is it just something that is changed and then treated as "static" after each change?
- ▫s it passed around between various users? Is there a workflow associated with a workpiece?
- Should it persist? In what forms? Should it be published or printed?

2.4.5 Variants

2.5 Transformation Problem Frame

Formatted: Bullets and Numbering

2.5.1 Problem Class

There are some given computer-readable input files whose data must be transformed to give certain required output files. 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 of Use

Finally, we consider multimedia messages, that is, email encoded in some particular way. The frame diagram below shows this problem: the Machine, the Email Decoder, must transform Encoded Email messages into Viewable Email messages, according to some Decoding Requirements.

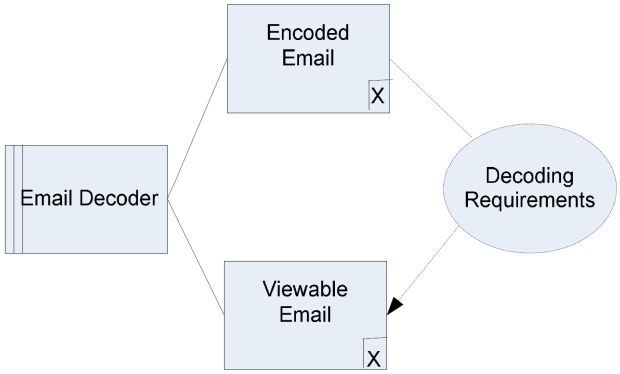


Figure 11: Multimedia decoding problem mapped onto the Transformation problem frame.

2.5.3 General Form

This problem fits into the transformation problem frame:

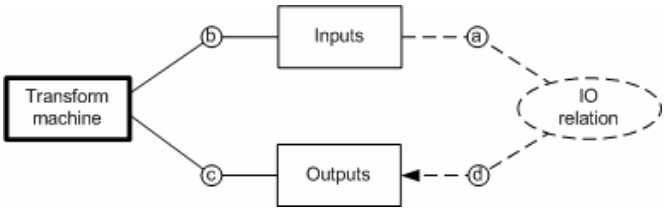


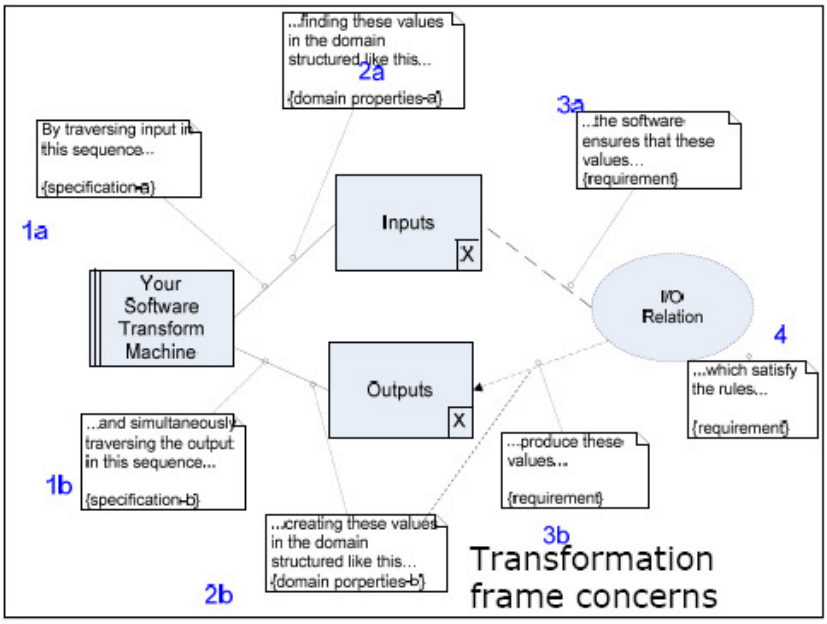
Figure 12: General form of the Transformation problem frame.

Participants:

- Transform Machine (Email Decoder) — to be built, transforms inputs into outputs without changing inputs
- Inputs (Encoded Email) a static lexical domain.
- Outputs (Decoded Email) — a static lexical domain that is to be made by the machine
- IO Relation (Send and Get Emails) — describes relationship between inputs and outputs

Frame Concern:

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 IO relation is satisfied



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

2.5.4 Application

- What data do you start with?
- How will it be changed?
- Is the transformation complex?
- Will it always work? What should happen when you encounter errors in the input?
- Is the transformation “lossy” or reversible?
- What speed, space, or time tradeoffs are there for performing any transformation?

2.5.5 Variants

3 Assessment and Conclusions

<<

This is where we should respond to as many of those questions we pose in the first part of the paper as we can -- we don't have to answer them, just says something that we've learned from the exercise of writing out the frames in pattern form and working the email client example through.

>>

4 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.