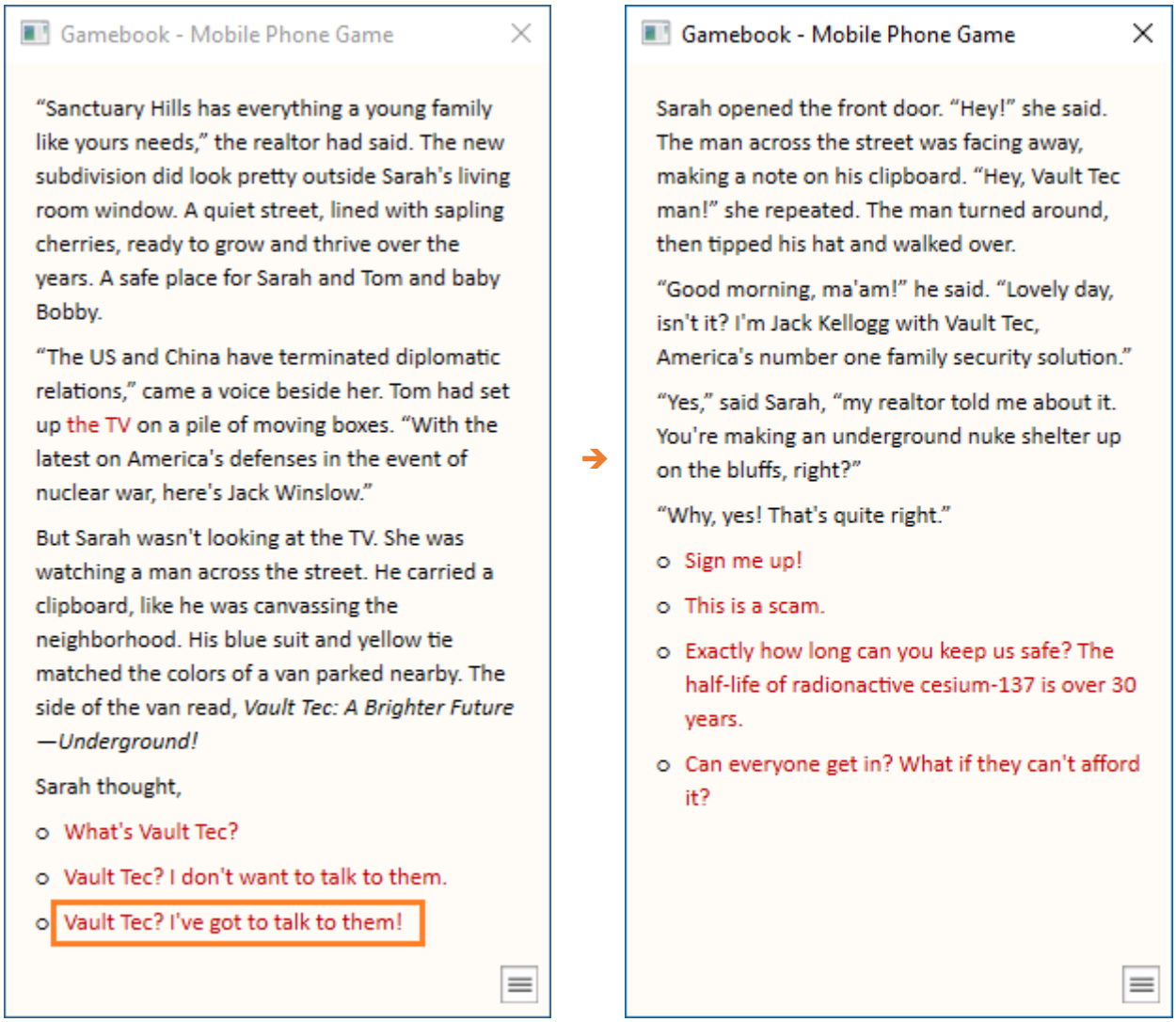
Text game

I created this program as a tool to help develop *Fallout: New Vegas*-style games. Rather than making a whole graphical game, this tool focuses on making it easy to experiment with and develop these features:

- Text descriptions of the game situations and environment.
- Dialog and action trees.
- Character personality scoring (like the "SPECIAL" system in Fallout).
- Characters, factions, backstories, etc.

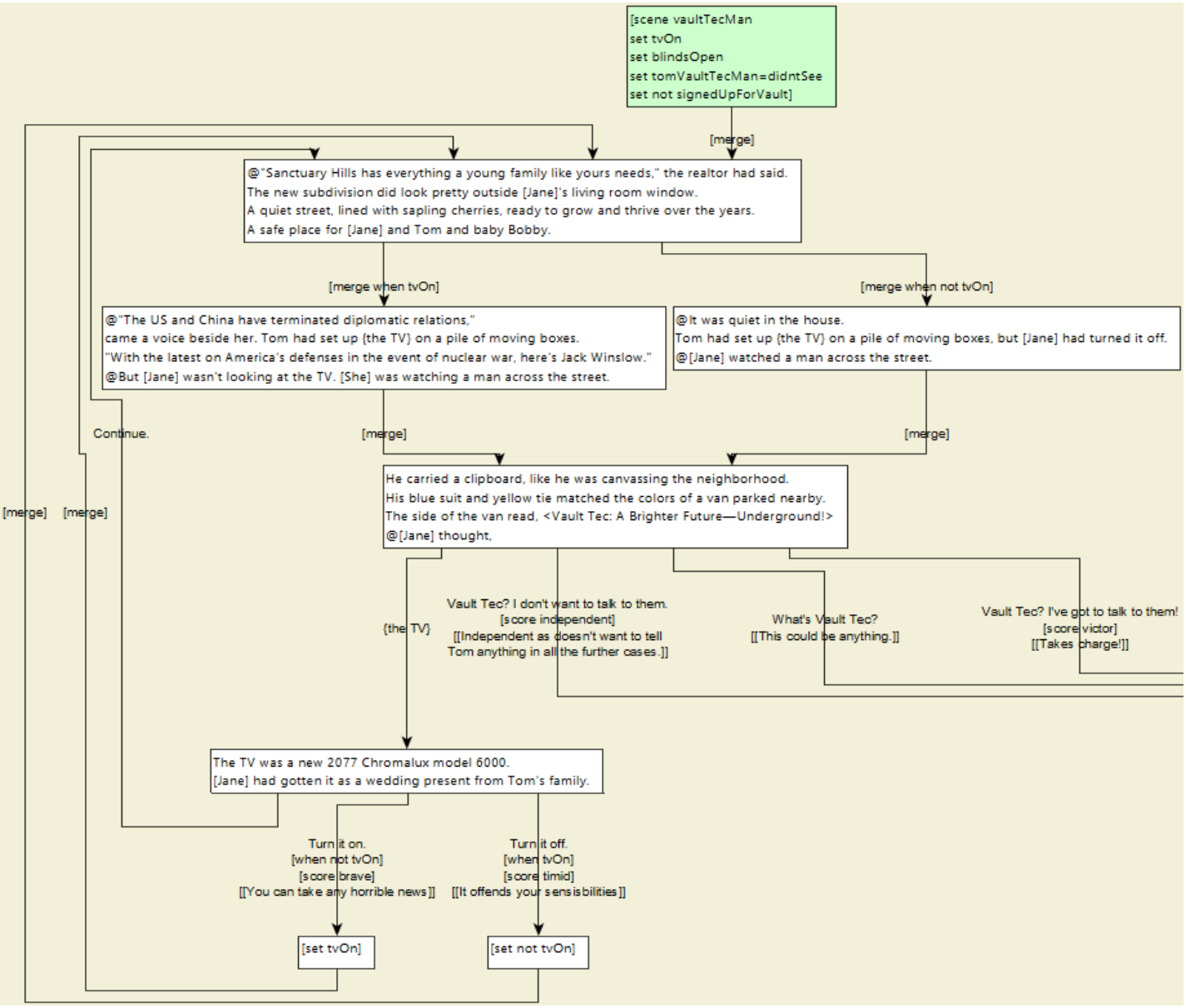
Example

Here's an example of how the game works. This uses a story based on the *Fallout 4* opening, but it's more detailed about the suburban family we see at the start of the game. The left screen is the opening. If you pick "Vault Tec? I've got to talk to them!", the screen changes to the one on the right:



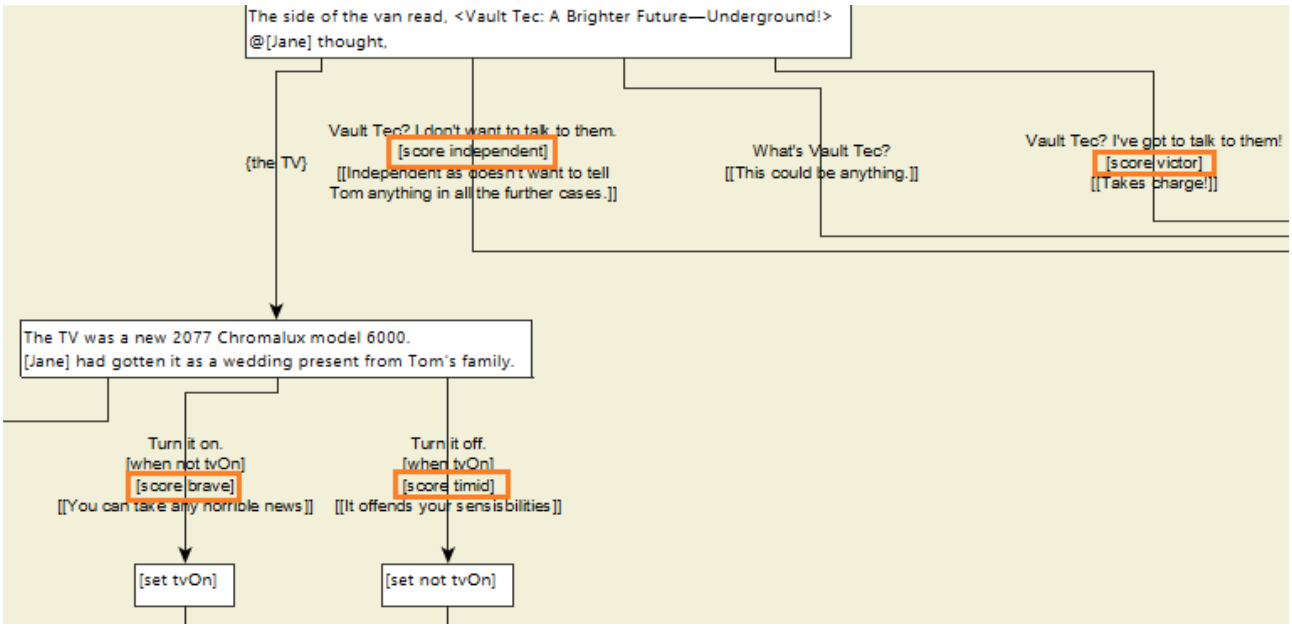
Game source code

The source code for the story is *.graphml* files produced by yWorks' graph editing program yEd. Here's the source code for the first page as seen in yEd. The [merge when …] arrows let you specify optional text, like whether the TV is on or off. The game interpreter edits the parts together:

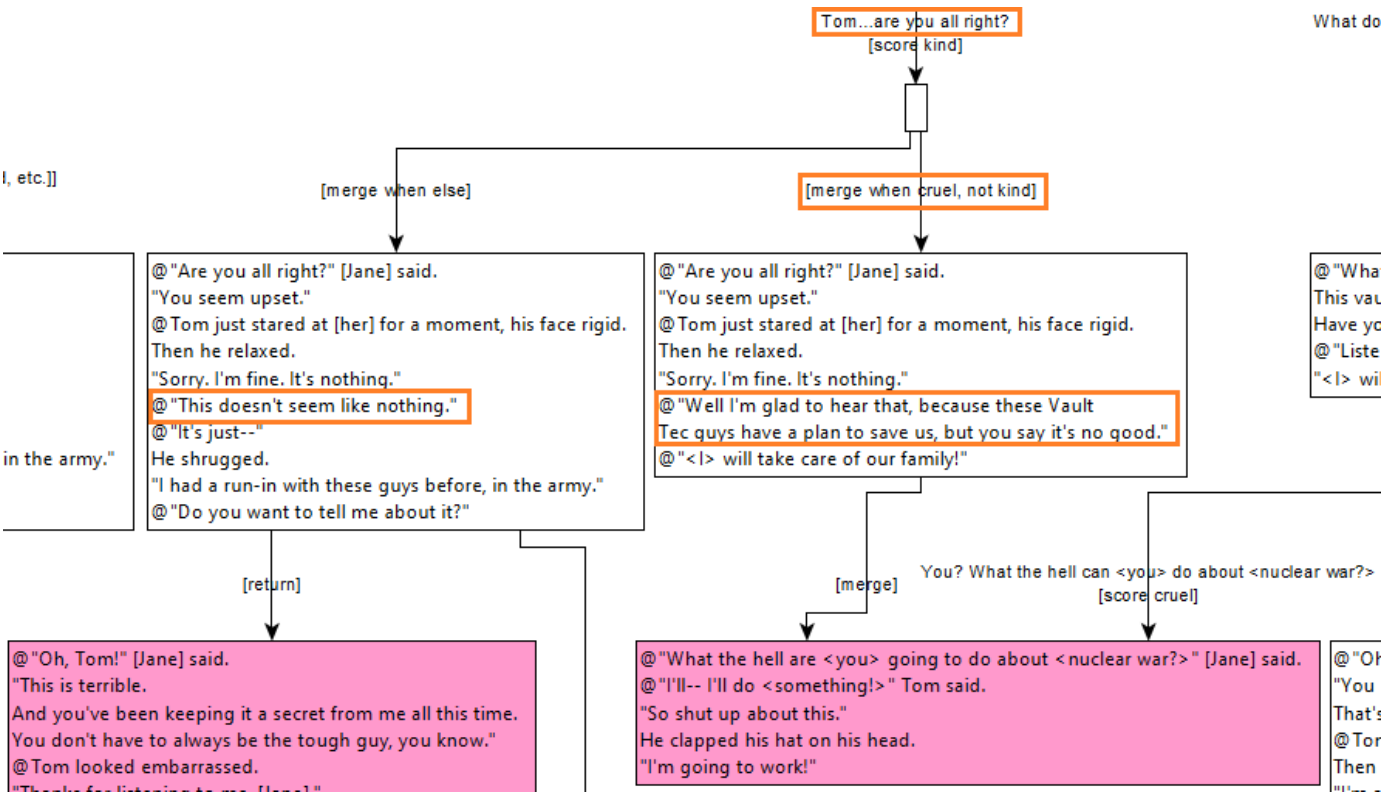


Personality scoring

In Bethesda's game Skyrim, if you shoot your bow, your archery skill goes up. Likewise, as you pick options in this game, it increments personality scores, such as independent, victor, brave, and timid:

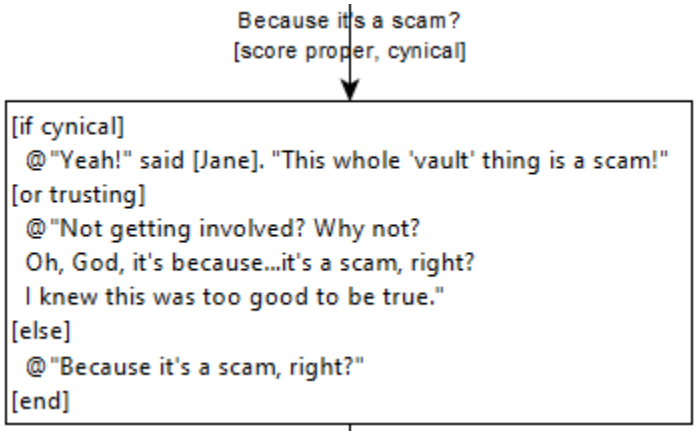


The scores affect what happens later in the game. For example, if you previously scored as cruel, when you ask your husband if he's all right, your character can't help but follow up with a critical remark, so he gets angry and leaves in a huff:



Conditional editing

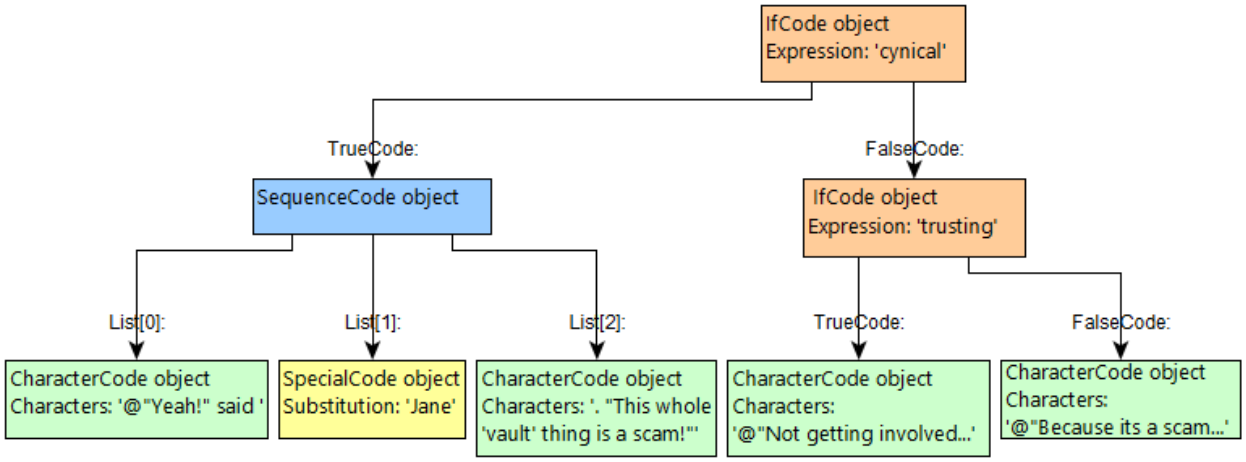
You can also do conditional editing within the text boxes:



Program source code example

The interpreter program is written in C#. As an example, here's how the code that implements conditional editing works:

- The program compiles the text box shown above into a tree of "code" objects in memory.
- There are different kinds of code objects: "if" objects, "character text" objects, "block sequence" objects, etc. Thus, the tree is heterogeneous, containing different kinds of objects:



- Even though the objects in the tree are different, they all implement the same "Traverse" function, which is defined in the abstract base class, "Code":

```
public abstract class Code
{
   // Traverse is the routine that allows other modules to execute the code.
   public abstract void Traverse(
     Func<Code, string, bool> examine,
     string originalSourceText);
}
```

- Every derived code class implements the Traverse function appropriately for itself. For example, IfCode calls the caller-provided "examine" function, then decides whether to traverse the TrueCode or FalseCode objects:

```
public class IfCode: Code
{
   private List<Expression> Expressions;
   public Code TrueCode { get; private set; }
   public Code FalseCode { get; private set; }

   public override void Traverse(
     Func<Code, string, bool> examine,
     string originalSourceText)
   {
      if (examine(this, originalSourceText))
         TrueCode.Traverse(examine, originalSourceText);
      else if (FalseCode != null)
         FalseCode.Traverse(examine, originalSourceText);
   }
```

SequenceCode has a list of code objects to traverse in sequence. Its Traverse function just iterates through their Traverse functions. There's no examination:

```
public class SequenceCode: Code
{
   // This is a sequence of code operations.
   private List<Code> Codes { get; set; } = new List<Code>();
   private string SourceText = null;

   public override void Traverse(
     Func<Code, string, bool> examine,
     string originalSourceText)
   {
      foreach (var code in Codes)
         code.Traverse(examine, originalSourceText);
   }
```

CharacterCode does no traversal. It just lets you examine the characters:

```
public class CharacterCode: Code
{
   public string Characters { get; private set; }

   public override void Traverse(
      Func<Code, string, bool> examine,
      string originalSourceText)
   {
      examine(this, originalSourceText);
   }
```

- Here's the client function that creates final text strings. It calls the abstract Traverse function, then checks what kind of concrete object it got:

```
   private string EvaluateText(
      Code value)
   {
      string accumulator = "";
      value.Traverse((code, originalSourceText) =>
      {
         switch (code)
         {
            case CharacterCode characterCode:
               accumulator += characterCode.Characters;
               break;
            case IfCode ifCode:
               return EvaluateConditions(ifCode.GetExpressions(), out var trace,
originalSourceText);
            case SpecialCode specialCode:
               accumulator += GetSpecialText(specialCode.Id, originalSourceText);
               break;
         }
         return true;
      });
      // Always returns an empty string if there is no useful text.
      return NormalizeText(accumulator);
   }
```

- Here's another routine that uses the same Traverse method just to execute [when] codes in conditional arrows (the tree structure is irrelevant for when codes):

```
   private (bool, bool) EvaluateWhen(
      Code topCode,
      out string outTrace)
   {
      string trace = "";
      // When there are no 'when' directives, it always succeeds.
      var allSucceeded = true;
      var hadWhen = false;
      topCode.Traverse((code, originalSourceText) =>
      {
         if (code is WhenCode whenCode)
         {
            hadWhen = true;
            if (!EvaluateConditions(whenCode.GetExpressions(), out trace,
originalSourceText))
               allSucceeded = false;
         }
         return true;
      });
      outTrace = trace;
      return (allSucceeded, hadWhen);
   }
```